Notes
CSC-362
Operating System

qxdeng1991

January 2025

Contents

1 Review of Computer System

A modern computer consists of several key hardware components working together to execute programs and process data. Broadly, we can identify four main parts:

1. **Processor (CPU)**
2. **Memory (RAM and storage hierarchy)**
3. **I/O Modules**
4. **System Bus**

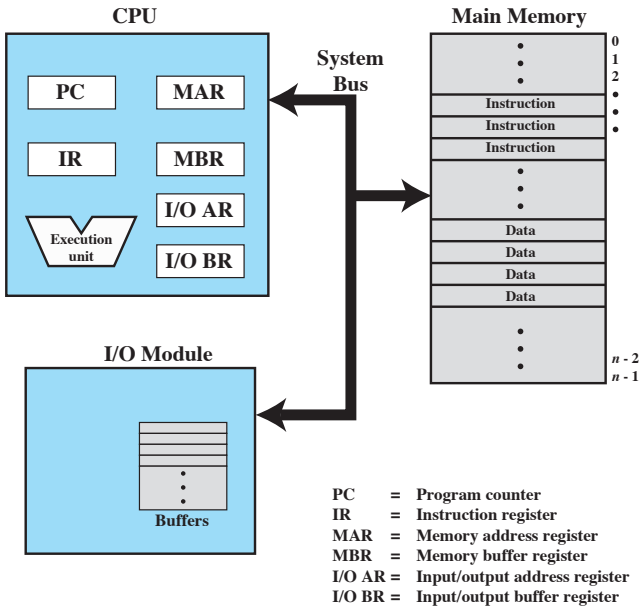


Figure 1: Computer System Overview

1.1 Processor (CPU)

The **Central Processing Unit (CPU)** is the "brain" of the computer. It interprets and executes instructions from programs, performing arithmetic, logical, and control tasks. Key subcomponents include:

- **Control Unit (CU)**: Fetches and decodes instructions, coordinating data movement within the CPU.
- **Arithmetic Logic Unit (ALU)**: Handles integer arithmetic (e.g. addition, subtraction) and logical operations (AND, OR, NOT).
- **Registers**: Small, fast storage locations for immediate data (including the Program Counter, Instruction Register, and general-purpose registers).
- **Cache**: A small, high-speed memory near or on the CPU for quickly accessing frequently used data (L1, L2, possibly L3).

Modern CPUs often feature multiple cores, pipelining, out-of-order execution, and other optimizations to increase throughput and performance.

1.1.1 Instructions Fetch and Execute

Instruction Fetch (IF). The process begins with the Program Counter (PC), a special CPU register that contains the memory address of the next instruction. The CPU sends a request to the main memory (usually RAM) using the address in the PC, and the memory subsystem responds by retrieving the instruction at that location. This binary-encoded instruction is then placed into the Instruction Register (IR). Finally, the CPU increments the PC so it points to the next instruction's address, readying the system for the subsequent fetch cycle.

Instruction Execute (EX) Once the instruction resides in the IR, the CPU's control unit decodes it, identifying the operation to be performed (e.g., addition, subtraction, load, branch) and determining any operands. If necessary, the CPU fetches these operands from memory or internal registers. It then executes the operation—such as adding two values—and writes the result to the appropriate destination (a register or a memory location). If the instruction modifies control flow (e.g., via a branch or jump), the Program Counter (PC) is updated with the new address, ensuring the next fetch occurs from the correct location. Considering a hepothetical machine below:

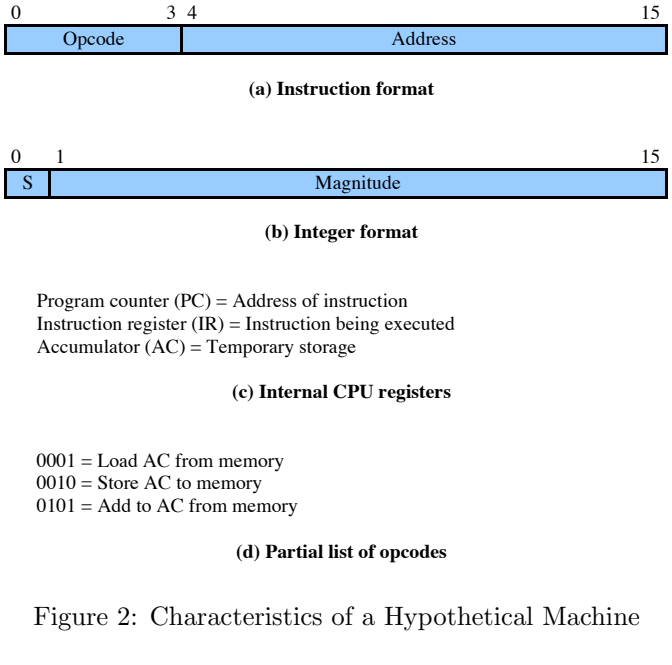


Figure 2: Characteristics of a Hypothetical Machine

Can you explain the instruction execution example below?

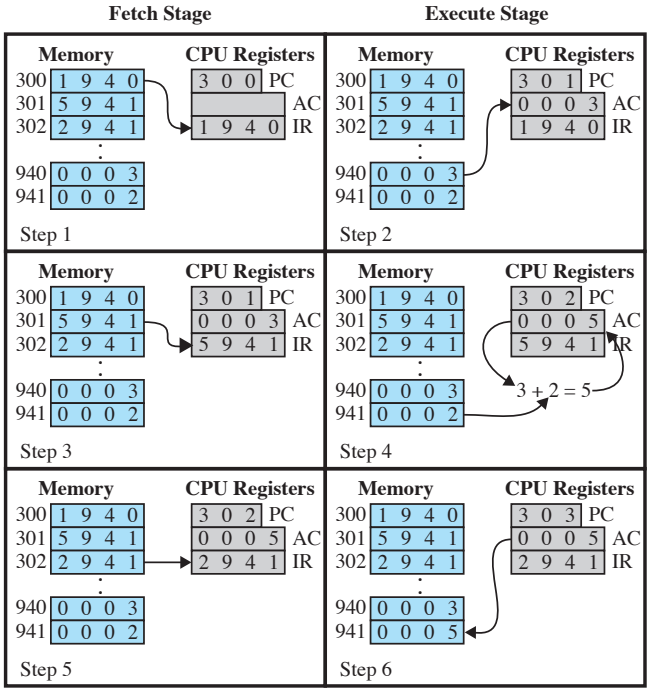


Figure 3: Instruction Execution Example

1.2 Interrupts

In a modern computer system, an **interrupt** is a signal that informs the processor (or CPU) that a particular event has occurred, requiring immediate attention or special handling. Rather than the CPU constantly polling devices to check whether they need service, interrupts allow devices (and sometimes software) to alert the CPU when they require action. This mechanism improves the overall efficiency of the system by avoiding needless busy-waiting.

1.2.1 Benefits of Interrupts

Interrupt-driven systems provide several advantages:

- **Efficiency:** Allows for efficient multitasking by enabling the CPU to switch between tasks.
- **Responsiveness:** Supports real-time processing by responding promptly to time-critical events.
- **Resource utilization:** Enables asynchronous communication with hardware devices.

- **Robustness:** Enhances system stability by handling errors and exceptional conditions.

1.2.2 Common Classes of Interrupts

Interrupts can be broadly categorized into the following classes:

1. **Hardware Interrupts (External Interrupts):** Generated by external hardware devices, such as keyboards, printers, network cards, or timers. For example, a network card might trigger an interrupt when it receives a new packet, or a timer interrupt allows the operating system to perform periodic tasks, such as context switching in multitasking environments.
2. **Software Interrupts (Traps or Exceptions):** Triggered by programs running on the CPU. These can occur for various reasons:
   - *System Calls* (e.g., a program requests an OS service).
   - *Exceptions* (e.g., division by zero, invalid memory access).

1.2.3 Example: Printer vs. Computer

Consider a personal computer (PC) whose CPU clock runs at 1 GHz (i.e., 1 billion cycles per second). Meanwhile, a typical consumer hard drive spins at 7200 rotations per minute (RPM).

Speed Comparison:

- *CPU speed:* 1 GHz = 1,000,000,000 cycles/second.
- *Hard drive speed:* 7200 RPM = $\frac{7200}{60}$ revolutions/second $\approx$ 120 revolutions/second.

Thus, the CPU is

$$\frac{1{,}000{,}000{,}000}{120} \approx 8{,}333{,}333$$

times *faster* than the physical rotation of the hard drive platter.

1.2.4 Program Timing

Now let's see two diagrams below. Could you explain it using your own words? Short IO means the IO waiting period is not longer enough to execute all code before next IO operation. Long IO means the IO waiting period is longer than the needed time to execute all the code before next IO operation.

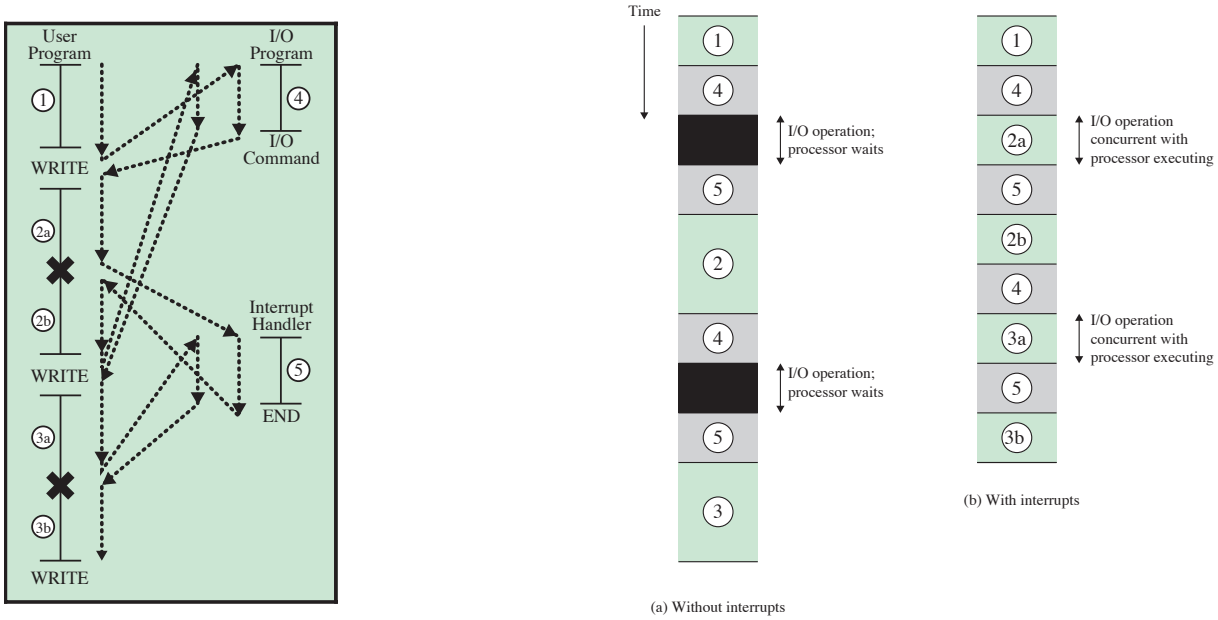


Figure 4: Program Timing: Short I/O Wait

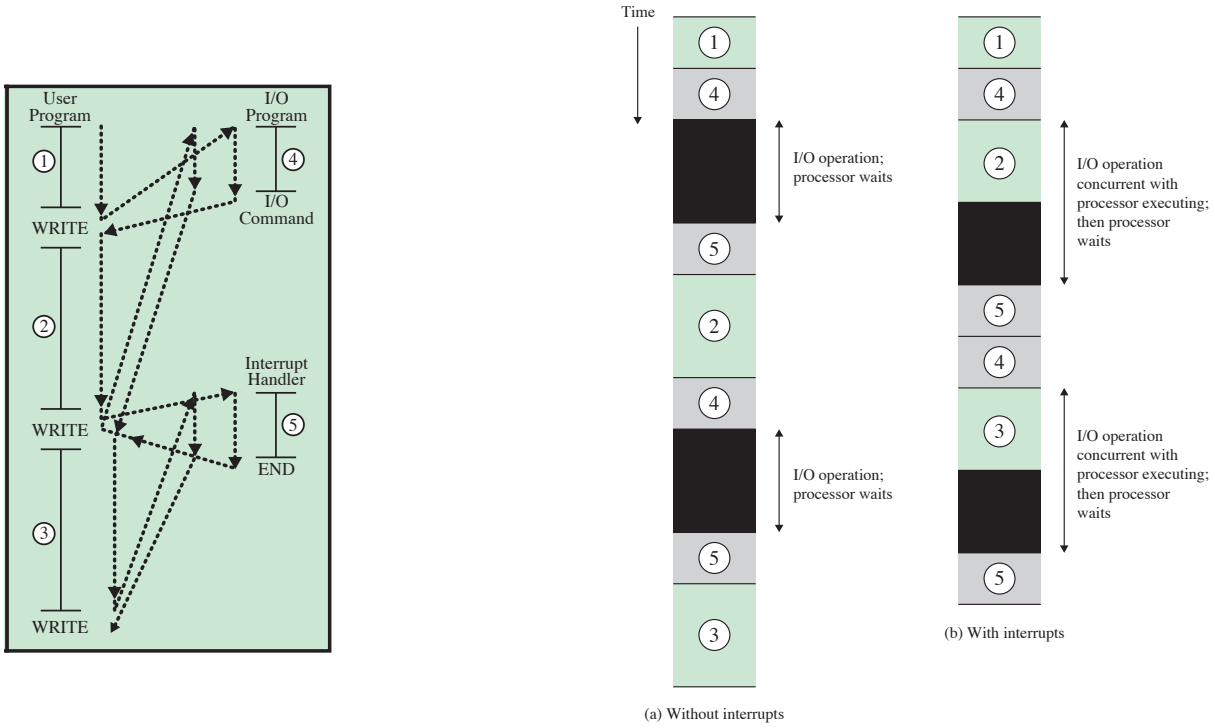


Figure 5: Program Timing: Long I/O Wait

1.2.5 Changes in memory and registers for an interrupt

In this case, a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction (N+1), a total of M words, are pushed onto the control stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.

The interrupt handler may now proceed to process the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.

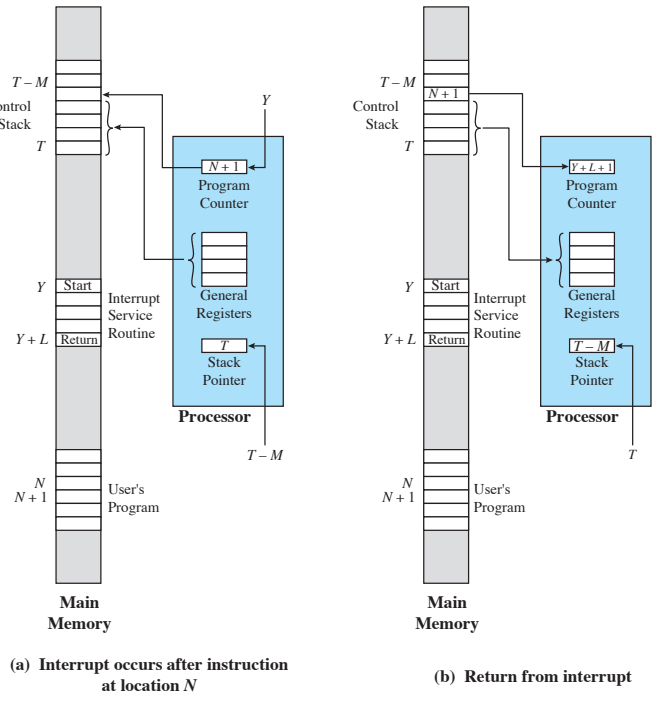


Figure 6: Changes in memory and registers for an interrupt

When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the

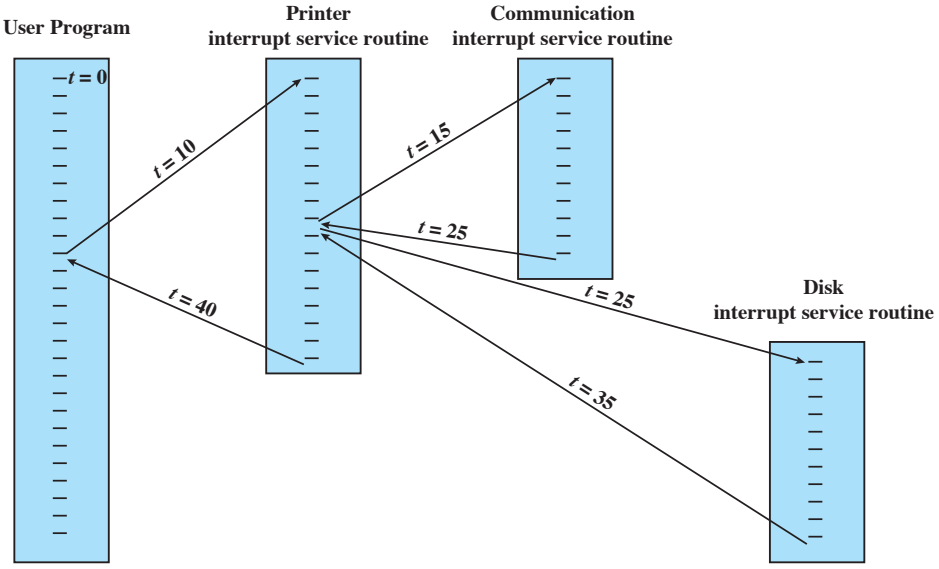
registers.The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

It is important to save all of the state information about the interrupted program for later resumption. This is because the interrupt is not a routine called from the program. Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program. Its occurrence is unpredictable.

1.2.6 Multiple interrupts.

Two approaches can be taken to dealing with multiple interrupts.

- The first is to disable interrupts while an interrupt is being processed. A disabled interrupt simply means that the processor ignores any new interrupt request signal.
- A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower–priority interrupt handler to be interrupted.



As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. A user program begins at t=0. At t=10, a printer interrupt occurs; user information is placed on the control stack and execution continues at the printer interrupt service routine (ISR). While this routine is still executing, at t=15 a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt request is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs (t=20). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

When the communications ISR is complete (t=25), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher priority disk interrupt and transfers control to the disk ISR. Only when that routine is complete (t=35) is the printer ISR resumed. When that routine completes (t=40), control finally returns to the user program.

1.3 Memory

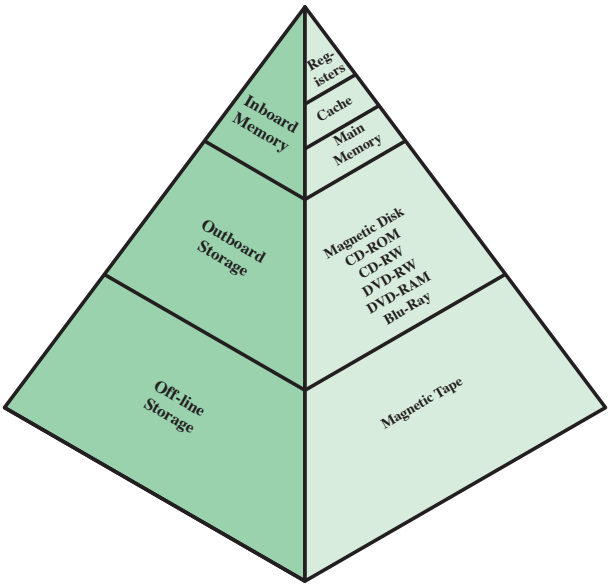


Figure 7: Memory Hierarchy

As one goes down the hierarchy, the following occur:

- Decreasing cost per bit
- Increasing capacity
- Increasing access time
- Decreasing frequency of access to the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is the decreasing frequency of access at lower levels.

We will focus on **cache** now, and later we will discuss more about the main memory external storage.

Principle of Locality: Programs tend to access data and instructions in localized regions, both in time (temporal locality) and in space (spatial locality). This behavioral pattern enables caching mechanisms to work efficiently, because the same or nearby data is likely to be reused soon or accessed in groups.

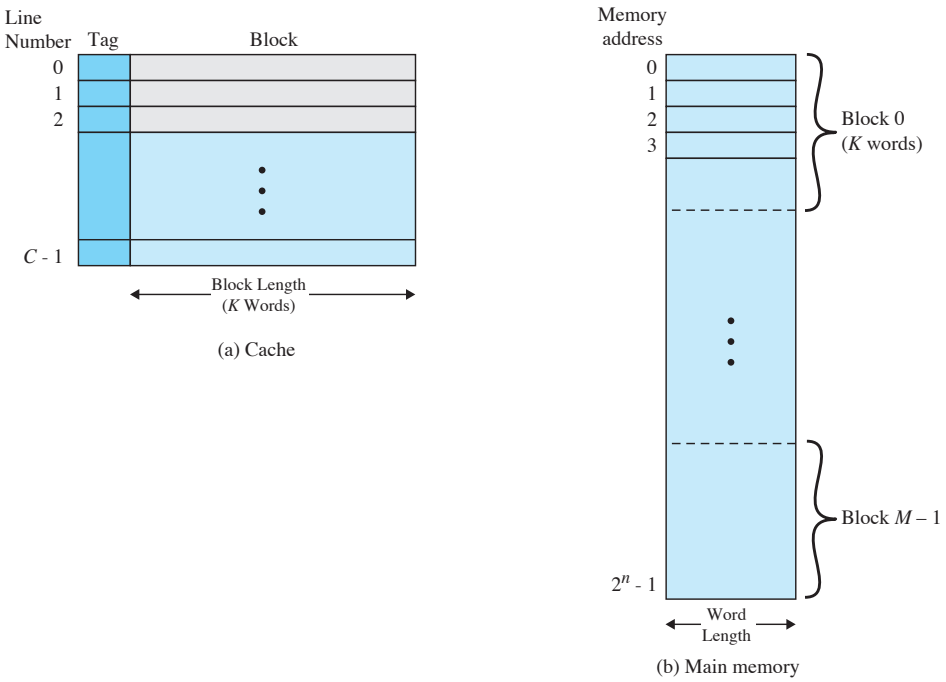


Figure 8: Cache Structure

Cache memory is invisible to the OS but works with memory management hardware. Every instruction cycle, the processor accesses memory at least once for the instruction, and often more for operands or results. Execution speed is limited by memory cycle time, which is much slower than processor speed. A fast, register-like main memory would solve this but is prohibitively expensive. Instead, we exploit locality by placing a small, fast cache between the processor and main memory.

When the processor reads a byte or word, it checks the cache first. If the data is there, it is delivered right away. Otherwise, a block of memory is fetched into the cache, and the requested data is then provided to the processor. Because of locality of reference, fetching a block to satisfy a single memory request often covers many upcoming requests for nearby addresses.

Main memory consists of up to 2^n addressable words, with each word having a unique n–bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = 2^n/K$ blocks. Cache consists of C slots (also referred to as lines) of K words each, and the number of slots is considerably less than the number of main memory blocks (C¡¡M). Some subset of the blocks of main memory resides in the slots of the cache. If a word in a block of memory that is not in the cache is read, that block is transferred to one of the slots of the cache.

1.4 I/O Modules

Programmed I/O. CPU directly controls data transfers by polling the device. **Pros:** Simple hardware. **Cons:** CPU constantly checks device status, wasting cycles.

Interrupt-Driven I/O. Device interrupts the CPU when ready. CPU handles I/O via an Interrupt Service Routine (ISR).
Pros: CPU does other work until device signals.
Cons: Still some CPU overhead in servicing interrupts.

Direct Memory Access (DMA). A dedicated DMA controller moves data between memory and the device, freeing the CPU.
Pros: Ideal for large/high-speed transfers; minimal CPU involvement.
Cons: Requires DMA hardware and careful management of bus access.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA) . The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows: When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written

The CPU delegates block transfers to the DMA module, which moves data between the I/O device and memory without passing through the CPU. Once the transfer is finished, the DMA module interrupts the CPU. The only CPU involvement is at the start and end of the transfer.

During DMA, the CPU may sometimes need the bus but must wait for the DMA module to finish transferring a word. While this slows the CPU slightly, DMA remains more efficient than interrupt-driven or programmed I/O for multi-word transfers.

Bus Arbitration. Both the CPU and the DMA controller share the system bus; when DMA is active, the CPU must wait if it needs bus access. This waiting can slow CPU operations.

Bus Contention Simultaneous bus requests from CPU and DMA cause contention. The CPU may be delayed until the DMA transfer completes its block, further slowing down CPU execution.

1.5 System Bus

The **system bus** is the set of wires or connections that allow the CPU, memory, and I/O modules to communicate. Historically, one main bus was used, but modern systems often have multiple specialized buses and high-speed point-to-point links. A traditional system bus usually has:

- **Data Bus**: Carries actual data being transferred.
- **Address Bus**: Specifies the memory location or I/O port involved in a transfer.

- **Control Bus**: Carries control signals (e.g. read/write signals, interrupts, clock lines).

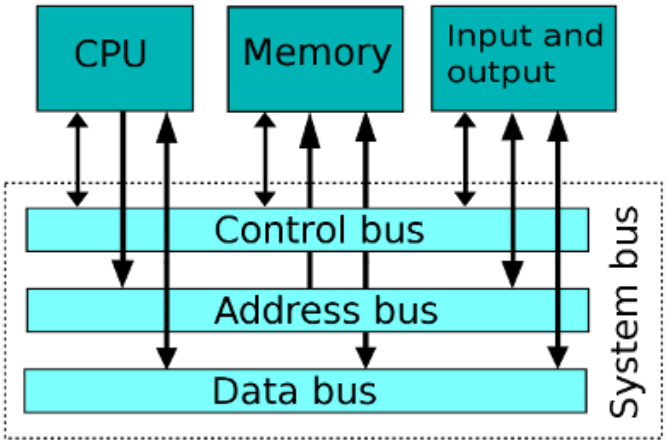


Figure 9: System bus

2 Operating System Overview

2.1 Computer System Structure

The end user typically does not focus on the intricacies of computer hardware and instead perceives a computer system as a collection of applications. These applications are developed by programmers. Rather than writing machine instructions directly, programmers rely on various libraries and utilities to facilitate tasks such as program creation, file management, and control of I/O devices. The operating system abstracts the hardware details from the programmer, offering a user-friendly interface for interacting with the system.

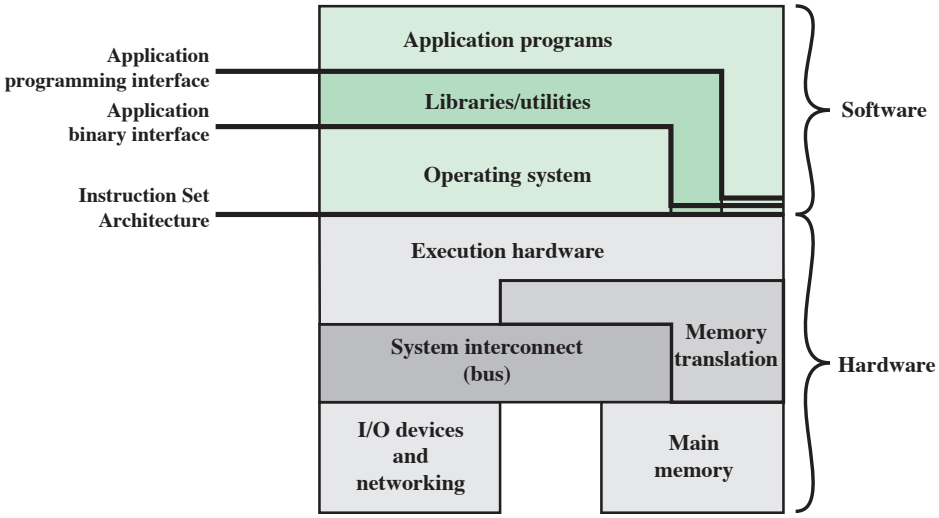


Figure 10: Computer Hardware and Software Structure

Briefly, the OS typically provides services in the following areas:

- **Program development**: such as editors and debuggers, to assist the programmer in creating programs.
- **Program execution**: The OS handles the creation, scheduling, and termination of processes and allocating CPU time, memory, and I/O devices to processes as needed.
- **Access to I/O devices**: The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.
- **Controlled access to files**: The OS manages who can read, write, or execute a file.
- **System access**: Through login credentials, ensuring only authorized users can access the system, and also protect against unauthorized external access by firewalls.
- **Error detection and response**: The OS maintains logs that record system events, which are useful for diagnosing and understanding system errors.
- **Accounting**: The OS keeps track of the use of system resources by different users or processes for billing or analysis purposes.

A typical computer system relies on three key interfaces:

- **Instruction Set Architecture (ISA):** The ISA serves as the interface between software and hardware, defining how software controls the hardware. Examples of ISAs include x86, ARM, MIPS, and PowerPC.
- **Application Binary Interface (ABI):** The ABI specifies how data structures and computational routines are accessed in machine code. This ensures that compiled programs function correctly with the operating system. Examples include the System V ABI for Unix and Linux, and the Microsoft Windows ABI.
- **Application Programming Interface (API):** The API provides programs access to hardware resources and system services via the user ISA, supplemented with high-level language (HLL) library calls.

And the interface roles are:

- The **ISA** is the foundational layer of system architecture, defining the interaction between the processor and software at the most fundamental level.
- The **ABI** ensures compatibility between compiled code and the operating system, managing procedure calls, data types, and system calls.
- The **API** abstracts system complexity, offering programmers pre-built functions and procedures to develop software applications more efficiently.

2.2 Evolution of Operating Systems

2.2.1 Serial Processing

With the earliest computers, from the late 1940s to the mid-1950s, programmers interacted directly with the computer hardware; there was no operating system (OS). These computers were operated from a console that consisted of display lights, toggle switches, some form of input device, and a printer. Programs, written in machine code, were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the display lights. If the program executed successfully, the output was printed.

These early systems had two primary challenges:

- **Scheduling:** Most installations used a hardcopy sign-up sheet to reserve computer time. Users might encounter issues or fail to finish within their allocated time, forcing them to stop before resolving their problems.
- **Setup time:** A single program, referred to as a *job*, involved several steps, including loading the compiler and the high-level language program (source program) into memory, saving the compiled program (object program), and then loading and linking the object program with common functions. Each step often required mounting or dismounting tapes or setting up card decks. If an error occurred, users typically had to restart the entire setup process. As a result, a significant amount of time was spent preparing the program for execution.

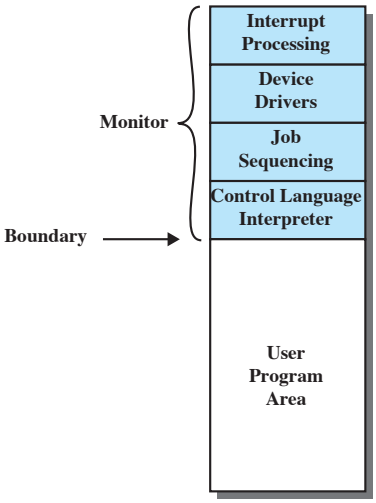
This mode of operation can be described as *serial processing*, highlighting the fact that users accessed the computer sequentially.

Figure 11: 1954, IBM established the 650 as its first mass-produced computer.

2.2.2 Simple Batch System

To improve system utilization, the concept of a batch operating system (OS) was introduced. It is believed that the first batch OS, and the first OS of any kind, was developed in the mid-1950s by General Motors for use on an IBM 701, which rented for $15,000 per month.

The central idea behind the batch-processing system is the use of software known as the *monitor*. With this type of OS, users no longer have direct access to the processor. Instead, users submit their jobs on cards or tape to a computer operator, who organizes the jobs sequentially into batches and loads the entire batch onto an input device for the monitor to process. Each program is designed to return control to the monitor upon completion, allowing the monitor to automatically load and execute the next program in the batch.



The monitor governs the sequence of operations. To achieve this, a significant portion of the monitor must reside in main memory at all times and remain available for execution. This portion is known as the *resident monitor*.

The remainder of the monitor consists of utilities and common functions, which are loaded as subroutines into the user program at the start of any job that requires them. The monitor reads jobs sequentially from the input device, typically a card reader or magnetic tape drive. Each job is placed into the user program area as it is read, and control is transferred to the job.

Upon completion, the job returns control to the monitor, which immediately reads and executes the next job. The output generated by each job is directed to an output device, such as a printer, for the user to retrieve.

Nothing is Free. Processor time alternates between execution of user programs and execution of the monitor, and some main memory is now given over to the monitor. Despite overhead, the simple batch system improves utilization of the computer.

2.2.3 Multiprogrammed Batch Systems

Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor.

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

The calculation concerns a program that processes a file of records and performs, on average, 100 machine instructions per record. In this example, the computer spends over 96% of its time waiting for I/O devices to finish transferring data to and from the file.

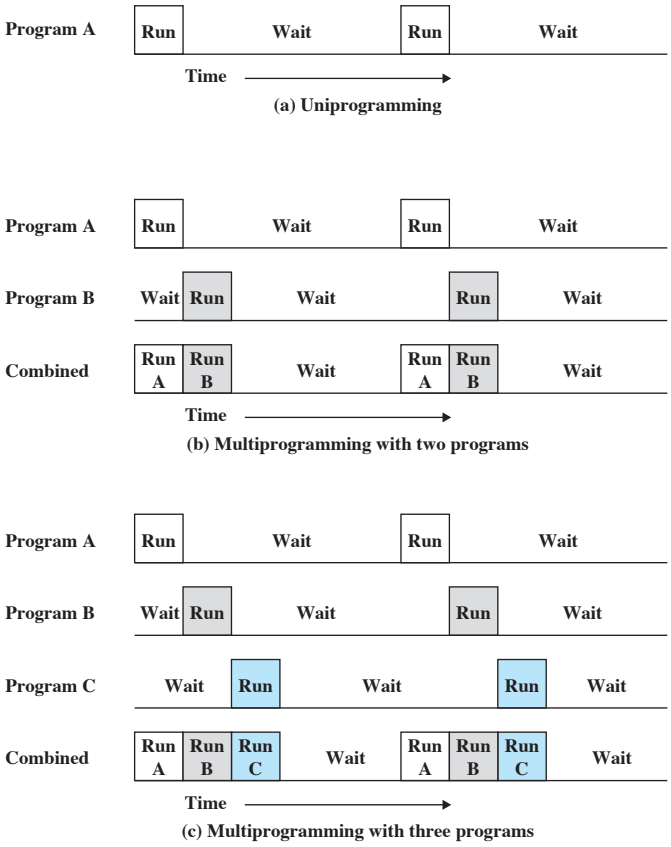
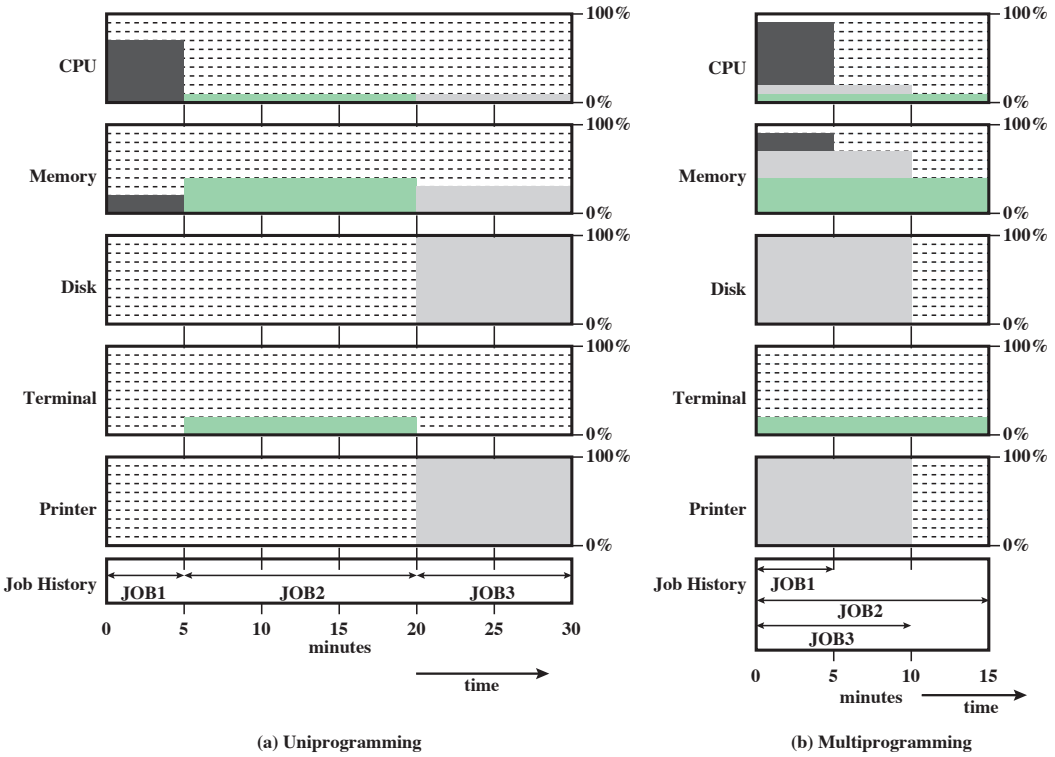


Figure 12: Multiprogramming

From the figure above, we can see that multiprogramming, or multitasking, significantly improves the system utilization. Consider a computer with 250 Mbytes of available memory (not used by the OS), a disk, a terminal, and a printer.

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Now suppose that the jobs are run concurrently under a multiprogramming OS. Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer (assuming that JOB2 and JOB3 are allotted enough processor time to keep their input and output operations active). From the CPU's view, JOB1 need to 70% time of 5 mins to do calculation. Mean time, CPU can run JOB2 and JOB3 instructions in other 30% CPU idle time.



2.2.4 Time-Sharing Systems

Just as multiprogramming enables the processor to handle multiple batch jobs simultaneously, it can also be utilized to manage multiple interactive jobs. In this context, the technique is referred to as *time sharing*, as processor time is distributed among multiple users.

In a time-sharing system, multiple users access the system concurrently through terminals. The operating system interleaves the execution of each user's program, allocating a short burst or quantum of computation to each. If there are n users actively requesting service at the same time, each user effectively receives, on average, $\frac{1}{n}$ of the system's capacity, excluding overhead from the operating system.

Despite this division of resources, the relatively slow reaction time of humans ensures that, in a well-designed system, the response time experienced by users is comparable to that of a dedicated computer.

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Figure 13: Batch Multiprogramming versus Time Sharing

Compatible Time-Sharing System (CTSS) was one of the earliest time-sharing operating systems, developed at MIT by a group known as Project MAC. The system was initially created for the IBM 709 in 1961.

The computer it ran on featured 32,000 36-bit words of main memory, with approximately 5000 words occupied by the resident monitor. CTSS utilized a technique called *time slicing*, where the system clock generated interrupts at a rate of approximately one every 0.2 seconds.

At each clock interrupt, the operating system regained control and could reassign the processor to another user. This mechanism ensured that the current user was preempted at regular intervals, and a new user was loaded into the system.

To facilitate this process, the status of the preempted user's program and data was preserved by writing them out to disk before loading the next user's programs and data. When the preempted program was granted a turn again, its code and data were restored into main memory, allowing execution to resume seamlessly.

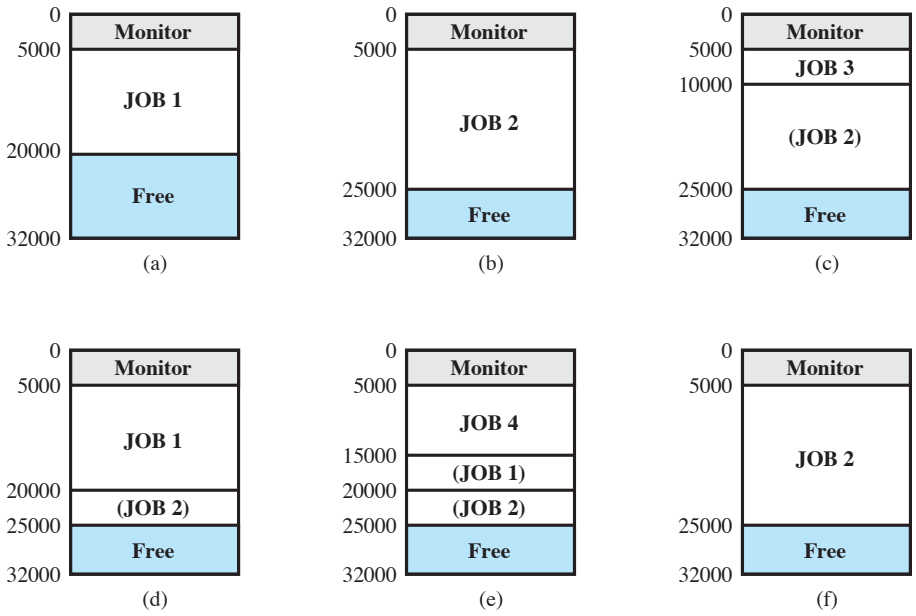


Figure 14: CTSS Operation

To minimize disk traffic, user memory was only written out when the incoming program would overwrite it.

2.3 Last Word Before Start

let's watch this video together: *Linux Boot Process Explained*. This is a pretty good video to show several important components within a modern Linux system.

3 System Call

A **system call** is a mechanism that allows a user program to interact with the operating system (OS). It serves as the interface between a process (running program) and the OS, enabling the program to request services like file operations, process control, or communication.

- System calls are typically implemented using a library in the user space, such as the C standard library (`libc`), to provide a convenient API.
- They transition the CPU from **user mode** to **kernel mode** to safely execute privileged operations.

3.1 Kernel Mode and User Mode

The CPU operates in two distinct privilege levels:

- **User Mode (Ring 3)**:
  - Executed by user-level applications.
  - Access to hardware and memory is restricted.
  - Privileged instructions are not allowed.
- **Kernel Mode (Ring 0)**:
  - Executed by the operating system.
  - Full access to hardware and all memory.
  - Allows execution of privileged instructions.

3.2 Mode Transitions

- A transition from user mode to kernel mode occurs when a system call or an exception is triggered.
- The CPU changes privilege levels and executes the appropriate handler in the operating system.
- After completing the operation, control is returned to user mode.

Kernel mode is the state in which the operating system code executes, while user mode is the state in which application program code executes. Both operating system kernel code and application program code are loaded into memory and executed. Therefore, kernel mode code and user mode code are placed in different memory regions.

Again, there are four privilege levels, numbered from 0 to 3, with 0 being the most privileged (kernel mode) and 3 being the least privileged (user mode).

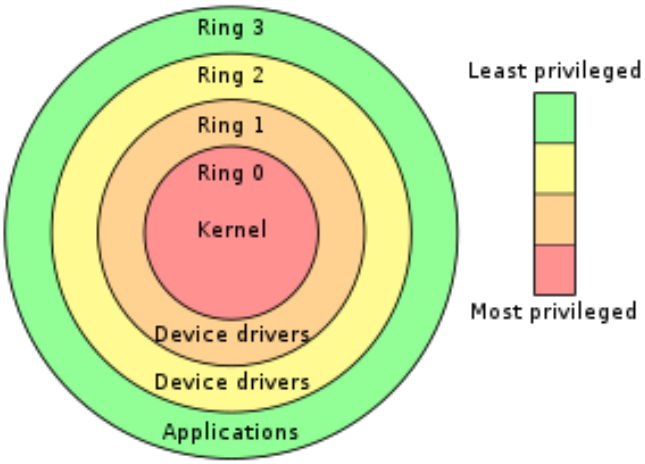


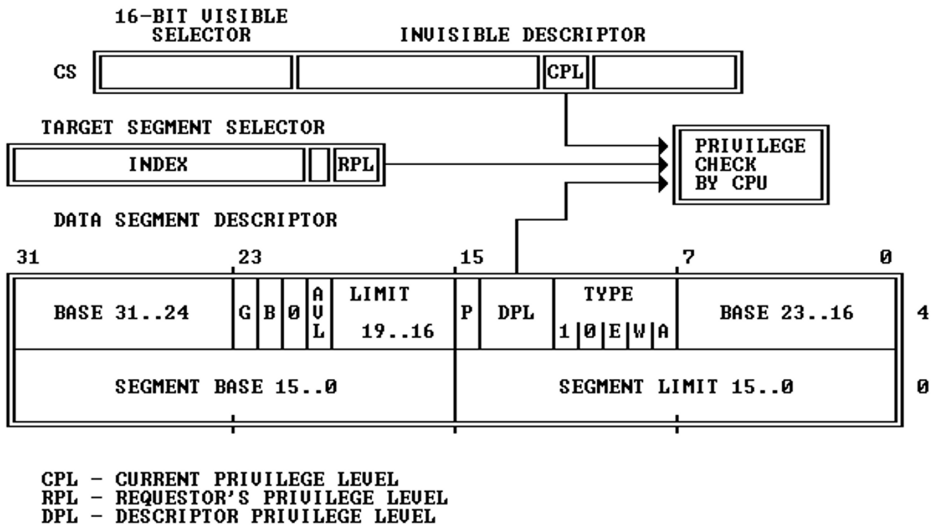
Figure 15: Mode Rings

3.3 privilege Check

Privilege levels are managed using three key components:

- **Current Privilege Level (CPL)**: is a 2-bit field that indicates the privilege level of the currently executing code.
- **Requested Privilege Level (RPL)**: Specified in segment selectors when accessing memory or resources. It represents the privilege level requested by the program.

- **Descriptor Privilege Level (DPL)**: Indicates the privilege level required to access a specific resource, as defined in the segment descriptor, providing a static access control mechanism.



Access to resources is granted if:

$$\text{Effective Privilege Level (EPL)} = \max(\text{CPL}, \text{RPL}) \leq \text{DPL}$$

Suppose a process in **User Mode** (CPL = 3) wants to access a memory segment defined by the operating system. The memory segment descriptor is stored in the **Global Descriptor Table (GDT)** and specifies a **DPL (Descriptor Privilege Level)** that determines which privilege levels can access the segment.

$$\text{EPL} = \max(\text{CPL}, \text{RPL})$$

The system computes the **Effective Privilege Level (EPL)** by taking the maximum of **CPL** (current privilege level) and **RPL** (requested privilege level from the selector).

Access is allowed only if:

$$\text{EPL} \leq \text{DPL}$$

- **CPL = 3**: The current privilege level of the process (User Mode).
- **RPL = 2**: The requested privilege level specified in the segment selector.
- **DPL = 1**: The descriptor privilege level of the memory segment.

$$\text{EPL} = \max(\text{CPL}, \text{RPL}) = \max(3, 2) = 3$$

$$\text{EPL} = 3 \quad \text{is greater than} \quad \text{DPL} = 1$$

The access is **denied** because:

$$\text{EPL} = 3 > \text{DPL} = 1$$

The process in **User Mode** cannot access this segment because its **Effective Privilege Level (EPL)** of 3 does not meet the descriptor's requirement of 1. How can a user program invoke system calls that belong to the system kernel? What mechanisms are in place to ensure the safety of the kernel memory, preventing the user program from causing any harm?

3.4 How Does System Call Work in Old Linux Kernel 0.12?

In Linux 0.11, system calls are implemented using an **Interrupt Descriptor Table (IDT)** entry for interrupt vector `0x80`. When a user program executes the `int 0x80` instruction, it triggers this interrupt, which transitions the CPU from **user mode (CPL = 3)** to **kernel mode (CPL = 0)**.

- **Interrupt Descriptor Table (IDT)**: The kernel sets up entry `0x80` in the IDT to point to the system call handler. The `int 0x80` entry is initialized during the kernel startup in the `trap_init` function in the `trap.c` file:

```c
// trap.c (Linux 0.11)
void trap_init(void) {
    // Set the system call interrupt (int 0x80)

    set_trap_gate(0x80, &system_call);
}
```

  - `set_trap_gate` is a macro that sets up an entry in the IDT for a trap gate.
  - `0x80` is the interrupt vector for system calls.
  - `&system_call` is the address of the system call handler function.

- **System Call Handler**: The handler is a piece of kernel code that processes the syscall request. The `system_call` function is the entry point for all system calls. It saves the CPU state, validates the system call number, and dispatches the request to the appropriate system call:

```assembly
// system_call.s (Linux 0.11, Assembly code)
.globl system_call
system_call:
    cmpl $NR_syscalls-1, %eax    # Check if syscall number is valid
    ja bad_syscall               # If invalid, jump to bad_syscall
    push %ds                     # Save data segment
    push %es                     # Save extra segment
    push %fs                     # Save fs segment
    pushl %eax                   # Save syscall number
    movl $0x10, %eax             # Load kernel data segment selector
    mov %ax, %ds                 # Set kernel data segment
    mov %ax, %es                 # Set kernel extra segment
    mov %ax, %fs                 # Set kernel fs segment
    call *sys_call_table(,%eax,4) # Dispatch to the syscall
    popl %eax                    # Restore syscall number
    pop %fs                      # Restore fs segment
    pop %es                      # Restore es segment
    pop %ds                      # Restore ds segment
    iret                         # Return to user mode
```

  - **Input:**
    - `%eax`: The syscall number.
    - Other registers: Arguments for the system call.
  - **Process:**
    - Validate the syscall number.
    - Save the user-space context.
    - Set up kernel data segments.
    - Dispatch the syscall using the `sys_call_table`.
    - Restore user program

- **System Call Table**: A table of function pointers maps syscall numbers to the actual system call implementations. It maps the syscall number to the corresponding kernel function:

```c
// system_call.c (Linux 0.11)
extern int sys_setup();
extern int sys_exit();
extern int sys_fork();
extern int sys_read();
extern int sys_write();

fn_ptr sys_call_table[] = {
    sys_setup,    // 0
    sys_exit,     // 1
```

```
    sys_fork,     // 2
    sys_read,     // 3
    sys_write,    // 4
    // More system calls...
};
```

- **User Program Invocation**: The user program uses the `int 0x80` instruction to make a system call. Here's an example of the `sys_write` system call, which writes data to a file descriptor:

```
// sys_write.c (Linux 0.11)
int sys_write(unsigned int fd, const char *buf, int count) {
    struct file *file;
    int written = 0;

    if (fd >= NR_OPEN || !(file = current->filp[fd]))
        return -EBADF;  // Invalid file descriptor
    if (!file->f_op->write)
        return -EINVAL; // Operation not supported
    written = file->f_op->write(file, buf, count);
    return written;
}
```

It is clearer from the assembly view:

```
section .data
    message db "Hello, Linux 0.11!", 0xA
    len equ $ - message

section .text
    mov eax, 4          ; syscall number for sys_write
    mov ebx, 1          ; file descriptor (stdout)
    mov ecx, message    ; pointer to the message
    mov edx, len        ; length of the message
    int 0x80            ; trigger system call

    mov eax, 1          ; syscall number for sys_exit
    xor ebx, ebx        ; exit code 0
    int 0x80            ; trigger system call
```

In user programs, the `int 0x80` instruction is used to invoke a system call.

3.5 fork()

The `fork()` system call is one of the most fundamental and powerful features of Unix-like operating systems, including Linux. It allows a process to create a new process (called the **child process**) that is an exact copy of the original process (called the **parent process**). After `fork()` is called, both processes run concurrently, and the child process inherits the state of the parent process, including memory, file descriptors, and program counter.

The return value of the fork function is decided by the process it is in:

- In the **parent process**, `fork()` returns the **process ID (PID)** of the child process.
- In the **child process**, `fork()` returns `0`.
- If `fork()` fails, it returns `-1`.

```
#include <stdio.h>
#include <unistd.h> // for fork()
#include <sys/types.h> // for pid_t

int main() {
    pid_t pid;

```

```
    // Create a child process
    pid = fork();

    if (pid < 0) {
        // fork() failed
        fprintf(stderr, "Fork failed!\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Hello from the child process! (PID: %d)\n", getpid());
    } else {
        // Parent process
        printf("Hello from the parent process! (PID: %d, Child PID: %d)\n", getpid(), pid);
    }

    return 0;
}
```

- The child process is an exact copy of the parent process at the time of the `fork()` call.
- Both processes continue execution from the point where `fork()` was called.
- Changes made in one process (parent or child) do not affect the other process.

3.6 Deep Dive into `fork()` in Old Linux Kernel 0.12

Linux kernel versions 0.12 were early versions of the Linux kernel, released in 1992, respectively. These kernels were much simpler than modern Linux kernels, but they already included the `fork()` system call.

The fork() system call is registered in the system call table:

```
// kernel/system_call.c
fn_ptr sys_call_table[] = {
    sys_setup,
    sys_exit,
    sys_fork, // Entry for fork()
    sys_read,
    sys_write,
    // ...
};
```

The sys_fork() function is the entry point for the fork() system call. It is implemented in sys_call.s:

```
_sys_fork:
    call _find_empty_process # get last_pid (kernel/fork.c, 143)
    testl %eax,%eax # pid in eax, if negative then ret.
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process # copy_process()(kernel/fork.c, 68).
    addl $20,%esp # discard.
1: ret
```

This function calls two other functions (*_find_empty_process* and *_copy_process*) defined in `kernel/fork.c`. We will focus on *_copy_process* here:

```

/*
 * Ok, this is the main fork-routine. It copies the system process
```

```
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in its entirety.
 */
// Copy process information.
// The parameters of this function start from the handler that enters the system-call interrupt
// INT 0x80, and until this function is called (sys_call.s line 231), These registers are
// gradually pushed into the kernel state stack of the process. The values (parameters) that
// are pushed onto the stack in sys_call.s file include:
// 1) The user stack ss, esp, eflags, and the return addresses cs, eip pushed when executing
// the INT instruction;
// 2) ds, es, fs, and edx, ecx, ebx pushed on stack on lines 85-91 just after entering;
// 3) The return address pushed when the sys_fork() in sys_call_table is called on line 97
// (represented by none);
// 4) On lines 226-230, gs, esi, edi, ebp, eax(nr) are pushed before calling copy_process().
// Among them, nr is the task array item index assigned by calling find_empty_process().
int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
                 long ebx, long ecx, long edx, long orig_eax,
                 long fs, long es, long ds,
                 long eip, long cs, long eflags, long esp, long ss) {
    struct task_struct *p;
    int i;
    struct file *f;

    // The code first allocates memory for the new task structure (if the allocation fails, it returns
    // an error code and exits). Then put the new task structure pointer into the nr item of the
    // task array. Where nr is task no, which is returned by the previous find_empty_process().
    // Then copy the contents of the task structure of the current process to the beginning of the
    // memory page p just applied.
    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;
    task[nr] = p;
    *p = *current; /* NOTE! this doesn't copy the supervisor stack */

    // Then some modifications are made to the content of the copied process structure as the task
    // structure of the new process. The state of the new process is first set to an uninterruptible
    // wait state to prevent the kernel from scheduling its execution. Then set the process ID pid
    // of the new process and initialize the process run time slice value equal to its priority
    // value (typically 15 ticks). Then reset the signal bitmap, the alarm timer, the session leader
    // flag, the running time statistics in the kernel and user mode, and the system time start_time
    // at which the process starts running.
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid; // new pid obtained from find_empty_process()
    p->counter = p->priority; // run time slice value (number of ticks).
    p->signal = 0; // signal bitmap.
    p->alarm = 0; // alarm timer.
    p->leader = 0; /* process leadership doesn't inherit */
    p->utime = p->stime = 0; // user state and core state running time.
    p->cutime = p->cstime = 0; // childs user state and core state running time.
    p->start_time = jiffies; // the start time of the process (current time ticks).

    // Now modify the task status section TSS content (see the description after the program list).
    // Since the system allocates 1 page of memory to the task structure p, esp0 = (PAGE_SIZE +
    // (long) p) causes esp0 to point exactly to the top of the page. ss0:esp0 is used as a stack
    // for the program to execute in kernel mode.
    // In addition, as we already know in Chapter 5, each task has two segment descriptors in the
    // GDT table: one is the task's TSS segment descriptor and the other is the task's LDT table
    // segment descriptor. The statement on line 110 is to store the selector of the LDT segment
    // descriptor of this task into the TSS segment. When performing task switch, the CPU
    // automatically loads the LDT segment selector from the TSS into the LDTR register.
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p; // Task kernel state stack pointer.
    p->tss.ss0 = 0x10; // selector for the kernel state stack.
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0; // This is why the new process will return 0.
```

```
70      p->tss.ecx = ecx;
71      p->tss.edx = edx;
72      p->tss.ebx = ebx;
73      p->tss.esp = esp;
74      p->tss.ebp = ebp;
75      p->tss.esi = esi;
76      p->tss.edi = edi;
77      p->tss.es = es & 0xffff; // The segment register has only 16 bits.
78      p->tss.cs = cs & 0xffff;
79      p->tss.ss = ss & 0xffff;
80      p->tss.ds = ds & 0xffff;
81      p->tss.fs = fs & 0xffff;
82      p->tss.gs = gs & 0xffff;
83      p->tss.ldt = _LDT(nr); // The selector for the task LDT descriptor (in GDT).
84      p->tss.trace_bitmap = 0x80000000; // (High 16 bits are valid).
85
86      // If the current task uses a coprocessor, its context is saved. The instruction CLTS is used
87      // to clear the task exchange flag TS in the control register CR0. The CPU sets this flag whenever
88      // a task switch occurs. This flag is used to manage the math coprocessor: if this flag is set,
89      // then each ESC instruction will be caught (Exception 7). If the coprocessor presence flag
90      // MP is also set, the WAIT instruction will also be captured. Therefore, if a task switch occurs
91      // after an ESC instruction begins execution, the contents of the coprocessor may need to be
92      // saved before executing the new ESC instruction. The capture handler will store the contents
93      // of the coprocessor and resets the TS flag. The instruction FNSAVE is used to save all states
94      // of the coprocessor to the memory area specified by the destination operand (tss.i387).
95      if (last_task_used_math == current)
96          __asm__("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));
97
98      // Next, the process page table is copied, that is, the base address and the limit in the new
99      // task code and data segment descriptors are set, and the page table is copied. If an error
100     // occurs (the return value is not 0), the corresponding entry in the task array is reset and
101     // the memory page allocated for the new task structure is released.
102     if (copy_mem(nr, p)) { // The return is not 0, indicating an error.
103         task[nr] = NULL;
104         free_page((long) p);
105         return -EAGAIN;
106     }
107
108     // Because the newly created child will share the open file with the parent process, if the
109     // file is opened in the parent, the number of times the corresponding file is opened needs
110     // to be increased by one. For the same reason, you need to increase the number of reference
111     // of the i nodes of the current process (parent process) by 1 for pwd, root, and executable.
112     for (i = 0; i < NR_OPEN; i++)
113         if (f = p->filp[i])
114             f->f_count++;
115     if (current->pwd)
116         current->pwd->i_count++;
117     if (current->root)
118         current->root->i_count++;
119     if (current->executable)
120         current->executable->i_count++;
121     if (current->library)
122         current->library->i_count++;
123
124     // Subsequently, the new task TSS and LDT segment descriptors entries are set in the GDT. The
125     // limit of both segments is set to 104 bytes (see include/asm/system.h, lines 52-66). Then
126     // set the relationship list pointers between the processes, that is, insert the new process
127     // into the child process linked list of the current process. That is, the parent process of
128     // the new process is set to the current process, and the latest child process pointer p_cptr
129     // and the young sibling process pointer p_ysptr of the new process are set to be empty. Then
130     // let the new process's buddy process pointer p_osptr be set equal to the parent's latest child
131     // pointer. If the current process has other child processes, let the young sibling pointer
132     // p_yspter of the neighboring process point to the new process, and let child pointer of the
133     // current process point to this new process. Then set the new process to the ready state and
134     // finally return the new process id. set_tss_desc() and set_ldt_desc() are defined in file
135     // include/asm/system.h, 52-66. "gdt+(nr<<1)+FIRST_TSS_ENTRY" is the address of the TSS
```

```
136     // descriptor of the task nr in the global table. Since each task occupies 2 items in the GDT
137     // table, '(nr<<1)' should be included in the above formula. Note that the task register TR
138     // is automatically loaded by the CPU during task switching.
139     set_tss_desc(gdt + (nr << 1) + FIRST_TSS_ENTRY, &(p->tss));
140     set_ldt_desc(gdt + (nr << 1) + FIRST_LDT_ENTRY, &(p->ldt));
141     p->p_pptr = current; // parent pointer.
142     p->p_cptr = 0;
143     p->p_ysptr = 0;
144     p->p_osptr = current->p_cptr; // old sibling.
145     if (p->p_osptr) // if old sibling exist, its young
146         p->p_osptr->p_ysptr = p; // sibling points to this new process.
147     current->p_cptr = p; // I am the current new child.
148     p->state = TASK_RUNNING; /* do this last, just in case */
149     return last_pid;
150 }
```

4 Process

4.1 What is a Process?

Here are different definitions of the term "process":

- A program that is currently being executed
- A running instance of a program on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity defined by the execution of a sequence of instructions, its current state, and a related set of system resources

A process can also be thought of as an entity made up of several components. Two key elements of a process are the **program code** (which might be shared by other processes running the same program) and the **data** that is associated with that code. When the processor starts executing this program code, we refer to the **executing entity** as a process.

4.2 Process Creation

When a new process is introduced, the operating system (OS) initializes the necessary data structures to manage it and allocates memory space accordingly. Four common events trigger the creation of a process:

- **Batch Environment:** A process is created when a job is submitted.
- **Interactive Environment:** A new user login initiates a process.
- **Application Request:** The OS generates a process to handle specific tasks, such as printing a file.
- **Process Creation by Another Process:** A running application may spawn a new process to handle a particular task concurrently.

Traditionally, the OS managed all process creation internally, without direct user or application involvement. This remains common in many modern systems. However, allowing one process to create another can be highly beneficial. For instance, an application generating large amounts of data may create a separate process to organize and store these data for later analysis. The new process runs in parallel with the original and activates periodically when new data become available.

4.3 Dive into Process Creation in Old Linux 0.11/0.12

In Linux Kernel 0.12, the boot process follows a structured sequence, from loading the kernel into memory to transitioning to **task 0 (Process 0)** and then launching **init (Process 1)**.

The bootloader is responsible for:

- Loading the kernel into memory.
- Entering **protected mode**.
- Jumping to `init/main.c` to begin system initialization.

The `main()` function in `init/main.c` performs:

- Memory management initialization.
- Interrupt handling setup.
- Block and character device initialization.
- Hard disk and floppy disk support configuration.

```
// The following is the initialization of all aspects of the kernel. It is better to
// follow into each init function when reading.
//line 153-163
mem_init(main_memory_start,memory_end); // Main mem area (mm/memory.c,443)
trap_init(); // trap gate (hw vector) init (kernel/traps.c, 181)
blk_dev_init(); // block dev init (blk_drv/ll_rw_blk.c, 210)
chr_dev_init(); // char dev init (chr_drv/tty_io.c, 402)
tty_init(); // tty init (chr_drv/tty_io.c, 105)
time_init(); // setting boot time (line 92)
sched_init(); // scheduler init (kernel/sched.c, 385)
buffer_init(buffer_memory_end); // buffer init(fs/buffer.c, 348)
```

```
hd_init(); // harddisk init (blk_drv/hd.c, 378)
floppy_init(); // floppy init(blk_drv/floppy.c, 469)
sti(); // All init has completed, so enable interrupt.
```

Once initialization is complete, execution control is moved to **task 0**. Task 0 is a **special process** that:

- Runs **idle operations** when no other process is scheduled.
- Calls `pause()` to trigger the scheduler.

The macro `move_to_user_mode()` in `include/asm/system.h` moves execution from **kernel mode (privilege level 0)** to **user mode (privilege level 3)**.

```
//// Move to user mode to run.
// This function uses the IRET instruction to move from kernel mode to initial task 0.
//line 01-14
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \ // Save stack pointer ESP to EAX register.
"pushl $0x17\n\t" \ // First push the user stack segment SS,
"pushl %%eax\n\t" \ // then push the stack pointer ESP on to stack,
"pushfl\n\t" \ // and push the EFLAGS register too.
"pushl $0x0f\n\t" \ // Then push the code segment CS of task0,
"pushl $1f\n\t" \ // and push the offset (EIP) at label 1.
"iret\n" \ // Execute the IRET, causes control ret to label 1.
"1:\tmovl $0x17,%%eax\n\t" \ // At this point, kernel begins to execute task 0.
"movw %%ax,%%ds\n\t" \ // Initialize segment register points to the data
"movw %%ax,%%es\n\t" \ // segment of this local descriptor table.
"movw %%ax,%%fs\n\t" \
"movw %%ax,%%gs" \
:::"ax")
```

This **simulates an interrupt return (IRET)**, causing the CPU to switch from privilege level 0 to level 3. After moving to Task 0, the system **creates the first user process (`init`)** using `fork()`.

```
// The task 0 is started by manipulating data in the stack and using interrupt return
// instruction. Then immediately derive a new task 1 (known as init process) in task 0
// and execute init() in the new task. For the child process being created, fork() will
// return 0, and for the original process (parent process) it will return the process
// number pid of the child process.
164 move_to_user_mode(); // see head file include/asm/system.h.
165 if (!fork()) { /* we count on this going ok */
166 init(); // run init() in task 1 (init process).
167 }
```

- Process 1 (`init`)** sets up user space and launches the shell.
- Process 0 (`idle`)** executes `pause()` and allows the scheduler to manage CPU time.

Once Task 0 is in user mode, it serves as the **idle process**. It does this by executing the `pause()` system call, allowing the **scheduler** to run other processes.

```
// The following code runs in task 0.
/*
* NOTE!! For any other task 'pause()' would mean we have to get a
* signal to awaken, but task0 is the sole exception
* as task 0 gets activated at every idle moment (when no other tasks
* can run). For task0 'pause()' just means we go check if some other
* task can run, and if not we return here.
*/
// The pause() syscall converts task 0 into an interruptible wait state before executing
// the scheduler function. However, the scheduler will switch back to task 0 as long as it
// finds that no other tasks in the system can run, and does not depend on the state of
// task 0. See (kernel/sched.c,144).
```

```
for(;;)
__asm__("int $0x80"::"a" (__NR_pause):"ax"); // syscall pause()
}
```

4.4 Process Traces

For a program to be executed, it must first be associated with a process (or task). From the processor's perspective, execution consists of sequentially fetching and executing instructions based on the values stored in the program counter register. Over time, the program counter may point to instructions from different programs, each belonging to separate processes. From the perspective of an individual program, its execution follows a specific sequence of instructions.

The behavior of a process can be described by the sequence of instructions it executes, which is known as the **trace** of the process. Similarly, the overall behavior of the processor can be analyzed by examining how traces from multiple processes are interleaved.

The **dispatcher** plays a crucial role in process execution by managing context switching and scheduling tasks on the CPU:

- `Context switching` is the process of saving the state of the currently executing process or thread and restoring the saved state of the next process or thread. This allows the CPU to switch efficiently between multiple tasks.
- Although the dispatcher does not determine which process runs next—that responsibility belongs to the scheduler—it ensures that the s`elected process is loaded onto the CPU for execution`.

In the figure below, the shaded areas represent code executed by the dispatcher. We assume that the OS only allows a process to continue execution for a maximum of `six` instruction cycles, after which it is interrupted:

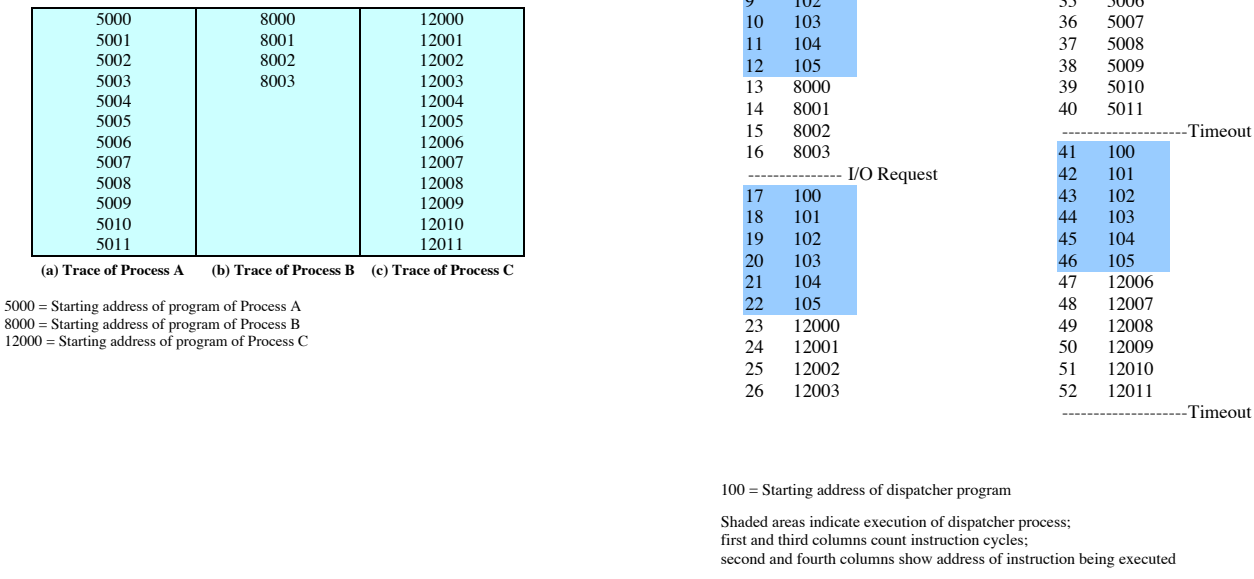


Figure 16: Traces of Processes

4.5 Process States

In this chapter, we consider a computer with a single processor, meaning only one process can be in the running state at any given time. The different process states are defined as follows:

- **New**: A process that has been created but has not yet been loaded into main memory.
- **Ready**: A process that is prepared to execute as soon as the CPU becomes available.

- **Running**: The process that is currently being executed by the processor.
- **Blocked/Waiting**: A process that cannot proceed until a specific event occurs, such as the completion of an I/O operation.
- **Exit**: A process that has completed execution and has been removed from the pool of active processes by the operating system.

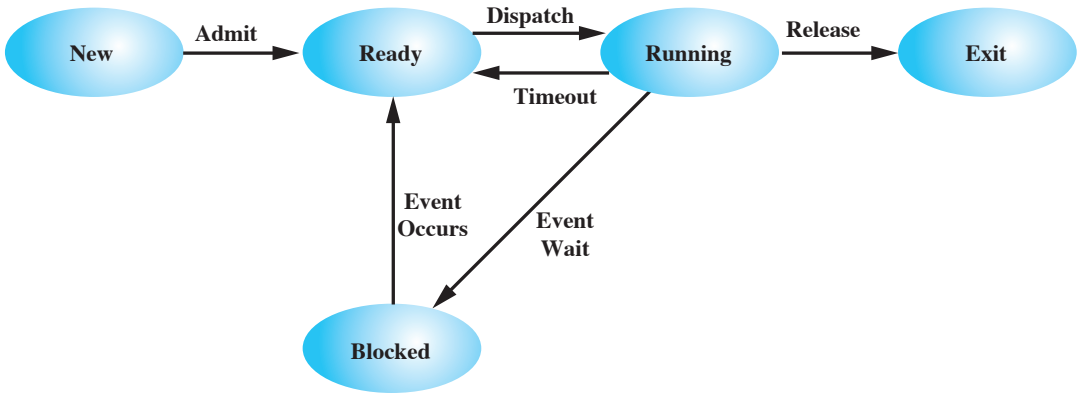


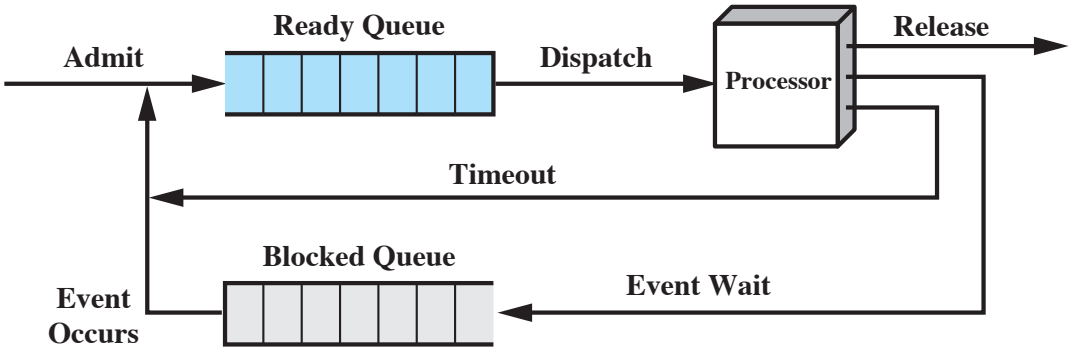
Figure 17: Process States

In an operating system, processes transition between different states based on system conditions and process requirements. The possible state transitions are as follows:

- **Null → New**: A new process is created to execute a program. This event can occur due to various reasons.
- **New → Ready**: The operating system moves a process from the *New* state to the *Ready* state when it is ready to accept additional processes. Most systems enforce a limit on the number of active processes or the amount of virtual memory allocated to ensure optimal performance.
- **Ready → Running**: When selecting a process to execute, the OS chooses one from the *Ready* state. This selection is managed by the scheduler or dispatcher.
- **Running → Exit**: A running process is terminated by the OS if it completes execution or aborts due to an error or external termination request.
- **Running → Ready**: The most common reason for this transition is that the running process has exhausted its allowed execution time. Most multiprogramming operating systems enforce time-based scheduling to ensure fair CPU usage. In some systems, priority levels also influence this transition.
- **Running → Blocked**: A process is moved to the *Blocked* state when it requests an operation that requires waiting. This request typically takes the form of a system service call, such as requesting a resource (e.g., a file or memory segment), initiating an I/O operation, or waiting for interprocess communication data.
- **Blocked → Ready**: A process in the *Blocked* state transitions to *Ready* when the event it was waiting for occurs, such as the completion of an I/O operation or the availability of a requested resource.
- **Ready → Exit**: This transition is not explicitly shown in the state diagram for clarity. However, in some systems, a parent process may terminate a child process at any time. Additionally, if a parent process terminates, all of its child processes may also be terminated.
- **Blocked → Exit**: Similar to the previous case, a blocked process may be terminated if its parent process terminates or if the OS forcibly removes it from execution.

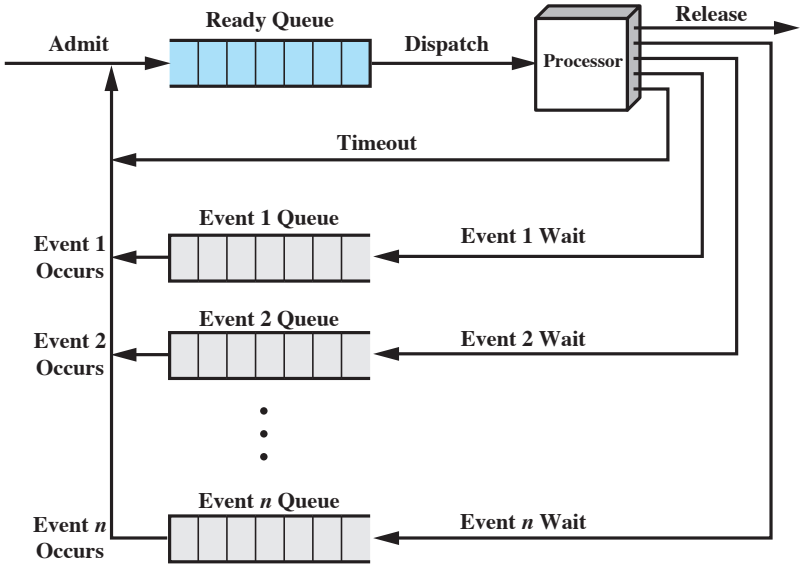
4.6 Using Queues

As each process is admitted to the system, it is placed in the Ready queue. When it is time for the OS to choose another process to run, it selects one from the **Ready queue**. In the absence of any priority scheme, this can be a simple **first-in-first-out queue**. When a running process is removed from execution, it is either terminated or placed in the Ready or Blocked queue, depending on the circumstances. Finally, when an event occurs, any process in the Blocked queue that has been waiting on that event only is moved to the Ready queue.



What if there are many events? How do I know which process needs to be moved to Ready Queue? This latter arrangement means that, when an event occurs, the OS must scan the entire blocked queue, searching for those processes waiting on that event. In a large OS, there could be hundreds or even thousands of processes in that queue.

Therefore, it would be more efficient to have a number of queues, one for each event. Then, when the event occurs, the entire list of processes in the appropriate queue can be moved to the Ready state.

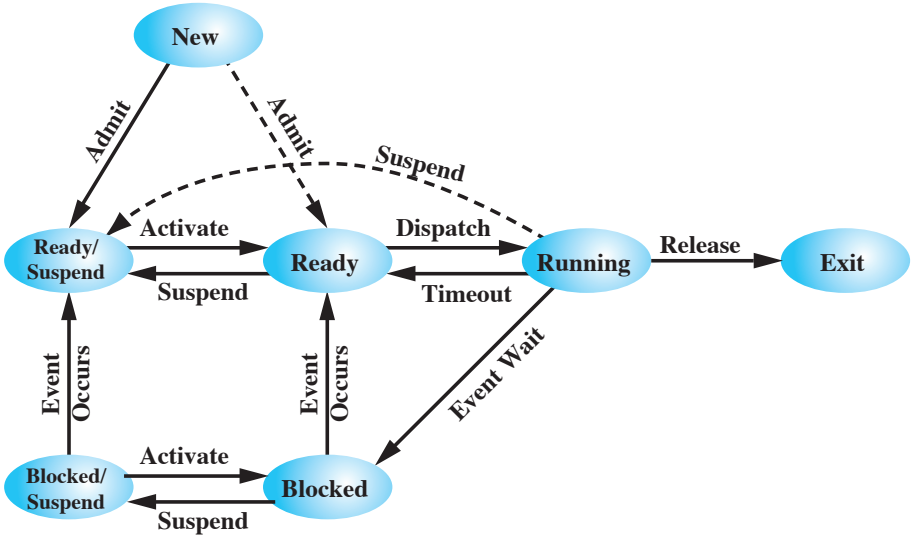


4.7 Suspended State

When none of the processes in main memory is in the Ready state:

- The OS swaps part or all of a blocked process out on to disk into a suspend queue. This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended.
- The OS then brings in another process from the suspend queue, or it honors a new-process request.
- Execution then continues with the newly arrived process

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. `But because disk I/O is generally the fastest I/O on a system (e.g., compared to tape or printer I/O), swapping will usually enhance performance.`



- `Blocked/Suspend`: the Suspend state was still blocked on a particular event.
- `Ready/Suspend`: the event occurs, the process is not blocked and is potentially available for execution

The state transitions are:

- `Blocked -> Blocked/Suspend`: at least one blocked process is swapped out to make room for another process that is not blocked.
- `Blocked/Suspend -> Ready/Suspend`: the event for which it has been waiting occurs.
- `Ready/Suspend -> Ready`: When there are no ready processes in main memory or there are available slots, the OS will need to bring one in to continue execution.
- `Ready -> Ready/Suspend`: if this is the only way to free up a sufficiently large block of main memory.

4.8 Dive into Process States in Old Linux 0.11/0.12

The constant symbol names used for the five states of the Linux process are shown below. They are defined in line 46-50 in the file `include/linux/sched.h`.

```
1 // This defines the state of a process.
2 #define TASK_RUNNING 0 // is running or ready to run
3 #define TASK_INTERRUPTIBLE 1 // is in an interruptible wait state.
4 #define TASK_UNINTERRUPTIBLE 2 // uninterruptible wait state for wait I/O operation.
5 #define TASK_ZOMBIE 3 // is in a zombie state and has been terminated but has not yet been cleaned up by its parent.
6 #define TASK_STOPPED 4 // The process has stopped.
```

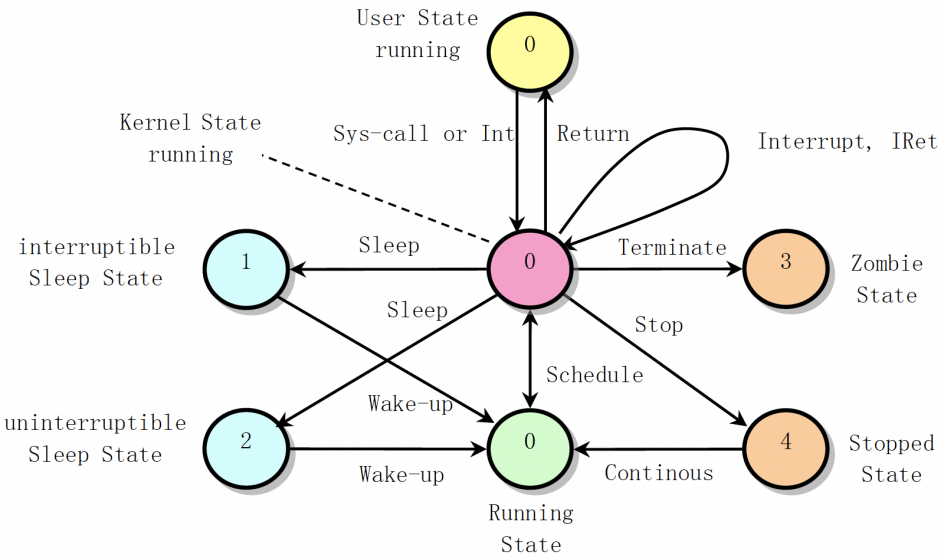


Figure 18: Process status and conversion relationship in Linux 0.11/0.12

- **Running status (0, TASK_RUNNING)**: When a process is being executed by the CPU, or is ready to be executed by the scheduler at any time, the process is said to be running state. In the figure, the middle three circles from top to bottom contains the same value 0, which means that they are all ready or running states. Processes can run in kernel mode or in user mode. When a process is running in kernel code, we call it kernel running state, or simply kernel mode; when a process is executing its own code, we call it user running state (user mode). When a new process has just been created, it is in this state (the bottom 0).
- **Interruptible sleep state (1, TASK_INTERRUPTIBLE)**: When a process is in an interruptible wait (sleep) state, the system does not schedule the process to execute. When the system generates an interrupt or releases the resource that the process is waiting for, or the process receives a signal, it can wake up the process to change to the running state.
- **Uninterruptible sleep state (2, TASK_UNINTERRUPTIBLE)**: This state is similar to the interruptible sleep state except that it is not woken up by the receipt of a signal. However, a process in this state can only be converted to a runnable ready state when it is explicitly awake using the wake_up() function. This state is typically used when a process needs to wait undisturbed or when a waiting event occurs quickly.
- **Zombie state (3, TASK_ZOMBIE)**: When a process has stopped running, but its parent has not called wait() to ask for its status, the process is said to be in a dead state. In order for the parent process to get the information that it stopped running, the task data structure information of the child process needs to be retained. Once the parent process calls wait() to get the information of the child process, the task data structure of the process in that state is released.
- **Stop status (4, TASK_STOPPED)**: When the process receives the signal SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU, it will enter the stop state. Any signal received by the process during debugging will enter this state. Processes in this state will be processed as process.termination.

4.9 Process Control Structure

Consider what the OS must know if it is to manage and control a process. First, it must know `where the process is located`; second, it must know `the attributes of the process` that are necessary for its management (e.g., process ID and process state).

The **process control block** (PCB) plays a crucial role in enabling the operating system (OS) to manage multiple processes and support multiprocessing efficiently. It contains essential information needed to interrupt a running process and later resume its execution seamlessly. When a process is interrupted, the PCB stores key data like the program counter and processor registers, allowing the OS to switch to another process. This enables the OS to handle multitasking effectively. In summary, a process comprises program code, associated data, and a PCB. In a single-processor system, only one process is executing at a time, typically in the running state.

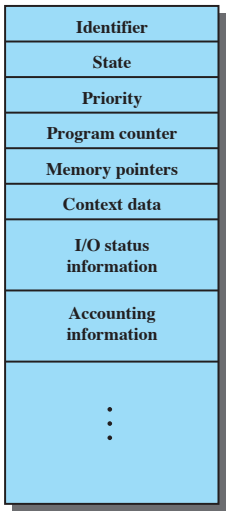
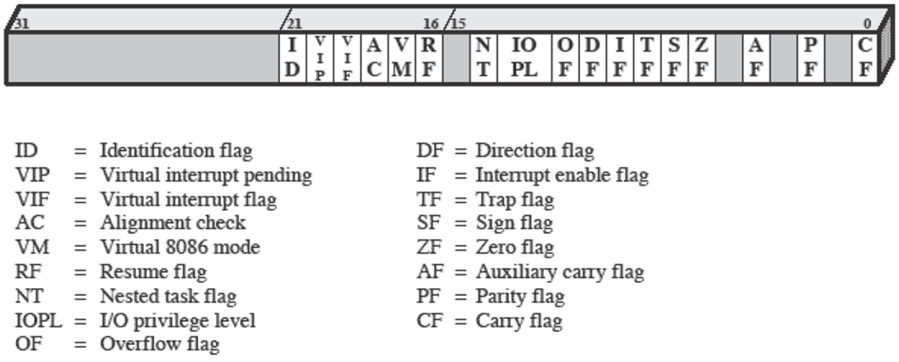


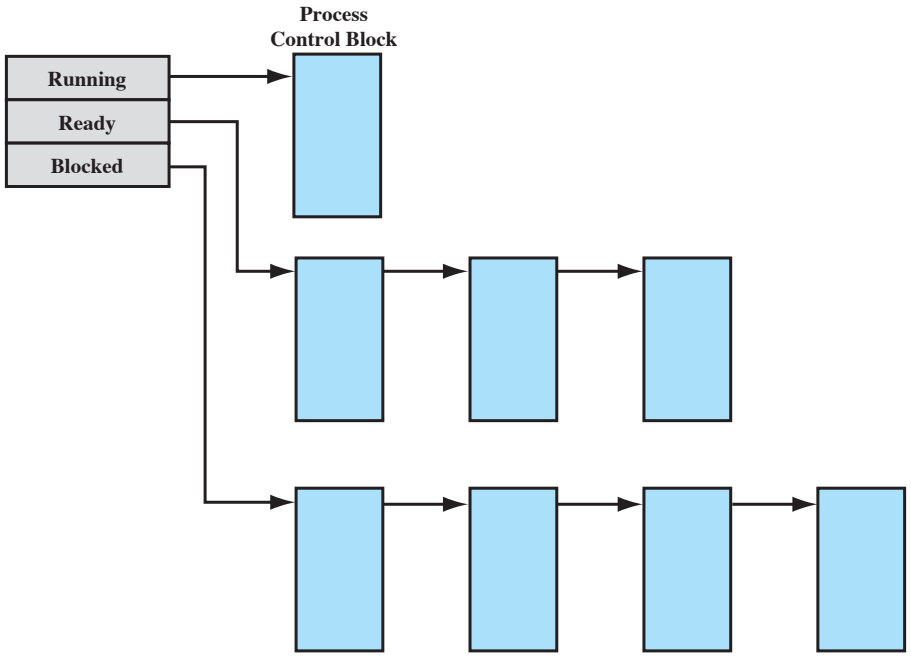
Figure 19: Process control block

Within PCB, we can further divide the infomation into two parts:

- **Process Identification**:In most operating systems, each process is allocated a `unique numeric identifier`, typically serving as an index in the primary process table or through a mapping mechanism for locating relevant tables based on this identifier. This numeric identifier is crucial for various functions within the operating system. It facilitates cross-referencing in tables related to memory, I/O, and files, aiding in tasks such as process allocation and communication destination identification. Moreover, process identifiers play a role in indicating parent-child relationships in process creation. Furthermore, apart from process identifiers, a process may also be assigned a `user identifier`, indicating the user associated with the task.
- **Processor State Information**:Processor registers (user-visible registers; control and status registers; stack pointers) and `Program status word` (PSW) (contains condition codes plus other status information; EFLAGS register is an example of a PSW used by any OS running on an x86 processor)



The process control block may contain structuring information, including pointers that allow the linking of process control blocks. Thus, the queues that were described in the preceding section could be implemented as linked lists of process control blocks.



4.10 Dive into the PCB in the Old Linux 0.11/0.12

In **Linux Kernel 0.12**, the **Process Control Block (PCB)** is represented by the `task_struct` structure, which stores all the necessary information for managing a process. The **Process Control Table** is an array of `task_struct` pointers (`task[]`), which maintains a list of all active processes.

The PCB is defined in `include/linux/sched.h`. It contains various fields for process state, scheduling, memory management, file descriptors, and parent-child relationships.

```
// Below is the task (process) data structure, or process control block, or process descriptor.
// See section 5.7 for detailed descriptions.
struct task_struct {
 long state; // -1 unrunnable, 0 runnable (ready), > 0 stopped.
 long counter; // Task run time tick (decrement), run time slice.
 long priority; // Priority. When task starts running, counter=priority.
 long signal; // Signal bitmap, each bit is a signal( = bit offset + 1).
 struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
 long blocked; // Process signal mask (Bitmap of masked signal).
 int exit_code; // Exit code after task stops, its parent will get it.
 unsigned long start_code; // Code start location in linear address space.
 unsigned long end_code; // Code length or size (bytes).
 unsigned long end_data; // Code size + data siez (bytes).
 unsigned long brk; // Total size (number of bytes).
 unsigned long start_stack; // Stack bottom location.
 long pid; // Process identifier.
 long pgrp; // Process group number.
 long session; // Process session number.
 long leader; // Leader session number.
 int groups[NGROUPS]; // Group numbers. A process can belong to more groups.
 task_struct *p_pptr; // Pointer to parent process.
 task_struct *p_cptr; // Pointer to youngest child process.
 task_struct *p_ysptr; // Pointer to younger sibling process created afterwards.
 task_struct *p_osptr; // Pointer to older sibling process created earlier.
 unsigned short uid; // User id.
 unsigned short euid; // Effective user id.
 unsigned short suid; // Saved user id.
 unsigned short gid; // Group id.
 unsigned short egid; // Effective group id.
 unsigned short sgid; // Saved group id.
 long timeout; // Kernel timing timeout value.
 long alarm; // Alarm timing value (ticks).
 long utime; // User state running time (ticks).
```

```
 long stime; // System state runtime (ticks).
 long cutime; // Child process user state runtime.
 long cstime; // Child process system state runtime.
 long start_time; // Time the process started running.
 struct rlimit rlim[RLIM_NLIMITS]; // Resource usage statistics array.
 unsigned int flags; // per process flags.
 unsigned short used_math; // Flag: Whether a coprocessor is used.
 int tty; // The tty subdevice number used. -1 means no use.
 unsigned short umask; // The mask bit of the file creation attribute.
 struct m_inode * pwd; // Current working directory i node structure pointer.
 struct m_inode * root; // Root i-node structure pointer.
 struct m_inode * executable; // The pointer to i-node structure of the executable file.
 struct m_inode * library; // The loaded library i-node structure pointer.
 unsigned long close_on_exec; // A bitmap flags that close file handles on execution.
 struct file * filp[NR_OPEN]; // File structure pointer table, up to 32 items.
 // The index is the value of file descriptor.
 struct desc_struct ldt[3]; // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
 struct tss_struct tss; // The task status segment structure TSS of the process.
};
```

This structure contains:

- `Process state` (`state`)
- `Scheduling information` (`counter, priority`)
- `Signal handling` (`signal, sigaction[]`)
- `Memory layout` (`start_code, end_code, start_stack`)
- `Process hierarchy` (parent/child/sibling relationships)
- `File system and resource limits`
- `Task state segment (TSS) and Local Descriptor Table (LDT)`

All process control blocks are stored in the **task table** (`task[]`), which is defined in `include/linux/sched.h`.

```
//line 6-32
#define NR_TASKS 64 // The max number of tasks in the system.
#define TASK_SIZE 0x04000000 // The size of each task (64MB).
#define LIBRARY_SIZE 0x00400000 // The size of the loaded library (4MB).

#if (TASK_SIZE & 0x3fffff)
#error "TASK_SIZE must be multiple of 4M"
#endif

#if (LIBRARY_SIZE & 0x3fffff)
#error "LIBRARY_SIZE must be a multiple of 4M"
#endif

#if (LIBRARY_SIZE >= (TASK_SIZE/2))
#error "LIBRARY_SIZE too damn big!"
#endif

#if (((TASK_SIZE>>16)*NR_TASKS) != 0x10000)
#error "TASK_SIZE*NR_TASKS must be 4GB"
#endif

// The location where the library is loaded in the process logical address space (at 60MB).
#define LIBRARY_OFFSET (TASK_SIZE - LIBRARY_SIZE)

// The following macros CT_TO_SECS and CT_TO_USECS are used to convert the current system ticks
// into seconds and microseconds.
#define CT_TO_SECS(x) ((x) / HZ)
#define CT_TO_USECS(x) (((x) % HZ) * 1000000/HZ)

//task is the Process Control Table
```

```
#define FIRST_TASK task[0] // Task 0 is special, so we define a symbol for it.
#define LAST_TASK task[NR_TASKS-1] // The last task in the task array.
```

- Maximum number of tasks: `NR_TASKS = 64`.
- The **task array** (`task[]`) keeps track of all active processes.

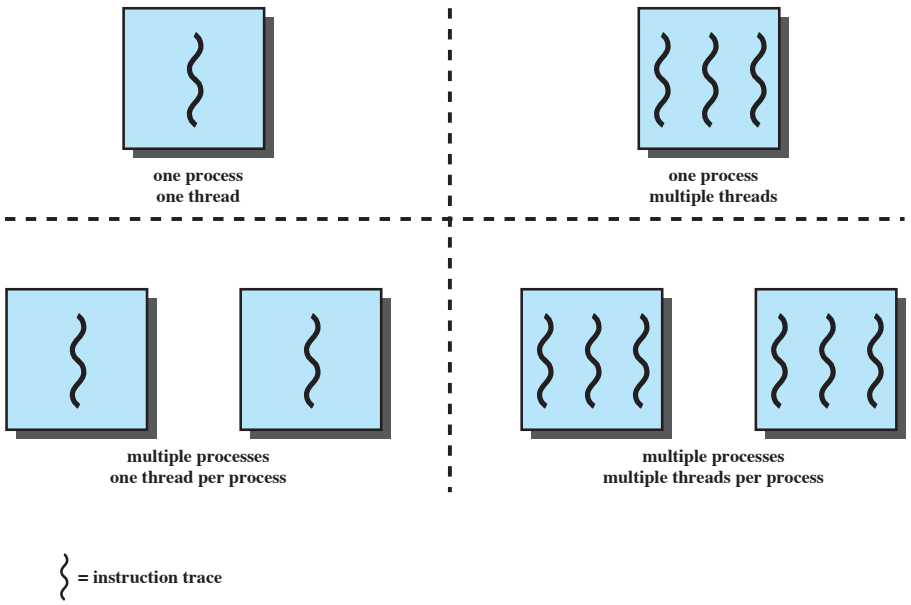
5 Thread

Typically, 'thread' or 'lightweight process' is used to describe the entity responsible for execution, and 'process' or 'task' describes the entity that holds resources. Multithreading enables an operating system to handle several execution paths simultaneously within one process. Reflecting on this will reveal that the OS can manage these two features separately, as they are independent. Modern operating systems, especially newer ones, adopt this approach.

The discussion we did has outlined two fundamental aspects of a process:

`Resource Ownership`: A process contains its program, data, stack, and attributes as defined in the process control block in somewhere located in main memory or disk. Additionally, processes may be granted control or ownership of various resources such as main memory, I/O channels, devices, and files. The operating system plays a crucial role in protecting these resources to prevent unwanted interference between processes.

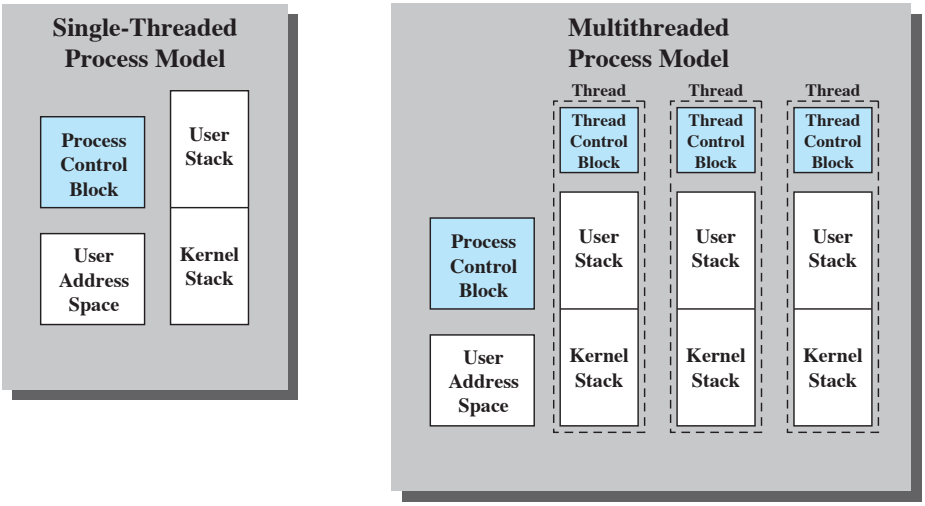
`Scheduling and Execution`: A process follows an execution path, executing one or more programs, which may be interleaved with other processes. Each process has an execution state (e.g., Running, Ready) and a dispatching priority, determining its order of execution by the operating system.



In a multithreaded setting, a process serves as the entity for allocating resources and ensuring security. Processes are linked with several key elements: (1)A virtual memory space containing the process's code and data; (2) Secure access to CPUs, communication with other processes, files, and input/output resources such as devices and channels.

Within a process, there may be one or more threads, each with the following:

1. A thread execution state (Running, Ready, etc.)
2. A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process
3. An execution stack
4. Some per-thread static storage for local variables
5. Access to the memory and resources of its process, shared with all other threads in that process



This figure highlights how threads differ from processes in terms of process control. In the traditional single-threaded process model, where threads are not a separate concept, a process is defined by its process control block, user address space, and both user and kernel stacks, which handle the execution flow. The process also manages the CPU's registers, whose values are stored when the process is not active. In contrast, within a multithreaded context, although the process still maintains a singular process control block and user address space, each thread has its own stacks and a unique thread control block. This thread control block stores specific details such as register values, priority, and other state information pertinent to each thread.

Feature	Process	Thread
Definition	An independent execution unit with its own memory space	A lightweight execution unit within a process
Memory	Separate memory space for each process	Threads share the same memory within a process
Communication	Interprocess communication (IPC) required	Threads share memory, so communication is easier
Overhead	More overhead (context switching, memory allocation)	Less overhead, faster context switching

5.1 Thread vs. Process

Threads within a process share its resources and state, existing in the same memory space and accessing the same data. This means if one thread modifies a piece of data, the change is visible to other threads. Similarly, if a thread opens a file for reading, other threads can also read that file.

The advantages of using threads are mainly performance-related:

- Creating or ending a thread in an existing process is significantly quicker than starting or ending a process.
- Switching between threads in the same process is quicker than switching between different processes.
- Threads enable more efficient communication within the same process. Unlike inter-process communication, which requires kernel intervention, threads can communicate directly thanks to shared memory and files.

Therefore, for tasks that can be divided into related execution units, implementing them as threads within a single process is much more efficient than using multiple separate processes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define NUM 100

void* threadFunc(void* arg) {
    return NULL;
```

```c
}

static void measureThreads() {
    pthread_t threads[NUM];
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    for (int i = 0; i < NUM; i++) {
        pthread_create(&threads[i], NULL, threadFunc, NULL);
    }

    for (int i = 0; i < NUM; i++) {
        pthread_join(threads[i], NULL);
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Time taken to create and join %d threads: %f seconds\n", NUM, cpu_time_used);
}

static void measureProcesses() {
    pid_t pids[NUM];
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    for (int i = 0; i < NUM; i++) {
        if ((pids[i] = fork()) == 0) {
            exit(0);
        }
    }

    for (int i = 0; i < NUM; i++) {
        waitpid(pids[i], NULL, 0);
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Time taken to create and wait for %d processes: %f seconds\n", NUM, cpu_time_used);
}
int main(){
    measureThreads();
    measureProcesses();
    return 0;
}
```

This short program demonstrates that the thread creation and elimination are much faster than the process creation and elimination.

5.2 Thread Use in A Single-User System

Thread can improve the system performance:

- `Foreground and Background Work`: In applications like spreadsheet programs, dividing tasks into threads can enhance user experience. For instance, one thread could manage menus and user input, while another thread executes commands and updates the spreadsheet.
- `Asynchronous Processing`: Asynchronous tasks, vital for functions like data backup in the face of power failures, can be handled through threads. For example, a word processor can employ a dedicated thread to periodically save its RAM buffer to disk without burdening the main program with intricate timing checks or coordination of input and output.
- `Speed of Execution`: Multithreading enables efficient processing by allowing a process to compute one batch of data while simultaneously reading the next batch from a device.

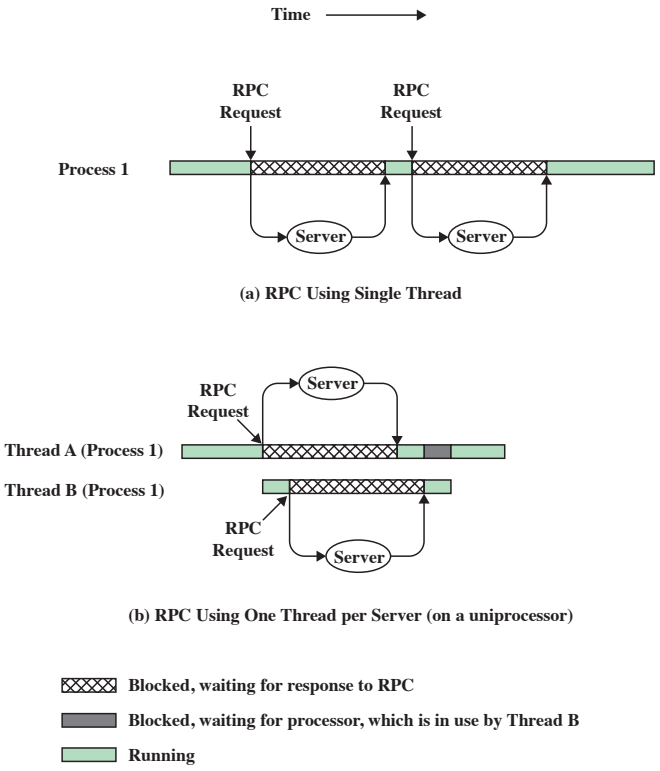
- `Modular Program Structure`: Threads facilitate the design and implementation of programs with diverse activities or input/output sources. This modular approach streamlines development, particularly for applications with varied tasks or data flows.

In operating systems that support threads, scheduling and dispatching are handled at the thread level, leading to most of the execution-related state information being stored in thread-level data structures. However, certain actions affect all threads within a process and require management at the process level by the operating system. For instance, suspension involves swapping out the address space of one process from main memory to accommodate another process, which results in the suspension of all threads within that process simultaneously due to their shared address space. Likewise, terminating a process results in the termination of all threads associated with it.

Do threads improve the performance for a uniprocessor computer system? Let's see a single process with two threads.

For example, consider a program making two remote procedure calls (RPCs) to different hosts to obtain a combined result. In a single-threaded program, the results are obtained sequentially, leading to wait times for each server's response in turn. However, rewriting the program to use a separate thread for each RPC significantly accelerates its execution. Even on a uniprocessor system, where requests and results must still be processed sequentially, concurrent waiting for the replies enhances efficiency.

A Remote Procedure Call (RPC) is a protocol that allows a program on one computer to execute code on a remote system as if it were local. It enables a client program to call functions or procedures on a server program located on another machine over a network.

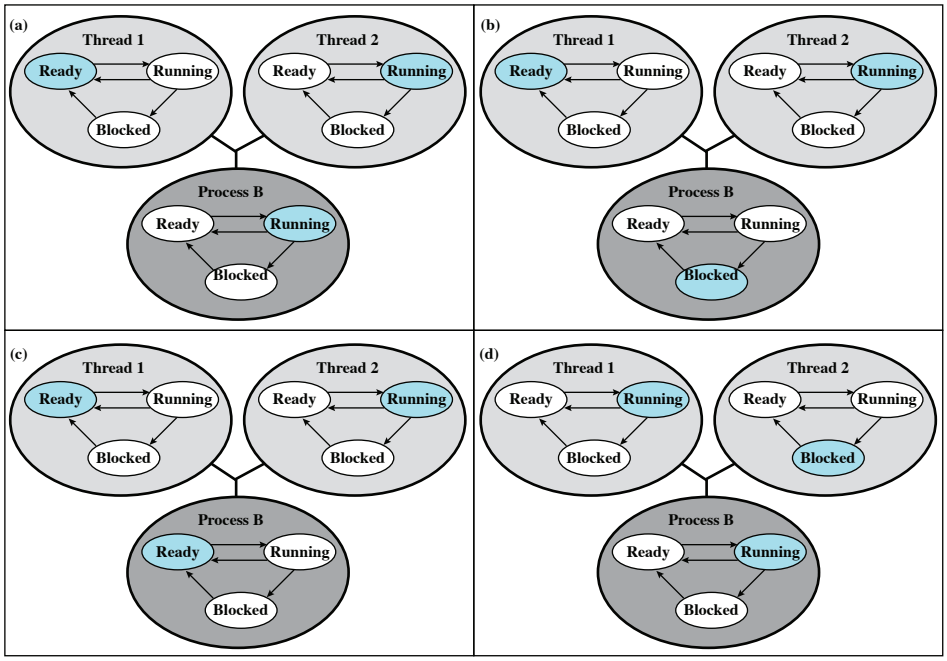


Let's watch a video from ByteByteGo about the differences between a thread and a process [Video].

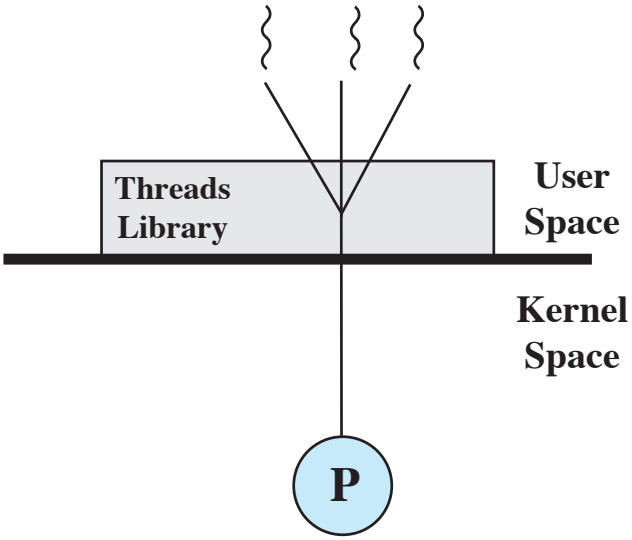
5.3 Types of Threads

There are two broad categories of thread implementation: **user-level threads (ULTs)** and **kernel-level threads (KLTs)**. The latter are also referred to in the literature as kernel-supported threads or lightweight processes.

In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.



All the described activity occurs within user space and within a single process, with the kernel being unaware of these actions. The kernel continues to schedule the process as a whole and assigns a single execution state (such as Ready, Running, or Blocked) to the entire process. The following examples illustrate the relationship between thread and process scheduling:



If a system call made by an application running in a thread causes the process to block, such as an I/O call, the kernel handles the I/O action, puts the process in a Blocked state, and switches to another process. Meanwhile, the thread remains in a perceived Running state within the threads library's data structure, even though it's not actually executing on a processor.

Thread 2 Perceived as Running: Within process B, there's a thread (thread 2) that initiated the I/O operation. From the perspective of the threading library (like Pthreads), this thread may still be considered in the "Running" state. This doesn't mean the thread is actively executing instructions on the CPU. Instead, it indicates that from the threading library's viewpoint, the thread hasn't completed its execution path; it's simply waiting for the I/O operation to finish. **The thread is not physically running on the processor but is logically in a state where it's ready to proceed once the I/O operation is complete.**

5.3.1 ULT's Pros and Cons

Using User-Level Threads (ULTs) offers several **advantages** over Kernel-Level Threads (KLTs), as outlined below:

- Thread switching does not require kernel mode as thread management data structures reside in user space, avoiding mode switch overhead.
- ULTs allow application-specific scheduling, enabling customized algorithms like round-robin or priority-based without affecting the OS scheduler.
- ULTs work with any OS without kernel modifications, as the threads library operates at the application level, ensuring platform independence.

There are two distinct **disadvantages** of ULTs compared to KLTs:

1. Many system calls are blocking in a typical OS. When a ULT executes a system call, the entire process, including all threads, is blocked.
2. A pure ULT strategy cannot fully utilize multiprocessing. Since the kernel is unaware of user-level threads, it schedules the entire process as a single unit on one processor, preventing parallel execution across multiple processors. While ULTs enable efficient thread switching within a process (multiprogramming), they do not leverage true multiprocessing.

To **overcome** ULT's disadvantages,

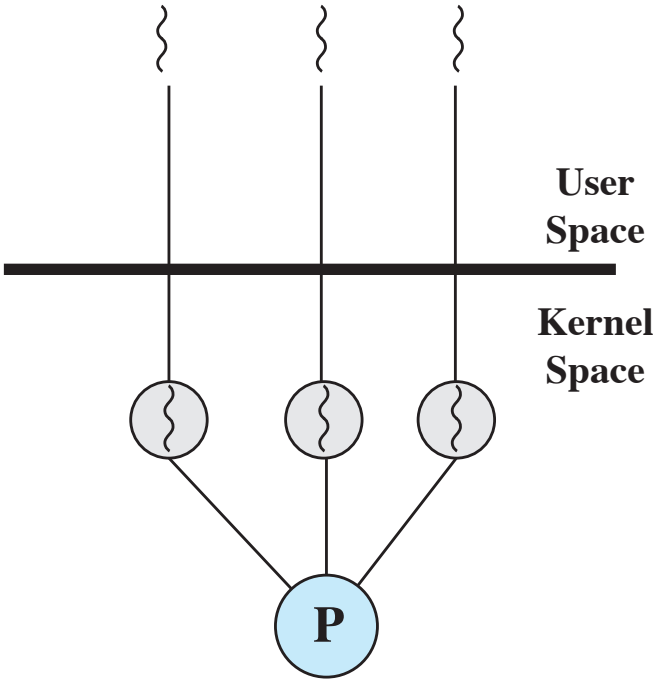
- Using multiple processes instead of threads, avoiding both issues. However, this increases overhead as thread context switches become process switches.
- The blocking issue can be addressed with "**jacketing**," where a blocking system call is converted into a non-blocking one. Instead of calling a system I/O routine directly, a thread invokes an application-level jacket routine. If the operation can proceed without blocking, it executes immediately; otherwise, the thread yields control and retries later.

Jacketing is a technique used to convert blocking system calls into non-blocking ones, preventing user-level threads (ULTs) from blocking the entire process.

- **Why is Jacketing Needed?**
  - Many system calls (e.g., file I/O, network I/O) are blocking.
  - In ULTs, the kernel does not recognize individual threads, so a blocking system call blocks the entire process.
- **How Does Jacketing Work?**
  - Instead of calling a blocking system call, a thread calls a jacket routine (a wrapper function).
  - The jacket routine:
    1. Checks if the operation can proceed without blocking.
    2. If non-blocking, executes the system call immediately.
    3. If blocking, yields control to another thread and retries later.
- **Example:** A thread reading from a file calls `jacket_read(fd, buffer, size)` instead of `read(fd, buffer, size)`.
  - The jacket routine first checks if data is available using a non-blocking method (e.g., `select(fd)`).
  - If data is ready, it performs the read operation.
  - Otherwise, the thread yields and retries later.
- **Benefits:**
  - Prevents full process blocking.
  - Improves thread responsiveness.
  - Enhances efficiency in ULT environments.
- **Limitations:**
  - Not all system calls can be made non-blocking.
  - Adds implementation complexity.

5.3.2 KLT's Pros and Cons

In a pure KLT facility, all of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility. Windows is an example of this approach.



The above figure depicts the pure KLT approach. The kernel maintains context information for the process as a whole and for individual threads within the process.

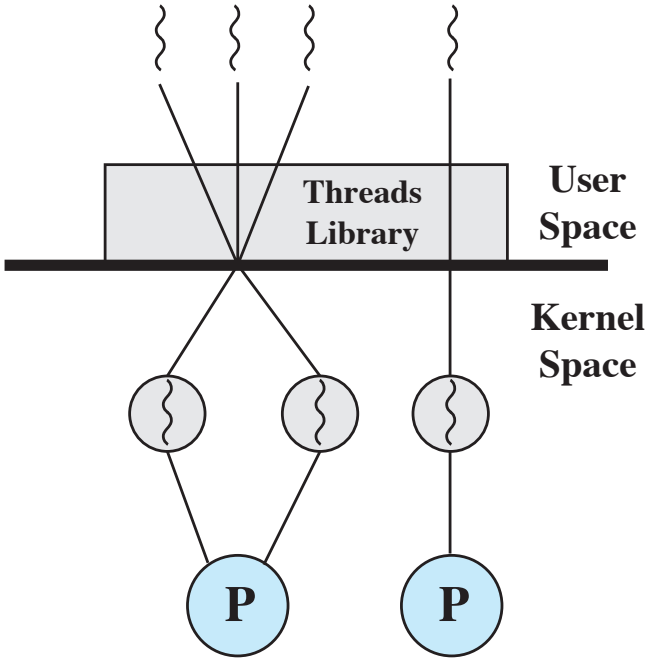
Advantages of KLTs:

- The kernel can efficiently schedule multiple threads from the same process across multiple processors simultaneously.
- When one thread in a process is blocked, the kernel can seamlessly switch to scheduling another thread within the same process.

The principal **disadvantage** of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a `mode switch` to the kernel.

5.3.3 A Combined Approaches

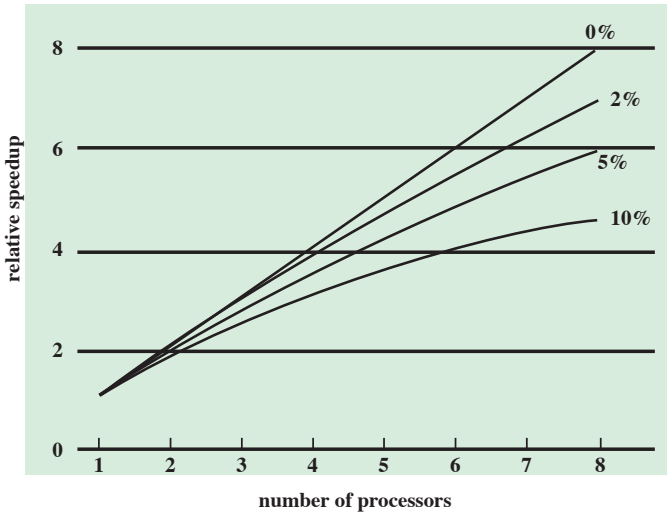
Some operating systems offer a combined User-Level Threads (ULT) and Kernel-Level Threads (KLT) capability. In this hybrid system, thread creation and most thread management tasks occur in user space. Multiple ULTs within an application are mapped to a smaller or equal number of KLTs. Programmers can adjust the number of KLTs to optimize performance for specific applications and processors.



In this combined approach, multiple threads from the same application can run concurrently on multiple processors, and a blocking system call does not necessarily stall the entire process. When well-implemented, this approach aims to harness the benefits of both pure ULT and KLT approaches while mitigating their drawbacks.

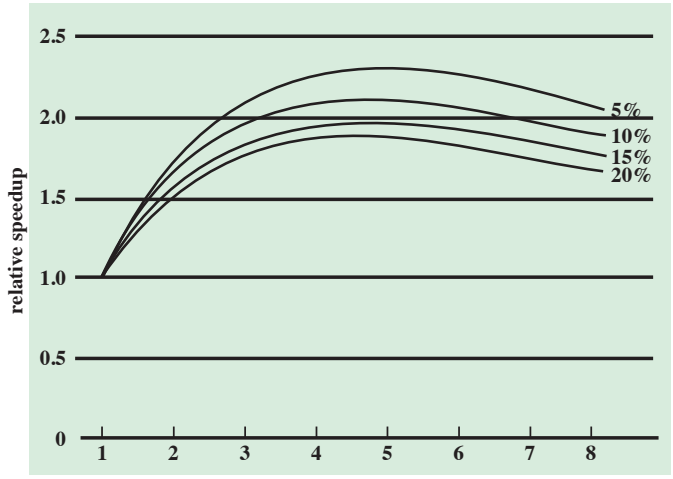
5.4 Performance Discussion

The effectiveness of multicore systems in improving performance hinges on their ability to fully utilize parallel resources. Consider a single application operating on a multicore system: if all program code is parallelizable, running it on eight processors can yield an eightfold performance increase. However, even a small amount of serial code significantly impacts performance. For instance, if only 10% of the code is inherently serial, the performance gain on an eight-processor system reduces to approximately 4.7.



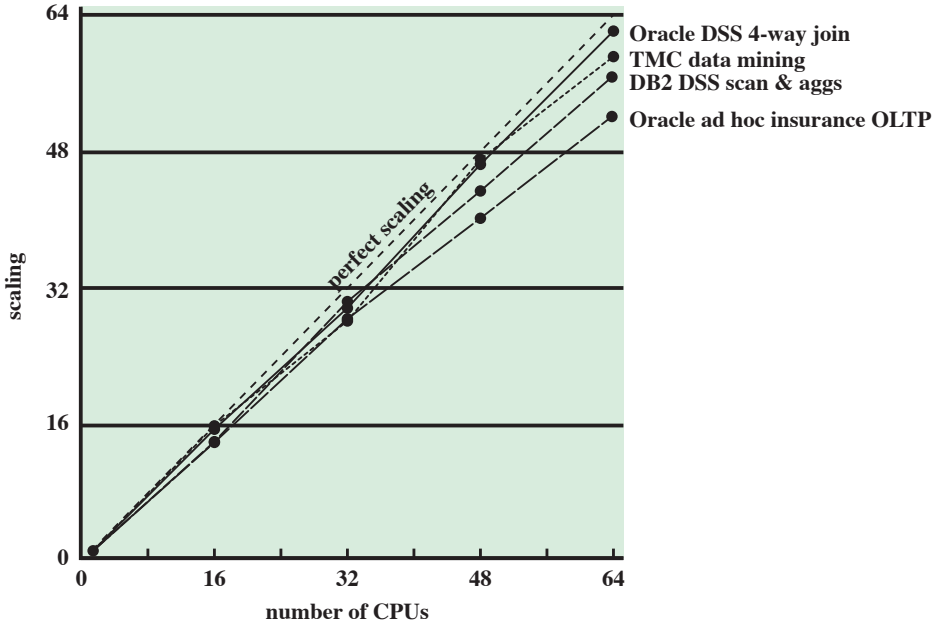
(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

Moreover, programs often incur overhead due to communication, distributing work among multiple processors, and maintaining cache coherence. This creates a performance curve where gains peak and then diminish as the overhead associated with utilizing multiple processors increases.



(b) Speedup with overheads

Software engineers have been actively tackling this challenge, successfully leveraging multicore systems in numerous applications. The most successful one is database application, emphasizing the reduction of serial components across hardware architectures, operating systems, and database software.



5.5 C Examples

5.5.1 The Thread Is Unit of Execution

Let's write a short program to see that thread is the unit of execution. We will do a simple loop task. We will use `sleep()` function to simulate a relatively long task.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

pthread_mutex_t lock;
int counter = 0;

static void* incrementCounter(void* arg) {
    int i;
    for (i = 0; i < 10; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        printf("Thread 1 incremented counter to %d\n", counter);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
    return NULL;
}

static void* decrementCounter(void* arg) {
    int i;
    for (i = 0; i < 10; i++) {
        pthread_mutex_lock(&lock);
        counter--;
        printf("Thread 2 decremented counter to %d\n", counter);
        pthread_mutex_unlock(&lock);
        sleep(3);
    }
    return NULL;
}

int unitOfExecution() {
    pthread_t thread1, thread2;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("Mutex init failed\n");
        return 1;
    }

    if (pthread_create(&thread1, NULL, incrementCounter, NULL)) {
        printf("Error creating Thread 1\n");
        return 1;
    }

    if (pthread_create(&thread2, NULL, decrementCounter, NULL)) {
        printf("Error creating Thread 2\n");
        return 1;
    }

    if (pthread_join(thread1, NULL)) {
        printf("Error joining Thread 1\n");
        return 1;
    }

    if (pthread_join(thread2, NULL)) {
        printf("Error joining Thread 2\n");
```

```
        return 1;
    }

    pthread_mutex_destroy(&lock);

    printf("Final counter value: %d\n", counter);
    printf("Both threads have finished executing.\n");

    return 0;
}

int main(){
    unitOfExecution();
    return 0;
}
```

5.5.2 Multi-Threading

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define NUM_THREADS 4
#define ARRAY_SIZE 10000000


pthread_mutex_t lock;
int array[ARRAY_SIZE];
long long sum = 0;



double current_time_ns() {
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    return (double)(ts.tv_sec * 1.0e9 + ts.tv_nsec) / 1000000000.0;
}

void *calculate_sum(void *threadid) {
    long tid = (long)threadid;
    long long local_sum = 0;
    long long start = (ARRAY_SIZE / NUM_THREADS) * tid;
    long long end = start + (ARRAY_SIZE / NUM_THREADS);

    for (long long i = start; i < end; i++) {
        local_sum += array[i];
    }

    pthread_mutex_lock(&lock);
    sum += local_sum;
    pthread_mutex_unlock(&lock);

    pthread_exit(NULL);
}

void splitAJob () {
    pthread_t threads[NUM_THREADS];
    long t;
    double start, end;
    double time_used;
```

```
    // Initialize array
    for(int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = 1;
    }

    // Sequential sum
    start = current_time_ns();
    long long sequential_sum = 0;
    for(int i = 0; i < ARRAY_SIZE; i++) {
        sequential_sum += array[i];
    }
    end = current_time_ns();
    time_used = (double) (end - start);
    printf("\nSequential read time: %f s\n", time_used);

    // Parallel sum

    start = current_time_ns();
    for(t = 0; t < NUM_THREADS; t++) {
        pthread_create(&threads[t], NULL, calculate_sum, (void *)t);
    }

    for(t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    end = current_time_ns();
    time_used = (double) (end - start);
    printf("\nParallel read time: %f s\n", time_used);
}

int main(){
    splitAJob();
    return 0;
}
```

6 Concurrency: Mutual Exclusion

Concurrency is a fundamental aspect of operating system (OS) design, addressing various challenges such as inter-process communication, resource management (including memory, file access, and I/O operations), process synchronization, and CPU scheduling. These challenges arise not only in multi-processor or distributed systems but also in single-processor environments that manage multiple programs concurrently. Concurrency manifests in several key scenarios:

- **Multiple applications**: The concept of multiprogramming was introduced to enable efficient allocation of processing time among multiple active applications.
- **Structured applications**: By adhering to modular design and structured programming principles, some applications can be effectively developed as a collection of concurrent processes.
- **OS architecture**: The benefits of structured design extend to system programs, with many operating systems being organized as a collection of cooperating processes or threads.

Below are some important concurrency concepts:

Term	Definition
Atomic Operation	A sequence of one or more instructions that executes as an indivisible unit. No other process can observe an intermediate state or interrupt its execution. Either all instructions complete as a group or none at all, ensuring isolation from concurrent processes.
Critical Section	A segment of code within a process that requires access to shared resources and must not be executed concurrently with another process's corresponding section.
Deadlock	A scenario where two or more processes are unable to proceed because each is waiting for the other to release a resource or take action.
Livelock	A condition where two or more processes continuously change states in response to each other without making meaningful progress. Unlike deadlock, processes remain active but ineffective.
Mutual Exclusion	A principle ensuring that when one process accesses a shared resource within its critical section, no other process can access that resource at the same time.
Race Condition	A situation where multiple threads or processes access and modify a shared resource, with the final outcome dependent on the order or timing of their execution.
Starvation	A scenario where a runnable process is repeatedly overlooked by the scheduler, preventing it from ever being executed, despite being ready to proceed.

6.1 Mutual Exclusion: Software Approaches

Software techniques can be employed to manage concurrent processes running on either a single-processor system or a multiprocessor system with shared main memory.

`These methods typically assume basic mutual exclusion at the memory access level.` When multiple processes attempt to read or write to the same memory location simultaneously, a memory arbiter ensures serialization of these accesses, though the order in which access is granted is not predetermined. Apart from this fundamental memory access control, no additional support is assumed from the hardware, operating system, or programming language.

```
/* PROCESS 0 /*
•
•
while (turn != 0)
   /* do nothing */ ;
/* critical section*/;
turn = 1;
•
```

```
/* PROCESS 1 */
•
•
while (turn != 1)
   /* do nothing */;
/* critical section*/;
turn = 0;
•
```

(a) First attempt

```
/* PROCESS 0 */
•
•
while (flag[1])
   /* do nothing */;
flag[0] = true;
/*critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */
•
•
while (flag[0])
   /* do nothing */;
flag[1] = true;
/* critical section*/;
flag[1] = false;
•
```

(b) Second attempt

The first solution ensures the mutual exclusion property but presents two significant drawbacks:

- **Strict Alternation Constraint:** Processes must strictly alternate access to their critical sections, meaning the execution speed is limited by the slower process. For example, if process P_0 enters its critical section only once per hour, while process P_1 intends to enter its critical section 1,000 times per hour, P_1 is forced to operate at the slower pace dictated by P_0.
- **Process Failure Leads to Blocking:** A more severe issue arises if one process fails, causing the other process to be indefinitely blocked. This occurs regardless of whether the failure happens inside or outside the critical section.

The second solution is even less effective than the first, as it fails to guarantee mutual exclusion. Consider the following sequence of execution:

1. P_0 evaluates the `while` condition and observes that `flag[1]` is set to `false`.
2. P_1 evaluates the `while` condition and observes that `flag[0]` is set to `false`.
3. P_0 sets `flag[0]` to `true` and enters its critical section.
4. P_1 sets `flag[1]` to `true` and enters its critical section.

Since both processes have now entered their critical sections simultaneously, the program violates the mutual exclusion property and is thus incorrect.

```
/* PROCESS 0 */
•
•
flag[0] = true;
while (flag[1])
   /* do nothing */;
/* critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */
•
•
flag[1] = true;
while (flag[0])
   /* do nothing */;
/* critical section*/;
flag[1] = false;
•
```

(c) Third attempt

```
/* PROCESS 0 */
•
•
flag[0] = true;
while (flag[1]) {
  flag[0] = false;
  /*delay */;
  flag[0] = true;
}
/*critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */
•
•
flag[1] = true;
while (flag[0]) {
   flag[1] = false;
   /*delay */;
   flag[1] = true;
}
/* critical section*/;
flag[1] = false;
•
```

(d) Fourth attempt

In the third attempt, if a process fails inside its critical section, including during the flag-setting operation that controls access, the other process becomes blocked. Another issue is: If both processes set their flags to `true` before either has evaluated the `while` condition, each will assume that the other has entered its critical section. As a result, both will indefinitely wait, leading to a **deadlock**.

The deadlock in the thrid approach occurs because each process insists on its right to enter its critical section, with no mechanism for backing off.

To resolve this, we introduce a strategy where each process:

- Sets its flag to indicate its **intent** to enter the critical section.
- Is willing to **reset its flag** to yield to the other process if needed.

This approach moves closer to a correct solution, but it still has a flaw. Consider the following sequence of events:

1. P_0 sets `flag[0]` to `true`.
2. P_1 sets `flag[1]` to `true`.
3. P_0 checks `flag[1]`.
4. P_1 checks `flag[0]`.
5. P_0 sets `flag[0]` to `false`.
6. P_1 sets `flag[1]` to `false`.
7. P_0 sets `flag[0]` to `true`.
8. P_1 sets `flag[1]` to `true`.

This sequence can repeat indefinitely, preventing either process from entering its critical section. However, this is not technically deadlock, because any variation in execution speed between the two processes could eventually break the cycle and allow one process to proceed. This condition is known as **livelock**, where processes remain active but continuously defer to each other without making progress.

6.2 Dekker's Algorithm

To resolve the issue of mutual courtesy (livelock), we must impose a structured order on the actions of both processes. The `turn` variable from the first approach can be utilized for this purpose. This variable determines which process has the right to persist in entering its critical section.

When P_0 intends to enter its critical section:

1. It sets its `flag[0]` to `true`.
2. It then checks `flag[1]`.
3. If `flag[1]` is `false`, P_0 may proceed directly into its critical section.
4. Otherwise, P_0 checks the value of `turn`.
5. If `turn` is set to 0, P_0 recognizes that it has the priority to enter and continues to check `flag[1]` until P_1 acknowledges its turn and sets its flag to `false`, allowing P_0 to proceed.

Once P_0 has completed its critical section:

- It sets `flag[0]` back to `false`, releasing access to the critical section.
- It updates `turn` to 1, transferring the priority to P_1.

This approach ensures a structured and enforceable order in which processes take turns entering their critical sections, eliminating the issue of indefinite deferral.

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1) /* do nothing
*/;
                    flag [0] = true;
               }
          }
          /* critical section  */;
          turn = 1;
          flag [0] = false;
          /* remainder   */;
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
          while (flag [0]) {
               if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0) /* do nothing
*/;
                    flag [1] = true;
               }
          }
          /* critical section   */;
          turn = 0;
          flag [1] = false;
          /* remainder   */;
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

// Shared variables
static int want_to_enter[2] = {0, 0}; // Flags for the two threads wanting to enter the critical section
static int turn = 0; // Variable to control whose turn it is

void enter_critical_section(int process) {
    //Dekker's
    printf("Process %d wants to enter the critical section.\n", process);
    want_to_enter[process] = 1; // Indicate that this process wants to enter the critical section
    while (want_to_enter[1 - process]) { // Loop while the other process wants to enter
        if (turn != process) { // If it's not this process's turn,
            want_to_enter[process] = 0; // Relinquish the desire to enter the critical section
            while (turn != process); // Wait until it's this process's turn
            want_to_enter[process] = 1; // Then, indicate again that this process wants to enter
        }
    }
}

void leave_critical_section(int process) {
    printf("Process %d has left the critical section.\n", process);
    turn = 1 - process; // Pass the turn to the other process
    want_to_enter[process] = 0; // Indicate that this process no longer wants to enter the critical section
}

void* process_function(void* arg) {
    int id = *(int*)arg; // Cast void* to int*

```

```
    enter_critical_section(id);

    // Critical section
    printf("Process %d is in the critical section.\n", id);
    sleep(1); // Simulate work in the critical section

    leave_critical_section(id);

    return NULL;
}

int mutualExclusion() {
    pthread_t threads[2];
    int thread_ids[2] = {0, 1};
    int test_round = 2;

    while (test_round--) {
        // Create two threads representing two processes
        for (int i = 0; i < 2; i++) {
            pthread_create(&threads[i], NULL, process_function, &thread_ids[i]);
        }

        // Wait for both threads to complete
        for (int i = 0; i < 2; i++) {
            pthread_join(threads[i], NULL);
        }
    }
    return 0;
}

int main() {
    mutualExclusion();
    return 0;
}
```

6.3 Peterson's Algorithm

Consider process P_0. Once it sets `flag[0]` to `true`, process P_1 is prevented from entering its critical section. If P_1 is already inside its critical section, then `flag[1]` is set to `true`, blocking P_0 from entering its own critical section. However, mutual blocking is avoided.

Suppose P_0 is waiting in its `while` loop. This implies that `flag[1]` is `true` and `turn` is set to 1. P_0 will be able to enter its critical section when either `flag[1]` becomes `false` or `turn` is set to 0.

Exhaustive Cases

Three possible cases arise from this scenario:

1. P_1 has no interest in its critical section. This scenario is impossible because it would imply that `flag[1]` is `false`, contradicting the assumption that P_0 is blocked.
2. P_1 is waiting to enter its critical section. This situation is also not possible, because if `turn` is set to 1, then P_1 would have already entered its critical section.
3. P_1 continuously monopolizes access to its critical section. This cannot occur, as P_1 is required to allow P_0 an opportunity to proceed by setting `turn` to 0 before each new attempt to enter its critical section.

Thus, the mechanism ensures fair access between the two processes while maintaining mutual exclusion.

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1) /* do nothing */;
          /* critical section  */;
          flag [0] = false;
          /* remainder  */;
     }
}
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0) /* do nothing */;
          /* critical section  */;
          flag [1] = false;
          /* remainder  */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

```c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

// Shared variables
static int want_to_enter[2] = {0, 0}; // Flags for the two threads wanting to enter the critical section
static int turn = 0; // Variable to control whose turn it is

void enter_critical_section(int process) {
    //Peterson's
    printf("Process %d wants to enter the critical section.\n", process);
    int other = 1 - process; // Determine the other process
    want_to_enter[process] = 1; // Indicate that this process wants to enter the critical section
    turn = other; // Give the turn to the other process
    while (want_to_enter[other] && turn == other) {
        // Busy wait
    }
}

void leave_critical_section(int process) {
    printf("Process %d has left the critical section.\n", process);
    want_to_enter[process] = 0; // Indicate that this process no longer wants to enter the critical section
}

// FIXED FUNCTION SIGNATURE
void* process_function(void* arg) {
    int id = *(int*)arg; // Cast void* to int*

    enter_critical_section(id);

    // Critical section
    printf("Process %d is in the critical section.\n", id);
    sleep(1); // Simulate work in the critical section

    leave_critical_section(id);

    return NULL;
}

```

```c
int mutualExclusion() {
    pthread_t threads[2];
    int thread_ids[2] = {0, 1};
    int test_round = 2;

    while (test_round--) {
        // Create two threads representing two processes
        for (int i = 0; i < 2; i++) {
            pthread_create(&threads[i], NULL, process_function, &thread_ids[i]);
        }

        // Wait for both threads to complete
        for (int i = 0; i < 2; i++) {
            pthread_join(threads[i], NULL);
        }
    }
    return 0;
}

int main() {
    mutualExclusion();
    return 0;
}
```

6.4 Race Condition

A race condition occurs when multiple processes or threads access and modify shared data in a way that the final outcome depends on the sequence of execution. To illustrate this, consider two simple examples.

In the first example, suppose two processes, P1 and P2, share a global variable, `a`. During execution, P1 assigns `a` the value 1, while P2 assigns `a` the value 2. Since both processes are competing to write to `a`, the final value of `a` depends on which process executes last, making it the "winner" of the race.

For the second example, consider two processes, P3 and P4, which share the global variables b and c, initially set to $b = 1$ and $c = 2$. During execution, P3 performs the assignment:

$$b = b + c$$

while P4 executes:

$$c = b + c.$$

Although they modify different variables, the final values of b and c depend on the order in which these assignments are executed.

- If P3 executes first, the final values will be $b = 3$ and $c = 5$. - If P4 executes first, the final values will be $b = 4$ and $c = 3$.

This demonstrates how the execution order of independent updates can still lead to different outcomes, highlighting the essence of a race condition.

6.5 Producer and Consumer Problem

The **Producer-Consumer Problem** is a classic synchronization problem where multiple producers generate data and store it in a shared buffer, while a single consumer retrieves data from the buffer one item at a time.

- Only **one producer or one consumer** can access the buffer at any given time.
- The **producer must not add data** if the buffer is **full**.
- The **consumer must not remove data** if the buffer is **empty**.

The challenge is to properly **synchronize** the producer and consumer processes to ensure smooth data flow while avoiding issues such as:

- **Buffer overflow** – Occurs when a producer adds data to a full buffer.

- **Buffer underflow** – Occurs when a consumer tries to remove data from an empty buffer.

Proper synchronization mechanisms, such as **mutexes, semaphores, monitor, msgpassing and pipe**, are required to coordinate producer and consumer access to the shared buffer.

6.5.1 Mutex

A **Mutex** (short for *Mutual Exclusion Object*) is a **binary synchronization primitive** designed to control access to a **shared resource**, ensuring that only **one thread** can use it at a time.

- **Purpose**: A mutex is primarily used to **protect critical sections** of code, preventing **race conditions** and ensuring **data consistency**.
- **Lock-Based Mechanism**:
  - A thread must **acquire the lock** before entering the critical section.
  - The thread must **release the lock** once it finishes execution.
  - The same thread must perform both `lock()` and `unlock()` operations.
- **Fairness**: A mutex follows a **"first come, first served"** principle, ensuring that threads gain access in the order they request it.

By enforcing mutual exclusion, mutexes help prevent **concurrent access issues**, making them essential in **multi-threaded applications**. Proper mutex usage ensures efficient resource management and system stability.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0; // Number of items in the buffer
int in = 0; // Next place to write
int out = 0; // Next place to read

pthread_mutex_t mutex;

// Producer function
static void* producer(void* arg) {
    for(int i = 0; i < 20; i++) {
            pthread_mutex_lock(&mutex);
            if(count < BUFFER_SIZE) { // Check if there's space in the buffer
                // Produce an item
                buffer[in] = i;
                in = (in + 1) % BUFFER_SIZE;
                count++;
                printf("Produced: %d, count = %d \n", i, count);
                pthread_mutex_unlock(&mutex);
            }
            else{
                i--;
                pthread_mutex_unlock(&mutex);
            }
        sleep(1);
        }
    return NULL;
}

// Consumer function
static void* consumer(void* arg) {
    for(int i = 0; i < 20; i++) {
            pthread_mutex_lock(&mutex);
            if(count > 0) { // Check if there's an item in the buffer
                // Consume an item
                int item = buffer[out];
                out = (out + 1) % BUFFER_SIZE;
                count--;
```

```
                printf("Consumed: %d\n", item);
                pthread_mutex_unlock(&mutex);
            }
            else {
                i--;
                pthread_mutex_unlock(&mutex);
            }
        sleep(2);
        }
    return NULL;
}

int main() {
    pthread_mutex_init(&mutex, NULL);
    pthread_t prod, cons;

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for the threads to finish
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Cleanup
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

6.5.2 Semaphore

A **semaphore** is an **integer-based synchronization mechanism** used for **signaling between threads**. It allows **multiple threads** to access a **shared resource** concurrently, up to a specified limit. There are two types of semaphore:

- **Counting Semaphore**: Manages access to a **fixed number** of resources by maintaining a count.
- **Binary Semaphore**: Functions similarly to a **mutex** but does not enforce ownership. It is mainly used for simple signaling between threads.

Two popular operations:

- `wait()`: Decreases the semaphore count, blocking the thread if the count reaches zero.
- `signal()`: Increases the semaphore count, allowing waiting threads to proceed.

Semaphores provide **flexibility** in managing **resource allocation** and controlling **concurrent access** in multi-threaded applications.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;

sem_t *empty;   // Number of empty slots; counting sems
sem_t *filled;  // Number of filled slots; counting sems
sem_t *mutex;   // Semaphore for mutual exclusion; binary sems

```

```c
void* producer(void* arg) {
    for (int i = 0; i < 20; i++) {
        sem_wait(empty);  // Wait if buffer is full
        sem_wait(mutex);  // Lock access to shared buffer

        // Critical Section: Producing an item
        buffer[in] = i;
        in = (in + 1) % BUFFER_SIZE;
        printf("Produced %d, count %d\n", i, ++count);

        sem_post(mutex);  // Unlock buffer access
        sem_post(filled); // Signal that an item is available

        sleep(1); // Simulate work
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < 20; i++) {
        sem_wait(filled);  // Wait if buffer is empty
        sem_wait(mutex);   // Lock access to shared buffer

        // Critical Section: Consuming an item
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consumed %d\n", item);
        count--;

        sem_post(mutex);  // Unlock buffer access
        sem_post(empty);  // Signal that a slot is available

        sleep(2); // Simulate work
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    char filled_sem[] = "/filled";
    char empty_sem[] = "/empty";
    char mutex_sem[] = "/mutex";

    sem_unlink(filled_sem);
    sem_unlink(empty_sem);
    sem_unlink(mutex_sem);

    // Initialize named semaphores
    filled = sem_open(filled_sem, O_TRUNC | O_CREAT, 0666, 0);  // Initially, buffer is empty
    empty = sem_open(empty_sem, O_TRUNC | O_CREAT, 0666, BUFFER_SIZE);  // Initially, all slots are empty
    mutex = sem_open(mutex_sem, O_TRUNC | O_CREAT, 0666, 1);  // Initially, mutex is available

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Cleanup
    sem_close(empty);
    sem_close(filled);
    sem_close(mutex);
    sem_unlink(filled_sem);
```

```
    sem_unlink(empty_sem);
    sem_unlink(mutex_sem);

    return 0;
}
```

Feature	Mutex	Semaphore
Ownership	Owned by the thread that locks it	No ownership; any thread can signal it
Count	No count, only locked/unlocked	Maintains a count of available resources
Usage	Ensures exclusive access to a resource	Controls access to a limited number of resources

Table 1: Comparison between Mutex and Semaphore

6.5.3 Monitor

A **monitor** is a programming language construct that provides functionality similar to that of semaphores but offers better control and ease of use. It encapsulates both the **synchronization mechanism** and the **shared resource**, ensuring that access to critical sections is well-structured and less error-prone. A monitor consists of two main components:

- **Mutex (Lock)**: Ensures that only **one thread** can execute any part of the monitor's code at a time.
- **Condition Variables**: Allow threads to **wait inside the monitor** and be **awakened** when a specific condition is met, typically signaled by another thread.

Monitors use **condition variables** for thread synchronization. These variables are **contained within the monitor** and are accessible only from within it. The two fundamental operations on condition variables are:

- **cwait(c)**: Suspends execution of the calling process on condition c.
- **csignal(c)**: Resumes execution of a process that was previously blocked on $cwait(c)$.

The advantages of Monitors are:

- Monitors use **high-level constructs**, making synchronization code more **readable and easier to implement correctly**.
- By encapsulating both the **mutex** and **condition variables**, monitors provide a **modular approach** to synchronization.
- The monitor **controls access** to the critical section, reducing the likelihood of programming errors such as forgetting to release a lock or signal a condition variable.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define BUFFER_SIZE 5

// Monitor structure
typedef struct {
    int buffer[BUFFER_SIZE];
    int in;
    int out;
    int count;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
    pthread_mutex_t mutex;
} Monitor;


// Initialize the monitor
void init_monitor(Monitor *mon) {
    mon->in = 0;
    mon->out = 0;
    mon->count = 0;
    pthread_cond_init(&mon->not_full, NULL);
```

```
    pthread_cond_init(&mon->not_empty, NULL);
    pthread_mutex_init(&mon->mutex, NULL);
}

// Producer function
void *producer(void *arg) {
    Monitor *mon = (Monitor *)arg;
    int item = 0;
    int i = 20;
    while (i--) {
        // Produce item
        item++;

        // Wait if buffer is full
        if (mon->count == BUFFER_SIZE)
        {
            pthread_cond_wait(&mon->not_full, &mon->mutex);
        }


        // Insert item into buffer
        mon->buffer[mon->in] = item;
        mon->in = (mon->in + 1) % BUFFER_SIZE;
        mon->count++;

        // Signal that buffer is not empty
        printf("Produced: %d count %d \n", item, mon->count);
        sleep(1);
        pthread_cond_signal(&mon->not_empty);
    }
    return NULL;
}

// Consumer function
void *consumer(void *arg) {
    Monitor *mon = (Monitor *)arg;
    int item;
    int i = 20;
    while (i--) {
        // Wait if buffer is empty
        if (mon->count == 0)
            pthread_cond_wait(&mon->not_empty, &mon->mutex);


        // Remove item from buffer
        item = mon->buffer[mon->out];
        mon->out = (mon->out + 1) % BUFFER_SIZE;
        mon->count--;

        // Signal that buffer is not full
        printf("Consumed: %d\n", item);
        sleep(2);
        pthread_cond_signal(&mon->not_full);
    }
    return NULL;
}


int main() {
    pthread_t prod, cons;
    Monitor mon;

    // Initialize the monitor
    init_monitor(&mon);

    // Create producer and consumer threads
```

```
    pthread_create(&prod, NULL, producer, (void *)&mon);
    pthread_create(&cons, NULL, consumer, (void *)&mon);


    // Wait for threads to finish (which will never happen due to infinite loop)
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    return 0;
}
```

6.5.4 Message Passing

All the previous methods are working on some shared data, it will be a different story when we talk processes, unlike threads, processes do not have shared memory space.

When processes interact with each other, two fundamental requirements must be met: **synchronization** and **communication**. Synchronization is essential to enforce mutual exclusion, while cooperating processes may require a means to exchange information. One effective approach to achieving both of these functions is **message passing**.

The core functionality of message passing is typically provided through a pair of fundamental primitives:

```
send(destination, message)
receive(source, message)
```

These two operations represent the minimum set required for processes to engage in message passing. A process transmits information by executing the `send` primitive, specifying the *destination* and the *message*. Conversely, a process receives information by executing the `receive` primitive, indicating the *source* and the *message*.

The message box is a queue; during the communication, we need to ensure that the queue will not overdraw or overflow!

Let's learn a few functions before we dive into coding example:

- `int msgctl(int msqid, int cmd, struct msqid_ds *buf);`

  `msqid` - The identifier of the message queue, obtained from msgget().

  `cmd` - The control operation to be performed. Some common values:

  1. IPC_STAT – Retrieves the current attributes of the message queue and stores them in buf.
  2. IPC_SET – Sets the attributes of the message queue using the values in buf (requires appropriate permissions).
  3. IPC_RMID – Removes the message queue and all associated messages.

  `buf` – A pointer to a struct msqid_ds structure, which holds information about the message queue.

- `int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);`

  `msqid` - The message queue identifier.

  `msgp` - Pointer to a buffer to store the received message.

  `msgsz` - Size of the message body.

  `msgtyp` - Specifies which message to receive.

  `msgflg` - Flags controlling message retrieval behavior.

- `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);`

  `msqid` - The message queue identifier.

  `msgp` - Pointer to the message structure.

  `msgsz` - Size of the message body.

  `msgflg` - Flags controlling message sending behavior.

- `int msgget(key_t key, int msgflg);`

  `key` - A unique identifier for the message queue.

  `msgflg` - Flags specifying permissions and creation options.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/wait.h>
#include <time.h>

#define QUEUE_PERMS 0666
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 3

struct message {
    char msg_text[100];
};

static void producer(int msgid, int producer_id) {
    struct message msg;
    struct msqid_ds buf;

    for (int i = 0; i < 3; ++i) {
        int num = producer_id * 10 + i; // Generate a message
        snprintf(msg.msg_text, sizeof(msg.msg_text), "Producer %d: %d", producer_id, num);

        // Check queue size before sending
        while (1) {
            if (msgctl(msgid, IPC_STAT, &buf) == -1) {
                perror("msgctl failed");
                exit(EXIT_FAILURE);
            }

            if (buf.msg_cbytes < buf.msg_qbytes) {
                break; // Queue has space, exit loop and send message
            }

            printf("Producer %d waiting, queue full...\n", producer_id);
            sleep(1); // Wait before retrying
        }

        // Send the message
        if (msgsnd(msgid, &msg, sizeof(msg.msg_text), 0) == -1) {
            perror("Producer: msgsnd failed");
            exit(EXIT_FAILURE);
        }

        printf("Producer %d sent: %s\n", producer_id, msg.msg_text);
        sleep(1); // Slow down production rate
    }
}


static void consumer(int msgid, int consumer_id) {
    struct message msg;

    for (int i = 0; i < 2; ++i) { // Each consumer reads 2 messages
        while (1) {
            if (msgrcv(msgid, &msg, sizeof(msg.msg_text), 0, IPC_NOWAIT) != -1) {
                break; // Successfully received a message
            }

            printf("Consumer %d waiting, queue empty...\n", consumer_id);
```

```
            sleep(1); // Wait before retrying
        }

        printf("Consumer %d received: %s\n", consumer_id, msg.msg_text);
        sleep(5); // Slow down consumption rate
    }
}


int main() {

    int msgid;
    key_t key = ftok("producer_consumer_ipc", 65); // Generate unique key
    //The msgget() function is used in System V IPC (Inter-Process Communication)
    //to create or access a message queue. Message queues allow processes to exchange data asynchronously.
    msgid = msgget(key, QUEUE_PERMS | IPC_CREAT); //  Create with read/write permissions
    if (msgid == -1) {
        perror("msgget failed");
        exit(EXIT_FAILURE);
    }
    struct msqid_ds buf;
    msgctl(msgid, IPC_STAT, &buf);
    buf.msg_qbytes = 65536;  // Set queue size to 64KB
    msgctl(msgid, IPC_SET, &buf);


    for (int i = 0; i < NUM_PRODUCERS + NUM_CONSUMERS; ++i) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork failed");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) { // Child process
            if (i < NUM_PRODUCERS) {
                producer(msgid, i);
            } else {
                consumer(msgid, i-2);
            }
            exit(0); // Child exits after its task
        }
    }

    // Parent waits for all child processes
    for (int i = 0; i < NUM_PRODUCERS + NUM_CONSUMERS; ++i) {
        wait(NULL);
    }

    // Cleanup message queue in parent
    if (msgctl(msgid, IPC_RMID, NULL) == -1) {
        perror("msgctl failed");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

6.5.5 Pipe

A pipe in an operating system is a communication mechanism that allows one process to send data to another process in a unidirectional flow. Pipes are commonly used in Inter-Process Communication (IPC), enabling processes to exchange information efficiently. Data is stored in a kernel buffer before being read. Here we will only discuss simplest pipe application, which is a blocking pipe. The process is:

1. Producer writes a message to the pipe.

2. Producer waits for acknowledgment.
3. Consumer reads the message.
4. Consumer sends an acknowledgment.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

// Global variables for pipes
int mayproduce[2];
int mayconsume[2];

static void producer() {
    char msg[100];
    char ack;

    for (int i = 0; i < 5; i++) {
        // Prepare a message
        snprintf(msg, sizeof(msg), "Message %d", i + 1);

        // Write the message to the consumer
        write(mayconsume[1], msg, strlen(msg) + 1);
        printf("Produced: %s\n", msg);

        // Wait for acknowledgment from the consumer
        read(mayproduce[0], &ack, 1);
    }
}

static void consumer() {
    char msg[100];

    while (read(mayconsume[0], msg, sizeof(msg)) > 0) {//pipes in blocking mode (the default behavior) will cause the reading proce
        printf("Consumed: %s\n", msg);

        // Send acknowledgment to the producer
        write(mayproduce[1], "1", 1);
    }
}


int main() {
    // Create pipes
    if (pipe(mayproduce) == -1 || pipe(mayconsume) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork a new process
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process: Consumer
        consumer();
    } else {
        // Parent process: Producer
        producer();
    }

    // Close pipes in the main process
    close(mayproduce[0]);
```

```
    close(mayproduce[1]);
    close(mayconsume[0]);
    close(mayconsume[1]);

    return 0;
}
```

6.6 Reader & Writer Problem

A shared data area exists, which multiple threads can access. This shared area could be a file, a block of memory, or a set of processor registers. Among the threads, some are readers (which only read the data) and others are writers (which only modify the data). The problem must be managed under the following constraints:

Multiple readers can access the shared data simultaneously. Only one writer can modify the shared data at a time. If a writer is actively modifying the shared data, no reader is allowed to access it. In summary, readers do not need to exclude each other, meaning multiple readers can operate concurrently. However, writers must have exclusive access, preventing both other writers and readers from accessing the data simultaneously.

6.6.1 Semophore and Mutex Solution

This is not a very rigorous implementation. We assume there is no shared data in the reader, even though we let readers update a shared counter. This counter should work in a more independent way or hardware level. We assume there is no syncronization issue for the reader_count shared variable in the code below.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_READER 3
#define NUM_WRITER 2

pthread_mutex_t mutex;
sem_t *write_sem;
sem_t *read_sem;

int reader_count = 0;
int share_resource = 0;

void *reader(void *arg){
    int id = *(int *)arg;
    int iteration = 0;

    while(iteration < 5){
        sem_post(read_sem);
        reader_count++;

        if(reader_count == 1){
            sem_wait(write_sem);
        }

        sem_wait(read_sem);

        printf("Reader %d reads: %d \n", id, share_resource);
        sleep(1);

        reader_count--;

        if(reader_count == 0){
            sem_post(write_sem);
        }

        sleep(rand() % 3);
```

```
        iteration++;
    }
    printf("reader %d finished \n", id);
    return NULL;
}

void *writer(void *args){
    int id = *(int *)args;
    int iteration = 0;

    while(iteration < 5){
        sem_wait(write_sem);
        pthread_mutex_lock(&mutex);
        share_resource++;
        printf("Writer %d writes: %d \n", id, share_resource);
        sleep(2);
        pthread_mutex_unlock(&mutex);

        sem_post(write_sem);
        sleep(rand() % 3);
        iteration++;
    }
    printf("writer %d finished! \n", id);
    return NULL;
}

int main(){
    pthread_t readers[NUM_READER], writers[NUM_WRITER];
    int i;

    srand(time(NULL));

    sem_unlink("write_sem");
    sem_unlink("read_sem");

    pthread_mutex_init(&mutex, NULL);

    write_sem = sem_open("/write_sem", O_CREAT, 0666, 1);
    read_sem = sem_open("/read_sem", O_CREAT, 0666, 0);

    for(i = 0; i < NUM_READER; ++i){
        int id = i;
        pthread_create(&readers[i], NULL, reader, (void *)&id);
    }

    for(i = 0; i < NUM_WRITER; ++i){
        int id = i;
        pthread_create(&readers[i], NULL, writer, (void *)&id);
    }

    for(i = 0; i < NUM_READER; ++i){
        pthread_join(readers[i], NULL);
    }

    for(i = 0; i < NUM_WRITER; ++i){
        pthread_join(writers[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_close(write_sem);
    sem_close(read_sem);
    sem_unlink("write_sem");
    sem_unlink("read_sem");

    return 0;

```

```
}
```

6.6.2 Monitor Solution

This is not a very rigorous implementation. We assume there is no shared data in the reader, even though we let readers update a shared counter. This counter should work in a more independent way or at a hardware level. We assume there is no syncronization issue for the active_readers shared variable in the code below.

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>

#define BUFFER_SIZE 5  // Size of the buffer

// Monitor structure
typedef struct {
    pthread_mutex_t mutex;          // Mutex for critical section
    pthread_cond_t readers_cond;    // Condition variable for readers
    pthread_cond_t writers_cond;    // Condition variable for writers
    int buffer;                     // Buffer for storing data
//    int active_readers;             // Number of readers currently reading
    int active_writers;             // Flag to indicate if a writer is active
} RW_Monitor;
int active_readers = 0;
RW_Monitor monitor = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, PTHREAD_COND_INITIALIZER, 0, 0};

void* reader(void* args) {
    int id = (int)(intptr_t)args;
    while (1) {
        // Wait until there are no active writers
        while (monitor.active_writers > 0) {
            pthread_cond_wait(&monitor.readers_cond, &monitor.mutex);
        }

        // Increase the number of active readers
        active_readers++;

        // Reading section (Simulated delay)
        printf("Reader %d is reading: %d\n", id, monitor.buffer);
        sleep(rand() % 10);

        active_readers--;

        // If last reader, signal waiting writers
        if (active_readers == 0) {
            pthread_cond_signal(&monitor.writers_cond);
        }

        sleep(1);  // Allow other readers/writers to execute
    }
    return NULL;
}

void* writer(void* args) {
    int data = (int)(intptr_t)args;
    while (1) {
        pthread_mutex_lock(&monitor.mutex);

```

```
        // Wait until no active readers or writers
        while (active_readers > 0 || monitor.active_writers > 0) {
            pthread_cond_wait(&monitor.writers_cond, &monitor.mutex);
        }

        // Mark writer as active
        monitor.active_writers = 1;

        // Writing section
        monitor.buffer = data;
        printf("Writer %d wrote: %d\n", data, data);
//        sleep(rand() % 3);  // Simulated writing time

        // Mark writing as complete
        monitor.active_writers = 0;

        // Signal waiting readers and writers
        pthread_cond_broadcast(&monitor.readers_cond);
        pthread_cond_signal(&monitor.writers_cond);

        pthread_mutex_unlock(&monitor.mutex);

        sleep(1);  // Allow other readers/writers to execute
    }
    return NULL;
}

int main() {
    pthread_t readers[2], writers[3];

    // Create reader threads
    for (int i = 0; i < 2; ++i) {
        pthread_create(&readers[i], NULL, reader, (void*)(intptr_t)(i + 1));
    }

    // Create writer threads
    for (int i = 0; i < 3; ++i) {
        pthread_create(&writers[i], NULL, writer, (void*)(intptr_t)(i + 1));
    }

    // Wait for all threads to finish (they run indefinitely)
    for (int i = 0; i < 2; ++i) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < 3; ++i) {
        pthread_join(writers[i], NULL);
    }

    return 0;
}
```

6.7 Mutual Exclusion in Linux 0.11

In Linux 0.11, mutual exclusion is handled using cli/sti and semaphores (implemented via sleep/wakeup). Since Linux 0.11 is a very early version of Linux, it relies on these primitive mechanisms rather than more modern techniques like mutexes or monitors.

6.7.1 Disabling Interrupts (cli/sti)

The simplest method for mutual exclusion in the kernel is disabling interrupts using cli (clear interrupt flag) and sti (set interrupt flag). This ensures that critical sections are not interrupted by other processes or interrupts.

```
cli()         ; Disable interrupts (critical section starts)
; Critical Section: modify shared data
sti()         ; Enable interrupts (critical section ends)
```

This is mostly used in short, non-preemptive critical sections to avoid concurrency issues.

An example is protecting the time list (`kernel/sched.c`). Linux 0.11 uses cli before modifying `time list` to prevent scheduling issues. This function is used for scheduling delayed tasks and implementing timeouts.

The `add_timer()` function is part of the Linux kernel scheduler, used to schedule delayed function execution in the kernel. It inserts a function (fn) into a timer queue, ensuring that it will be executed after a specified delay (jiffies). What is jiffies? jiffies is a system tick counter in the Linux kernel. It increments at a fixed rate (e.g., every 10 ms in a 100 Hz system). The function schedules execution jiffies ticks into the future.

```c
//kernel/sched.c line 291~322
void add_timer(long jiffies, void (*fn)(void)) {
    struct timer_list *p;

    // Exit if the function pointer is NULL.
    if (!fn)
        return;

    cli();  // Disable interrupts

    // If the timer delay is non-positive, execute the function immediately.
    if (jiffies <= 0) {
        fn();
    } else {
        // Find a free slot in the timer list.
        for (p = timer_list; p < timer_list + TIME_REQUESTS; p++) {
            if (!p->fn)
                break;
        }

        // If no free slot is found, halt the system.
        if (p >= timer_list + TIME_REQUESTS)
            panic("No more time requests free");

        // Initialize the timer entry.
        p->fn = fn;
        p->jiffies = jiffies;
        p->next = next_timer;
        next_timer = p;

        // Sort the list based on expiration time.
        // Ensures that the earliest timeout remains at the head of the list.
        while (p->next && p->next->jiffies < p->jiffies) {
            p->jiffies -= p->next->jiffies;

            // Swap function pointers
            fn = p->fn;
            p->fn = p->next->fn;
            p->next->fn = fn;

            // Swap jiffies values
            jiffies = p->jiffies;
            p->jiffies = p->next->jiffies;
            p->next->jiffies = jiffies;

            // Move to the next element in the list
            p = p->next;
        }
    }

    sti();  // Enable interrupts
```

```
}

//--------------Example----------------------
void my_callback_function(void) {
    printk("Timer expired! Function executed.\n");
}

int main() {
    add_timer(50, my_callback_function);  // Schedule execution in 50 jiffies
}
```

We can find the definition of `cli()` and `sti()` in Linux 0.11, the `cli()` and `sti()` macros are defined in `include/asm/system.h`:

```
#define sti() __asm__ ("sti"::) // enable interrupt.
#define cli() __asm__ ("cli"::) // disable interrupt.
```

6.7.2 Semaphores

In Linux 0.12, semaphores are implemented using `sleep_on()` and `wake_up()` functions. These functions allow processes to wait for resources without busy-waiting, preventing CPU waste.

```
static inline void __sleep_on(struct task_struct **p, int state) {
    struct task_struct *prev_task;

    // If the provided pointer is invalid, exit immediately.
    // If the current task is the initial task (task 0), trigger a kernel panic.
    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");

    // Store the current task at the front of the wait queue and
    // keep a reference to the previously waiting task (if any).
    prev_task = *p;
    *p = current;
    current->state = state;

    // Repeatedly call schedule() until properly woken up.
    while (1) {
        schedule();

        // If another task has entered the queue after the current task,
        // wake it up and put the current task back into an uninterruptible wait state.
        if (*p && *p != current) {
            (**p).state = 0; // Wake up the next task
            current->state = TASK_UNINTERRUPTIBLE; // Put current task back to sleep
        } else {
            break; // Exit loop when properly woken up
        }
    }

    // If the queue head is NULL, warn about potential scheduling issues.
    if (!*p)
        printk("Warning: *P = NULL\n\r");

    // Restore the previous waiting task in the queue and wake it up if it exists.
    if ((*p = prev_task))
        prev_task->state = 0; // Wake up previous task
}
```

Example walkthrough:

Assume three processes P_1, P_2, P_3 call `__sleep_on()` in sequence.

1. P_1 Goes to Sleep

$$*p = P_1$$
$$\text{prev\_task} = \text{NULL}$$
P1 calls `schedule()` and stops running.

2. P_2 Enters the Queue

$$\text{prev\_task} = P_1$$
$$*p = P_2$$
P2 calls `schedule()` and stops running.

3. P_3 Enters the Queue

$$\text{prev\_task} = P_2$$
$$*p = P_3$$
P3 calls `schedule()` and stops running.

Wake-Up sequence:

1. An external event (e.g., disk I/O completion) calls `wake_up(&p)`.
2. P_3 wakes up first because it is at the front of the queue.
3. P_3 sees that P_2 is in the queue, so it wakes up P_2 and goes back to sleep (`goto repeat`).
4. P_2 wakes up, sees that P_1 is still in the queue, wakes up P_1, and goes back to sleep.
5. P_1 finally wakes up and runs.

This ensures that tasks are woken up in the same order they went to sleep.

Another function is `wake_up`:

```c
void wake_up(struct task_struct **p) {
    if (p && *p) {
        // Print a warning if the task is stopped or a zombie
        if ((**p).state == TASK_STOPPED)
            printk("wake_up: TASK_STOPPED");
        if ((**p).state == TASK_ZOMBIE)
            printk("wake_up: TASK_ZOMBIE");

        // Set the task state to TASK_RUNNING (ready to execute)
        // Allowing the scheduler to execute it.
        (**p).state = 0;
    }
}
```

Example: Buffer Locking Mechanism (fs/buffer.c)

This part explains how the Linux 0.12 kernel manages buffer synchronization using `sleep_on()` and `wake_up()`. It ensures that only one process can modify a buffer at a time, preventing race conditions.

If a process tries to access a locked buffer, it must wait until it is free.

```c
// Function to put the process to sleep if the buffer is locked
void wait_on_buffer(struct buffer_head *bh) {
    cli();  // Disable interrupts to prevent race conditions
    while (bh->b_lock)   // If buffer is locked, put process to sleep
        sleep_on(&bh->b_wait);
    sti();  // Enable interrupts
}
```

When a process finishes using the buffer, it unlocks it and wakes up any waiting process.

```
// Function to unlock the buffer and wake up waiting processes
void unlock_buffer(struct buffer_head *bh) {
    cli();
    bh->b_lock = 0;  // Unlock the buffer
    wake_up(&bh->b_wait);  // Wake up any waiting process
    sti();
}
```

Assume two processes, P_1 and P_2, try to access the same buffer.

1. P_1 locks the buffer and begins modification.
2. P_2 tries to access the buffer but finds it locked.
3. P_2 goes to sleep using `sleep_on()`.
4. P_1 finishes its work and calls `unlock_buffer()`.
5. `wake_up()` is called, waking P_2.
6. P_2 resumes execution and can now access the buffer.

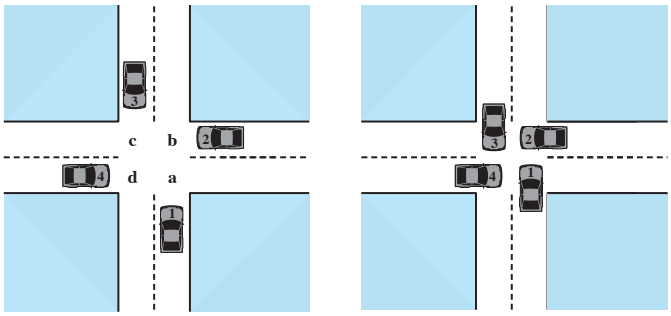
7 Deadlock

`Definition`: A state in which a group of processes becomes permanently stuck, unable to proceed because they either compete for system resources or rely on communication with each other.

`Description`: Deadlock occurs when every process in a given set is waiting for an event—typically the release of a required resource—that can only be triggered by another process in the same set, which is also blocked. Since none of these events occur, the deadlock persists indefinitely. Unlike other concurrency issues, deadlocks have no universally efficient solution.

7.1 What Is A Deadlock Scenario?

7.1.1 4-Way Stop



(a) Deadlock possible (b) Deadlock

All deadlocks arise from conflicting demands for resources by two or more processes. A typical example of deadlock is traffic congestion. Figure 6.1a illustrates a scenario where four cars arrive at a four-way stop intersection simultaneously. The intersection is divided into four quadrants, each representing a resource required for movement. Specifically, if all four cars wish to proceed straight through the intersection, their resource requirements are as follows:

- Car 1, traveling north, requires quadrants a and b.
- Car 2 requires quadrants b and c.
- Car 3 requires quadrants c and d.
- Car 4 requires quadrants d and a.

If all four cars arrive at roughly the same time, each car will yield, resulting in a potential deadlock. This deadlock remains only a possibility, as the necessary resources (quadrants) are still available for any vehicle to proceed. If one car eventually moves, the deadlock is resolved.

In contrast, if all four cars ignore the rules and cautiously proceed into the intersection simultaneously, each car will acquire one quadrant but will be unable to advance further because the second required quadrant is already occupied by another car. This situation constitutes an actual deadlock, as no vehicle can proceed until a resource is released.

7.1.2 Two Processes and Two competing Resources

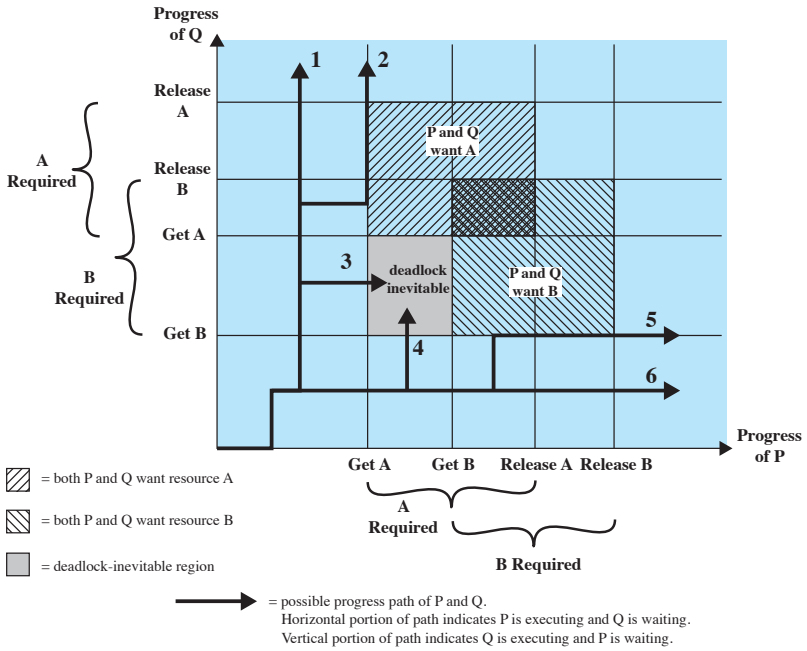
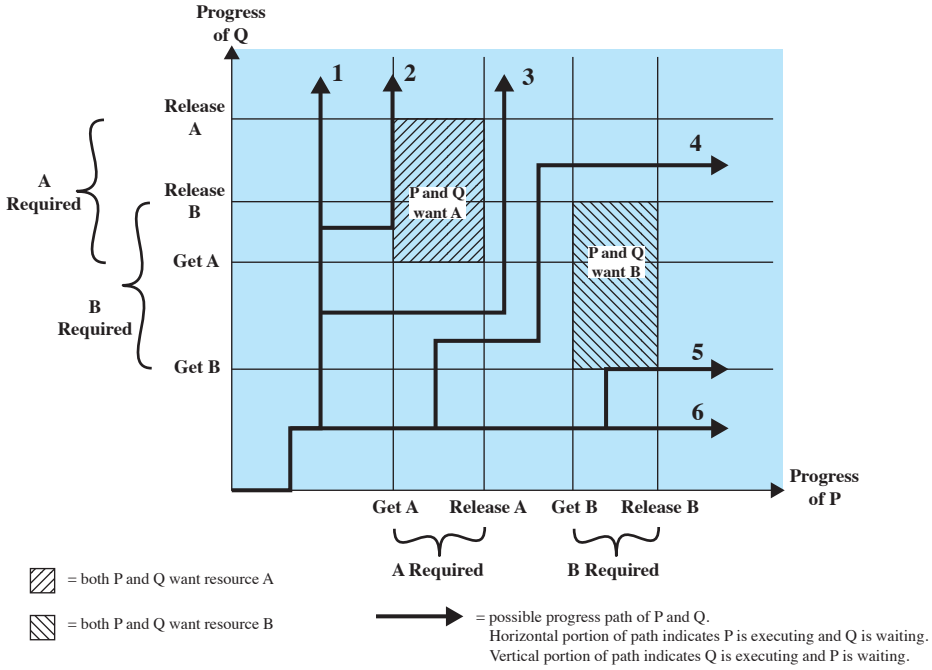


Figure 20: Example of Deadlock

This diagram represents the execution progress of two processes competing for two resources. Each process requires exclusive access to both resources for a certain period. The x-axis denotes the progress of process P, while the y-axis represents the progress of process Q. The combined execution of both processes is depicted as a trajectory moving northeast from the origin. Since the system is assumed to be uniprocessor, only one process executes at a time. As a result, the trajectory consists of alternating horizontal and vertical segments—horizontal segments indicate periods when P executes while Q waits, and vertical segments indicate periods when Q executes while P waits.

The diagram illustrates six distinct execution paths, which can be summarized as follows:

1. Q acquires B, then A, and subsequently releases both resources. When P resumes, it can acquire both resources without issue.
2. Q acquires B and then A, while P executes and blocks when requesting A. After Q releases both resources, P can proceed and acquire them.
3. Q acquires B while P acquires A. Deadlock becomes inevitable because Q will block waiting for A and P will block waiting for B.
4. P acquires A while Q acquires B. As execution continues, Q will block on A and P will block on B, leading to deadlock.
5. P acquires A, then B, while Q executes and blocks waiting for B. After P releases both resources, Q can proceed.
6. P acquires A and B, then releases both resources. Once Q resumes execution, it can acquire the required resources.



A careful analysis reveals that, irrespective of the relative timing of the two processes, deadlock cannot occur in this particular scenario.

7.1.3 Types of Resource

Resources can generally be classified into two categories: *reusable* and *consumable*.

A **reusable resource** is one that can be safely utilized by only one process at a time and is not depleted by its usage. Processes acquire resource units and later release them, making them available for other processes. Examples of reusable resources include processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores.

A **consumable resource**, on the other hand, is one that can be generated (produced) and removed (consumed). Examples of consumable resources include interrupts, signals, messages, and data stored in I/O buffers.

7.1.4 Deadlock Involving Reusable Resource

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

An example of deadlock involving *reusable resources* can be observed when two processes compete for exclusive access to a disk file D and a tape drive T. Deadlock arises when each process holds one resource while simultaneously requesting the other. Specifically, deadlock occurs if the multiprogramming system interleaves the execution of the two processes in the following order:

$$p_0 \quad p_1 \quad q_0 \quad q_1 \quad p_2 \quad q_2$$

Another example is: A total of 200 Kbytes of memory is available for allocation. The following sequence of events takes place:

P1
...
Request 80 Kbytes;
...
Request 60 Kbytes;

P2
...
Request 70 Kbytes;
...
Request 80 Kbytes;

A deadlock occurs if both processes reach their second memory request.

7.1.5 Deadlock Involving Consumable Resource

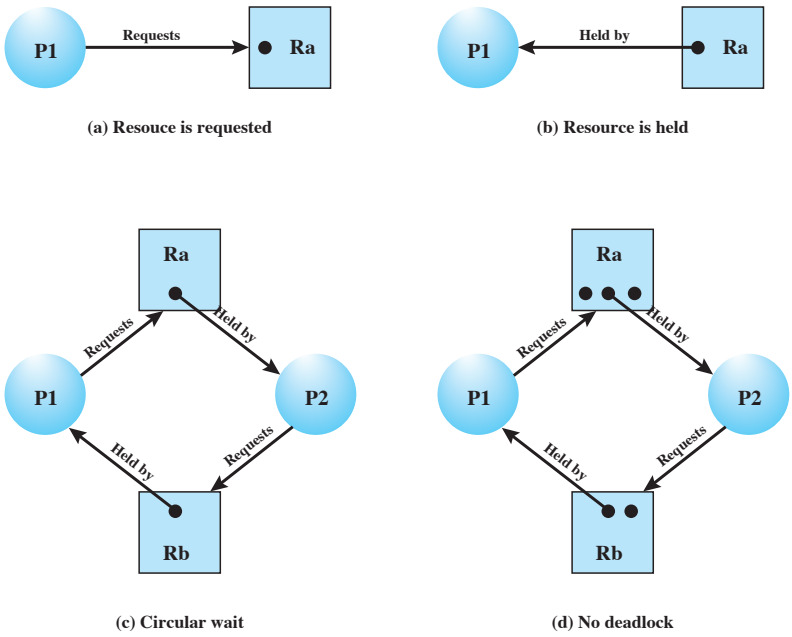
Consider a scenario involving two processes, where each process first attempts to receive a message from the other process before sending a message in return:

P1
...
Receive (P2);
...
Send (P2, M1);

P2
...
Receive (P1);
...
Send (P1, M2);

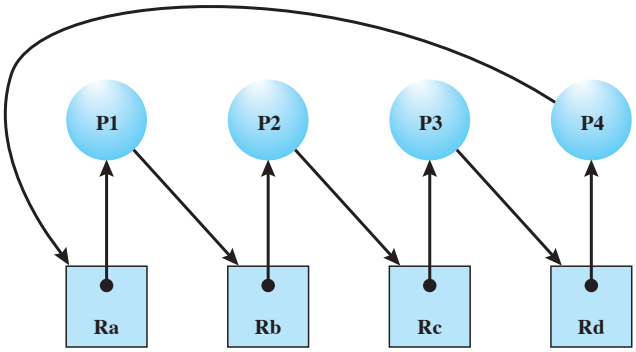
A deadlock occurs if the `Receive` operation is blocking. The deadlock arises due to a design flaw in which both processes wait indefinitely for a message that is never sent. Such errors can be subtle and challenging to identify. Moreover, deadlock may only occur under rare and specific conditions.

7.1.6 Resource Allocation Graph



In a resource allocation graph, a directed edge from a process node to a resource node represents a resource that has been requested by the process but has not yet been granted. Each resource node contains dots, where each dot represents an instance of that resource. Some resource types, such as I/O devices managed by the operating system's resource management module, may have multiple instances.

A directed edge from a dot within a reusable resource node to a process node indicates that the request has been granted, meaning the process has been allocated one unit of the resource.



The above resource allocation graph corresponds to the deadlock situation in the 4-way stop scenario. There is a circular chain of processes and resources that results in deadlock.

7.2 Approaches to Handling Deadlock

There is no single universal strategy that effectively addresses all types of deadlock. However, three common approaches exist:

- **Deadlock Prevention**: Prevent deadlock by eliminating one of the three necessary conditions for its occurrence or by ensuring that a circular wait condition never arises.
- **Deadlock Avoidance**: Deny resource requests if granting them could potentially lead to a deadlock situation.
- **Deadlock Detection**: Allow resource allocation when possible but periodically check for deadlock and implement recovery measures when it is detected.

7.2.1 Necessary and Sufficient Conditions for Deadlock

For a deadlock to be possible, the following three policy conditions must be present:

1. **Mutual Exclusion**: At any given time, only one process can utilize a resource. A process cannot access a resource unit that is currently allocated to another process.
2. **Hold and Wait**: A process holding allocated resources may continue to request additional resources without releasing its currently held resources.
3. **No Preemption**: Resources cannot be forcibly taken from a process that holds them.

These conditions are often desirable in many systems. For instance, mutual exclusion is crucial for ensuring the consistency of results and maintaining database integrity. Similarly, preemption should not occur arbitrarily—when dealing with data resources, preemption must be accompanied by a rollback recovery mechanism that restores a process and its resources to a previous valid state, allowing the process to resume execution correctly.

While these three conditions are necessary, they are not sufficient for deadlock to occur. An additional condition is required:

1. **Circular Wait**: A closed chain of processes exists, where each process holds at least one resource required by the next process in the chain.

The circular wait condition is a natural consequence of the first three conditions. If mutual exclusion, hold and wait, and no preemption exist, a sequence of events may unfold that results in an unresolvable circular wait. This unresolvable circular wait is, in fact, the definition of deadlock. The presence of all four conditions together constitutes the necessary and sufficient conditions for deadlock.

7.2.2 Deadlock Prevention Strategies

The fundamental idea behind deadlock prevention is to design a system in such a way that deadlock cannot occur. Deadlock prevention methods can be classified into two categories:

- **Indirect prevention**: Preventing the occurrence of at least one of the first three necessary conditions for deadlock.
- **Direct prevention**: Eliminating the possibility of circular wait.

Mutual Exclusion

In most cases, the requirement of mutual exclusion cannot be eliminated. If a resource must be accessed exclusively, the operating system must enforce mutual exclusion. Some resources, such as files, may permit multiple concurrent reads but require exclusive access for writes. Even in such cases, deadlock can still occur if multiple processes request write access simultaneously.

Hold and Wait

The hold-and-wait condition can be avoided by requiring that a process request all its needed resources at once, blocking it until all requests can be satisfied simultaneously. However, this approach has significant drawbacks:

- A process may experience unnecessary delays waiting for all requested resources, even when it could proceed with a subset.
- Resources allocated to a process may remain idle for long periods, preventing other processes from utilizing them.
- A process may not always know in advance all the resources it will require.

This issue becomes even more complex in modular or multithreaded applications, where resource needs are determined dynamically across different modules or execution threads.

No Preemption

This condition can be prevented through several strategies:

- If a process is denied a resource request, it must release all currently held resources and attempt to request them again, including the new resource.
- If a process requests a resource currently held by another process, the operating system can preempt the second process, forcing it to release its resources.

However, this approach is effective only if no two processes have equal priority. Furthermore, preemption is feasible only for resources whose state can be easily saved and restored, such as the CPU.

Circular Wait

Circular wait can be prevented by enforcing a strict ordering of resource types. Specifically, if a process holds resources of type R, it may only request resources of a type that appears later in the predefined ordering.

To illustrate, consider associating an index with each resource type, such that resource R_i precedes R_j if $i < j$. Suppose two processes, A and B, enter a deadlock scenario where:

- A holds R_i and requests R_j.
- B holds R_j and requests R_i.

This scenario would be impossible because it implies both $i < j$ and $j < i$, which is contradictory.

Despite being an effective strategy, preventing circular wait may lead to inefficiencies, as it can slow down processes and restrict resource access unnecessarily.

7.2.3 Deadlock Avoidance Strategies

Deadlock avoidance is an alternative approach to handling deadlock that differs subtly from deadlock prevention. In **deadlock prevention**, resource requests are constrained to eliminate at least one of the four necessary conditions for deadlock. While effective, deadlock prevention often leads to inefficient resource utilization and process execution.

In contrast, **deadlock avoidance** allows the three necessary conditions to exist but makes careful resource allocation decisions to ensure that the system never enters a deadlock state. Because avoidance does not impose strict constraints on resource requests, it allows for greater concurrency than prevention.

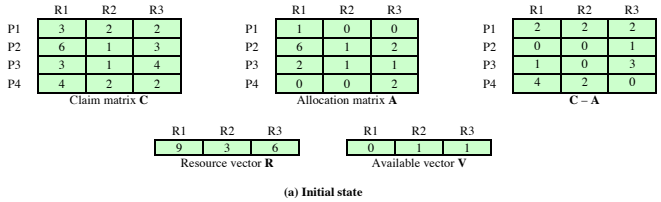
Deadlock avoidance dynamically determines whether granting a resource request could potentially lead to deadlock. **This approach requires knowledge of future resource requests by processes.**

- **Resource Allocation Denial**: A process is denied an incremental resource request if granting it might lead to deadlock.
- **Process Initiation Denial**: A process is not started if its resource demands might result in deadlock.

The resource allocation denial strategy, commonly known as the **Banker's Algorithm**. Consider a system with a fixed number of processes and a fixed number of resources. At any given time, a process may hold zero or more allocated resources. The *state of the system* represents the current allocation of resources to processes. This state is defined by:

- Two vectors: **Resource** and **Available**.
- Two matrices: **Claim** and **Allocation**.

A **safe state** is one in which there exists at least one sequence of resource allocations that ensures all processes can complete without leading to deadlock. Conversely, an **unsafe state** is one that does not guarantee the system can avoid deadlock.



(a) Initial state

The system consists of four processes and three resource types. The total available units of resources R_1, R_2, and R_3 are 9, 3, and 6, respectively. At the current state, resource allocations have already been made to the four processes, leaving 1 unit of R_2 and 1 unit of R_3 available.

To determine whether this is a **safe state**, we need to answer an intermediate question: **Can any of the four processes complete with the currently available resources?** In other words, can the difference between a process's maximum requirement and its current allocation be satisfied using the available resources?

Mathematically, for a process P_i to proceed, the following condition must hold for all resource types j:

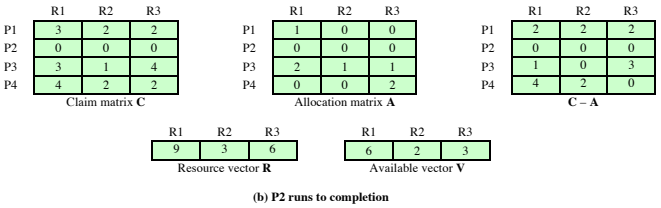
$$C_{ij} - A_{ij} \leq V_j \quad \forall j$$

where:

- C_{ij} is the maximum claim of process P_i for resource R_j.
- A_{ij} is the current allocation of resource R_j to process P_i.
- V_j is the number of currently available units of resource R_j.

Examining this condition for P_1, we observe that P_1 holds only 1 unit of R_1 but requires an additional 2 units of R_1, along with 2 units each of R_2 and R_3. Since these demands exceed the available resources, P_1 cannot proceed at this time.

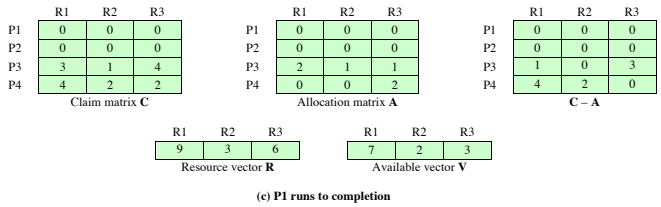
However, allocating one unit of R_3 to P_2 would fulfill its maximum resource requirements, allowing P_2 to complete execution. Once P_2 finishes, its allocated resources are released back into the system, increasing the available resource pool.



(b) P2 runs to completion

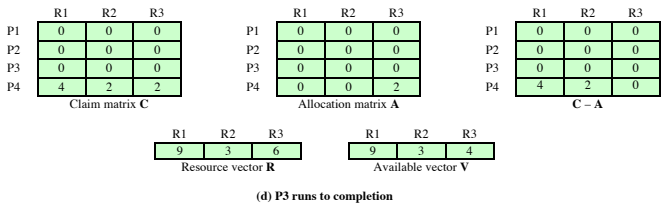
After completing process P_2, the system transitions to a new state. In this updated state, each of the remaining processes has sufficient resources available to complete execution.

Suppose we select P_1 for execution. We allocate the necessary resources to P_1, allowing it to complete. Once P_1 finishes execution, all resources held by P_1 are returned to the pool of available resources.



(c) P1 runs to completion

Next, we can complete P3.



(d) P3 runs to completion

At this stage, process P_4 can also complete execution.

Once P_4 finishes, all processes have successfully run to completion. Therefore, the initial state is confirmed to be a **safe state**.

The strategy follows these steps:

1. When a process requests a set of resources, assume the request is granted and update the system state accordingly.
2. Evaluate whether the new system state remains a safe state.
3. If the resulting state is safe, grant the resource request.
4. If the resulting state is unsafe, deny the request and block the process until it is safe to proceed.

Deadlock avoidance offers the advantage of preventing deadlock **without requiring process preemption and rollback**, as is necessary in deadlock detection. Additionally, it imposes **fewer constraints** than deadlock prevention. However, deadlock avoidance comes with several limitations:

- The maximum resource requirement for each process must be known in advance.
- The processes must be **independent**, meaning their execution order cannot be dictated by synchronization constraints.
- The system must have a **fixed number of resources** available for allocation.
- No process may terminate while still holding allocated resources.

Below is an example of Banker's algorithm:

```c
#include <stdio.h>
#include <stdbool.h>

#define P 5 // Number of processes
#define R 3 // Number of resources


void calculateNeed(int need[P][R], int claim[P][R], int alloc[P][R]) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = claim[i][j] - alloc[i][j];
}

bool findSafeSeq(int avail[R], int claim[P][R], int alloc[P][R]) {
    int need[P][R];
    calculateNeed(need, claim, alloc);

    bool finished[P] = {0}; // track of which processes have completed
    int temp_avail[R]; // This vector represents the available resources at any point during the algorithm's execution.
    for (int i = 0; i < R; i++)
        temp_avail[i] = avail[i];

    int count = 0;
    /*The algorithm enters a loop that continues until all processes are marked as finished. Inside the loop, it
     * searches for a process p that is not yet finished (finished[p] == false) and whose resource needs can be
     * satisfied with the current work vector (i.e., for all j, need[p][j] <= work[j]). If such a process is found,
     * the algorithm simulates the execution of this process by adding its allocated resources back to the work
     * vector (since the process would release these resources upon completion). The process is then marked as
     * finished (finished[p] = true). The loop continues, each time looking for another process that can be satisfied
     * with the updated work vector.
     * */
    while (count < P) {
```

```
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (!finished[p]) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > temp_avail[j])
                        break;

                if (j == R) {
                    for (int k = 0; k < R; k++)
                        temp_avail[k] += alloc[p][k];

                    finished[p] = true;
                    found = true;
                    count++;
                    printf("Process %d is run to completion!\n", p);
                }
            }
        }

        if (!found) {
            return false; // System is not in a safe state
        }
    }

    return true; // If all processes could finish, system is in a safe state
}

bool isRequestSafe(int request[R], int pID, int avail[R], int alloc[P][R], int claim[P][R]) {
    int need[P][R];
    calculateNeed(need, claim, alloc);

    // Check if request is less than or equal to need
    for (int i = 0; i < R; i++) {
        if (request[i] > need[pID][i]) {
            printf("Process has exceeded its maximum claim.\n");
            return false;
        }
    }

    // Check if request is less than or equal to available
    for (int i = 0; i < R; i++) {
        if (request[i] > avail[i]) {
            printf("Resources are not available.\n");
            return false;
        }
    }

    // Try to allocate requested resources temporarily
    for (int i = 0; i < R; i++) {
        avail[i] -= request[i];
        alloc[pID][i] += request[i];
    }

    // Check if system is still in a safe state after allocation
    bool safe = findSafeSeq(avail, claim, alloc);

    // Rollback the allocation for next checks
    for (int i = 0; i < R; i++) {
        avail[i] += request[i];
        alloc[pID][i] -= request[i];
    }

    return safe;
}

```

```c
int main() {
    int processes[] = {0, 1, 2, 3, 4};

    // Example matrices and vectors
    int claim[P][R] = {{7, 5, 3},
                       {3, 2, 2},
                       {9, 0, 2},
                       {2, 2, 2},
                       {4, 3, 3}
    };

    int alloc[P][R] = {{0, 1, 0},
                       {2, 0, 0},
                       {3, 0, 2},
                       {2, 1, 1},
                       {0, 0, 2}
    };
    int avail[] = {3, 3, 2};

    // Example request
    int request[] = {1, 2, 1}; // Request for process 1
    int pID = processes[0]; // Process making the request

    if (isRequestSafe(request, pID, avail, alloc, claim)) {
        printf("The request can be safely granted.\n");
    } else {
        printf("The request cannot be safely granted.\n");
    }

    return 0;
}
```

7.2.4 Deadlock Detection Strategies

A deadlock check can be performed as frequently as every resource request or at longer intervals, depending on the likelihood of deadlock occurrence. Checking at each resource request offers two advantages: **it enables early detection and relies on a relatively simple algorithm based on incremental system state changes.** However, performing checks this frequently incurs significant processor overhead.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Above figure can be used to demonstrate the deadlock detection algorithm, which proceeds as follows:

1. Mark P_4 since it has no allocated resources.
2. Initialize $W = (0, 0, 0, 0, 1)$.
3. The request of process P_3 is less than or equal to W, so mark P_3 and update W as follows:

$$W = W + (0, 0, 0, 1, 0) = (0, 0, 0, 1, 1).$$

4. No other unmarked process has a row in Q that is less than or equal to W. Thus, the algorithm terminates.

At the end of the algorithm, processes P_1 and P_2 remain unmarked, indicating that they are deadlocked.

Wait-for Graph (WFG): A directed graph in which nodes represent processes in the system. An edge from node A to node B indicates that process A is waiting for a resource currently held by process B.

Construction: The graph is derived from the current allocation of resources to processes, along with their pending resource requests.

Nodes: Each active process in the system is represented as a node in the graph.

Edges: If a process P_1 holds a resource R that has been requested by process P_2 (and P_2 is waiting for R to be released by P_1), then a directed edge from P_2 to P_1 is added to the graph.

Cycle Detection: A deadlock is present in the system if and only if the Wait-for Graph (WFG) contains at least one cycle. A cycle in the WFG signifies a set of processes that are mutually waiting for resources in a circular dependency, preventing further progress.

Algorithm: To identify deadlocks, the system periodically constructs the WFG and examines it for cycles. Various cycle detection algorithms, such as Depth-First Search (DFS), can be employed for this purpose.

Below is an example that constructs WFG and detect cycles.

```c
#include <stdio.h>
#include<stdbool.h>

#define P 5  // Number of processes
#define R 3  // Number of resource types

int WFG[P][P] = {{0}};  // Wait-For-Graph

void populateWFG(int req[P][R], int avail[R], int alloc[P][R]) {
    for (int i = 0; i < P; ++i) {
        for (int j = 0; j < R; ++j) {
            if (req[i][j] > avail[j]) {
                for (int k = 0; k < P; ++k) {
                    if (alloc[k][j] > 0) {
                        WFG[i][k] = 1;  // Process i is waiting for resources held by process k
                    }
                }
                /*
                 * At this point, even if process i is waiting for additional resources, the critical information
                 * that it is waiting (and hence a potential participant in a deadlock) is already captured by adding
                 * the relevant edges in the WFG. Adding more edges for other resources it's waiting for won't change
                 * its status as a waiting process nor will it affect the detection of a cycle (deadlock) in the graph,
                 * because a deadlock is characterized by a set of processes waiting for each other in a circular
                 * chain, regardless of the specific resources they're waiting for. Therefore, once a process is
                 * identified as waiting for a resource, further checks for other resources in the context of
                 * populating the WFG for deadlock detection are redundant.
                 */
                break;
            }
        }
    }
}


int dfs(int node, int visited[], int current_path[]) {
    if (!visited[node]) {
        visited[node] = 1;
        current_path[node] = 1;

        for (int i = 0; i < P; i++) {
            // Check if there's an edge from the current node to node 'i'
            if (WFG[node][i]) {
                // If 'i' is on the current path, a cycle is detected
                if (current_path[i]) return true;

                // If 'i' has not been visited, visit it and check for a cycle
```

```
                if (!visited[i])
                    if (dfs(i, visited, current_path)) return true;
            }
        }
    }
    current_path[node] = 0;  // Remove the node from recursion stack
    return 0;
}

bool isDeadlock() {
    int visited[P] = {0};
    int current_path[P] = {0};
    //ensure that all components of the graph are explored, particularly in cases where the graph might be disconnected
    for (int i = 0; i < P; i++) {
        if (dfs(i, visited, current_path))
            return true;  // Deadlock detected
    }

    return false;  // No deadlock
}

int main() {
    int req[P][R] = {
            {0, 0, 1},  // P0
            {1, 0, 0},  // P1
            {0, 1, 0}   // P2
    };
    int avail[R] = {0, 0, 0};


    int alloc[P][R] = {
            {1, 1, 0},  // P0
            {0, 2, 1},  // P1
            {1, 1, 0}   // P2
    };

    // Populate the WFG based on current resource allocation state
    populateWFG(req, avail, alloc);

    // Check for deadlocks
    if (isDeadlock()) {
        printf("Deadlock detected.\n");
    } else {
        printf("No deadlock detected.\n");
    }

    return 0;
}
```

7.2.5 Deadlock Recovery

Once a deadlock is detected, a recovery strategy must be implemented. The following approaches, listed in increasing order of sophistication, can be considered:

1. **Terminate all deadlocked processes.** Surprisingly, this is one of the most commonly used solutions in operating systems.
2. **Rollback each deadlocked process to a previously defined checkpoint and restart all processes.** This approach requires built-in rollback and restart mechanisms. However, there is a risk that the deadlock may recur. Nevertheless, the nondeterministic nature of concurrent processing may prevent this from happening.
3. **Gradually abort deadlocked processes until the deadlock is resolved.** The selection of processes for termination should be based on a cost-minimization criterion. After each termination, the deadlock detection algorithm must be rerun to determine whether deadlock still exists.

4. **Gradually preempt resources until the deadlock is resolved.** Similar to approach (3), the selection of resources for preemption should follow a cost-based strategy, and the detection algorithm must be rerun after each preemption. Any process that loses a resource must be rolled back to a state prior to acquiring that resource.

For strategies (3) and (4), the selection criteria for process termination or resource preemption may include:

- The process that has consumed the least processor time so far.
- The process that has produced the least amount of output.
- The process with the highest estimated remaining execution time.
- The process with the fewest total resources allocated.
- The process with the lowest priority.

Some of these metrics are easier to measure than others. In particular, estimating the remaining execution time is highly uncertain. Additionally, aside from using priority as a measure, these criteria primarily assess the cost to the system rather than the impact on the user.

Instead of designing an operating system facility that strictly follows a single deadlock management strategy, a more efficient approach is to apply different strategies depending on the situation.

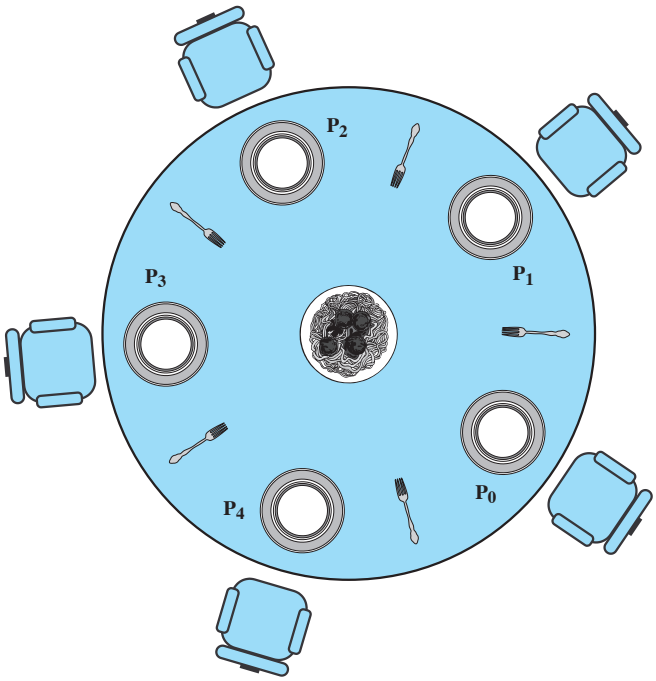
- Classify resources into distinct **resource classes**.
- Utilize the previously discussed **linear ordering strategy** to prevent circular wait conditions and thus avoid deadlocks between different resource classes.
- Within each resource class, implement the most suitable deadlock prevention or recovery algorithm based on the characteristics of that class.

Resource Classes and Strategies:

- **Swappable Space:**
  - Deadlock prevention can be achieved by requiring that all necessary resources be allocated at once, following the hold-and-wait prevention strategy.
  - This approach is effective when maximum storage requirements are known in advance.
- **Process Resources:**
  - Deadlock avoidance is often effective in this category since processes can typically declare their required resources ahead of time.
  - Deadlock prevention using resource ordering within this class is also feasible.
- **Main Memory:**
  - Deadlock prevention through preemption is the most suitable approach.
  - When a process is preempted, it can be swapped to secondary storage, thereby freeing memory to resolve the deadlock.
- **Internal Resources:**
  - Deadlock prevention can be implemented using resource ordering.

7.3 Dining Philosophers Problem

The dining philosophers problem, originally introduced by Dijkstra, presents a classic synchronization challenge. The scenario involves five philosophers residing in a house, where a table is set for their meals. Each philosopher's life revolves around two primary activities: thinking and eating. Through extensive contemplation, they have collectively agreed that spaghetti is the only food that enhances their intellectual endeavors. However, due to their lack of manual dexterity, each philosopher requires two forks to eat.



The dining arrangement is straightforward (see Figure 6.11): a circular table is set with a large bowl of spaghetti, five plates (one for each philosopher), and five forks. When a philosopher wishes to eat, they must sit at their designated place, take the two forks adjacent to their plate, and proceed to eat.

7.4 The Problem Statement

The challenge is to develop an algorithm that allows the philosophers to eat while adhering to the following constraints:

- **Mutual Exclusion:** No two philosophers can use the same fork simultaneously.
- **Deadlock Prevention:** The system must not reach a state where all philosophers are indefinitely waiting for forks.
- **Starvation Avoidance:** Every philosopher should eventually get a chance to eat, preventing indefinite waiting.

7.5 Relevance and Importance

Although this problem may seem abstract, it effectively illustrates fundamental issues related to deadlock and starvation. The dining philosophers problem serves as a representative case for managing shared resource coordination, a common challenge in applications involving concurrent threads. Consequently, it is widely used as a benchmark for evaluating synchronization strategies.

7.5.1 Semophore Solution

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h> // For O_CREAT, O_EXEC

#define NUM_PHILOSOPHERS 5
int philosopher_number[NUM_PHILOSOPHERS];

// Names for the semaphores
char fork_names[NUM_PHILOSOPHERS][20];
char room_name[] = "/room_sem";

// Semaphore pointers for forks and room
```

```c
sem_t* forks[NUM_PHILOSOPHERS];
sem_t* room;

void take_fork(int phil) {
    sem_wait(forks[phil]); // Take left fork
    sem_wait(forks[(phil + 1) % NUM_PHILOSOPHERS]); // Take right fork

    printf("Philosopher %d is eating.\n", phil + 1);
}

void put_fork(int phil) {
    sem_post(forks[phil]); // Put doum left fork
    sem_post(forks[(phil + 1) % NUM_PHILOSOPHERS]); // Put doum right fork

    printf("Philosopher %d is thinking.\n", phil + 1);
}

void* philosopher(void* num) {
    while (1) {
        int* i = num;

        sleep(1);
        sem_wait(room); // Enter critical region
        take_fork(*i);
        sleep(1); // Eating
        put_fork(*i);
        sem_post(room); // Leave critical region
    }
}

int main() {
    pthread_t thread_id[NUM_PHILOSOPHERS];

    // Initialize named semaphores
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        snprintf(fork_names[i], sizeof(fork_names[i]), "/fork_sem_%d", i);
        if ((forks[i] = sem_open(fork_names[i], O_CREAT, 0666, 1)) == SEM_FAILED) {
            perror("sem_open");
            exit(EXIT_FAILURE);
        }
        philosopher_number[i] = i;
    }

    if ((room = sem_open(room_name, O_CREAT, 0666, NUM_PHILOSOPHERS - 1)) == SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &philosopher_number[i]);
        printf("Philosopher %d is thinking.\n", i + 1);
    }

    // Join philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(thread_id[i], NULL);
    }

    // Close and unlink semaphores
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_close(forks[i]);
        sem_unlink(fork_names[i]);
    }
    sem_close(room);
    sem_unlink(room_name);
```

```
    return 0;
}
```

7.5.2 Monitor Solution

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#define NUM_PHILOSOPHERS 5

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t forks_ready[NUM_PHILOSOPHERS];
    int forks[NUM_PHILOSOPHERS];
} Pho_Monitor;

Pho_Monitor *pho_monitor = NULL;

static void init_monitor() {
    pho_monitor = (Pho_Monitor*)malloc(sizeof(Pho_Monitor));
    pthread_mutex_init(&pho_monitor->mutex, NULL);
    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        pthread_cond_init(&pho_monitor->forks_ready[i], NULL);
        pho_monitor->forks[i] = 1; // Initialize all forks as available
    }
}

static void pickup(int philosopher_id) {
    pthread_mutex_lock(&pho_monitor->mutex);
    if (!pho_monitor->forks[philosopher_id]) {
        pthread_cond_wait(&pho_monitor->forks_ready[philosopher_id], &pho_monitor->mutex);
    }
    pho_monitor->forks[philosopher_id] = 0; // Mark the left fork as unavailable
    if (!pho_monitor->forks[(philosopher_id + 1) % NUM_PHILOSOPHERS]) {
        pthread_cond_wait(&pho_monitor->forks_ready[(philosopher_id + 1) % NUM_PHILOSOPHERS], &pho_monitor->mutex);
    }
    pho_monitor->forks[(philosopher_id + 1) % NUM_PHILOSOPHERS] = 0; // Mark the right fork as unavailable
    pthread_mutex_unlock(&pho_monitor->mutex);
    printf("Philosopher %d is eating.\n", philosopher_id);
}

static void putdown(int philosopher_id) {
    pthread_mutex_lock(&pho_monitor->mutex);
    pho_monitor->forks[philosopher_id] = 1; // Mark the left fork as available
    pthread_cond_signal(&pho_monitor->forks_ready[philosopher_id]); // Signal the right philosopher
    pho_monitor->forks[(philosopher_id + 1) % NUM_PHILOSOPHERS] = 1; // Mark the right fork as available
    pthread_cond_signal(&pho_monitor->forks_ready[(philosopher_id + 1) % NUM_PHILOSOPHERS]); // Signal the right philosopher
    pthread_mutex_unlock(&pho_monitor->mutex);
}

static void* philosopher(void *arg) {
    int id = (int)(intptr_t)arg;

    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(1); // thinking for 1 second
        pickup(id);
        sleep(1); // eating for 1 second
        putdown(id);
    }
    return NULL;
```

```
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    init_monitor();

    printf("start!\n");
    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        pthread_create(&philosophers[i], NULL, philosopher, (void*)(intptr_t)(i + 1));
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; ++i)
        pthread_join(philosophers[i], NULL);

    return 0;
}
```

7.6 Deadlock Strategies Used in Linux 0.11/0.12

Unlike modern operating systems, Linux 0.11/0.12 did not implement mechanisms to prioritize critical processes for deadlock resolution. The kernel did not explicitly track circular wait conditions, relying instead on preemption and scheduling policies. To address deadlock, Linux employs a simple timeout-based approach or process scheduling to resolve cyclic waiting conditions.

- **Round-Robin Scheduling:** Ensures that no single process can indefinitely block others.
- **Preemptive Multitasking:** If a process fails to release a resource, the kernel can forcibly preempt it.

```
// This code is also looped from the last task of the task array and skips the empty
// slots. It compares the counter (the countdown number of task run time) of each ready
// state task. Which value is large, it means that there are still many running times of
// the task, and next points to the task number of that task.
while (--i) {
    if (!*--p)
        continue;
    if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
        c = (*p)->counter, next = i;
 }
```

From the code above we can see that if a process becomes stuck in a deadlock state, the scheduler will eventually preempt it. The `counter` variable ensures that no single process monopolizes CPU time, thereby reducing the risk of indefinite waiting. Since preemption was based solely on CPU time counters, lower-priority processes could suffer indefinite postponement.

8 Memory Management

Memory management mechanism is intended to satisfy the following requirements:

- **Relocation**:

  Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program. Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization. Once a program has been swapped out to disk, it would be nearly impossible to specify that the next time it is swapped back it must be placed in the same area of main memory as before. Therefore, we may need to relocate the process to a different memory area.

- **Protection**

  Each process should be protected against unwanted interference by other processes, whether accidental or intentional.

  - Processes need to acquire permission to reference memory locations for reading or writing purposes.
  - Location of a program in main memory is unpredictable. (cannot be checked at compile time to assure protection).
  - Memory references generated by a process must be checked at runtime mechanisms that support relocation also support protection.

  Note that the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software). This is because the OS cannot anticipate all of the memory references that a program will make (very time-consuming).

- **Sharing**

  Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory.

  - Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
  - Memory management must allow controlled access to shared areas of memory without compromising protection

- **Logical organization**

  Memory is organized as linear address space, consisting of a sequence of bytes or words. While this organization closely mirrors the actual machine hardware, it does not correspond to the way in which programs are typically constructed. Most programs are organized into modules, some of which are unmodifiable (read only, execute only) and some of which contain data that may be modified. If the operating system and computer hardware can effectively deal with user programs and data in the form of modules of some sort, then a number of advantages can be realized:

  - Modules can be written and compiled independently, with all references from one module to another resolved by the system at run time.
  - With modest additional overhead, different degrees of protection (read only, execute only) can be given to different modules.
  - It is possible to introduce mechanisms by which modules can be shared among processes. The advantage of providing sharing on a module level is that this corresponds to the user's way of viewing the problem, and hence it is easy for the user to specify the sharing that is desired.

- **Physical organization**

  Computer memory is organized into at least two levels, referred to as main memory and secondary memory. Main memory provides fast access at relatively high cost. In addition, main memory is volatile; that is, it does not provide permanent storage. Secondary memory is slower and cheaper than main memory and is usually not volatile. Thus secondary memory of large capacity can be provided for long-term storage of programs and data, while a smaller main memory holds programs and data currently in use.

  In this two-level scheme, the organization of the flow of information between main and secondary memory is a major system concern. The responsibility for this flow could be assigned to the individual programmer, but this is impractical and undesirable for two reasons:

  1. The main memory available for a program plus its data may be insufficient.

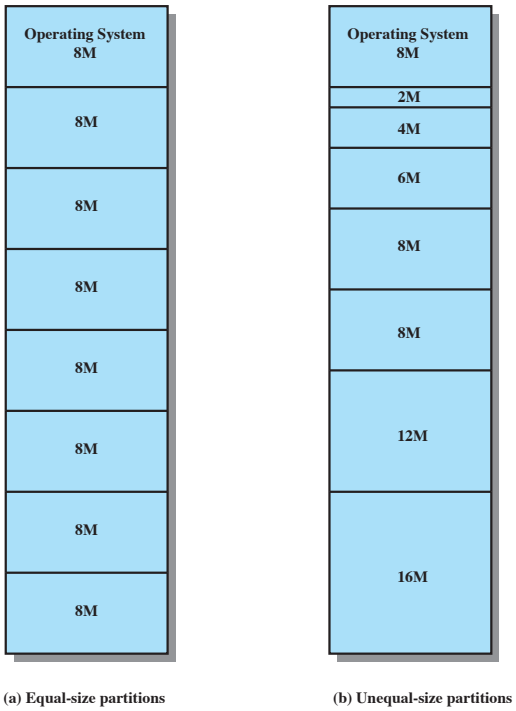
2. In a multiprogramming environment, the programmer does not know at the time of coding how much space will be available or where that space will be.

8.1 Memory Paritioning

8.1.1 Fixed Equal Partitioning

The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries. If a process's size exceeds the size of the largest available partition, the operating system might reject the process or terminate it if it's already running.

Equal-Size Fixed Partitions



(a) Equal-size partitions (b) Unequal-size partitions

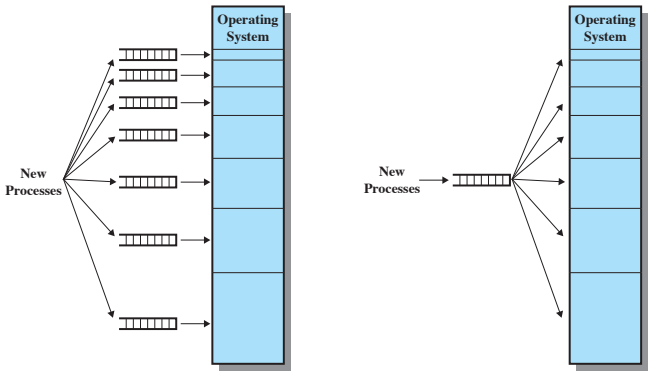
PLACEMENT ALGORITHM: As long as there is any available partition, a process can be loaded into that partition.

There are **two difficulties** with the use of equal-size fixed partitions:

(1) A program may be too big to fit into a partition. In this case, the programmer must design the program with the use of **overlays** so that only a portion of the program need be in main memory at any one time. When a module is needed that is not present, the user's program must load that module into the program's partition, overlaying whatever programs or data are there.

(2) Main memory utilization is **extremely inefficient**. Any program, no matter how small, occupies an entire partition. In our example, there may be a program whose length is less than 2 Mbytes; yet it occupies an 8-Mbyte partition whenever it is swapped in. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as **internal fragmentation**.

Unequal-Size Fixed Partitions



(a) One process queue per partition (b) Single queue

PLACEMENT ALGORITHM:

(1)The simplest way is to **assign each process to the smallest partition within which it will fit**. In this case, a scheduling queue is needed for each partition, to hold swapped-out processes destined for that partition (a). The advantage of this approach is that processes are always assigned in such a way as to minimize wasted memory within a partition (internal fragmentation).

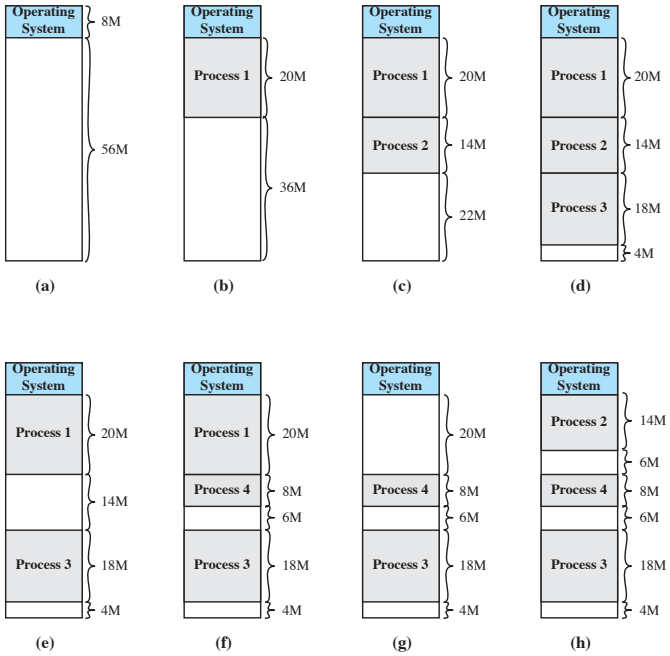
(2) Consider a case in which there are no processes with a size between 12 and 16M at a certain point in time. In that case, the 16M partition will remain unused, even though some smaller process could have been assigned to it. Thus, a preferable approach would be to employ a single queue for all processes (b). When it is time to load a process into main memory, **the smallest available partition that will hold the process is selected**.

Disadvantages of Fixed Partitions

- The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.
- Because partition sizes are preset at system generation time, small jobs will not utilize partition space efficiently. In an environment where the main storage requirement of all jobs is known beforehand, this may be reasonable, but in most cases, it is an inefficient technique.

8.1.2 Dynamic Partitioning

With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.



- (a)Initially, main memory is empty, except for the OS.

- (b, c, d): The first three processes are loaded in. This leaves a "hole" at the end of memory that is too small for a fourth process.

  At some point, none of the processes in memory is ready.

- (e) The operating system swaps out process 2.
- (f) load a new process 4. Because process 4 is smaller than process 2, another small hole is created.

  At some point, none of the processes in memory is ready. Process 2, in the Suspend-Ready state.

- (g) system swaps process 1 out
- (h) swaps process 2 back in.

This method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. This phenomenon is referred to as **external fragmentation**.

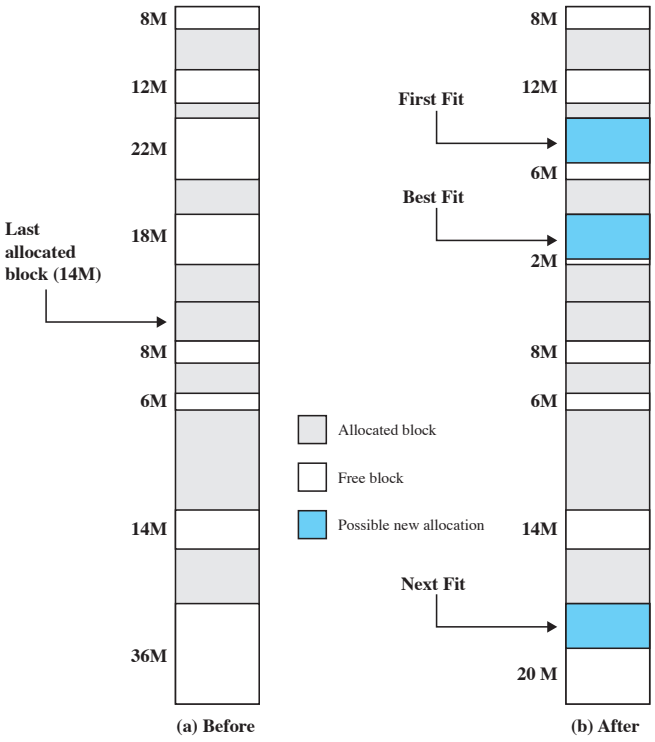
One technique for overcoming external fragmentation is **compaction** : From time to time, the OS shifts the processes so that they are contiguous and so that all of the free memory is together in one block. The difficulty with compaction is that **it is a time-consuming procedure and wasteful of processor time**. Note that compaction implies the need for a dynamic relocation capability. That is, it must be possible to move a program from one region to another in main memory without invalidating the memory references in the program.

Placement Algorithms

Because memory compaction is time consuming, the OS designer must be clever in deciding how to assign processes to memory (how to plug the holes). When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.

Three placement algorithms that might be considered are best-fit, first-fit, and next-fit. All, of course, are limited to choosing among free blocks of main memory that are equal to or larger than the process to be brought in.

- **Best-fit** chooses the block that is closest in size to the request.
- **First-fit** begins to scan memory from the beginning and chooses the first available block that is large enough.
- **Next-fit** begins to scan memory from the location of the last placement, and chooses the next available block that is large enough.



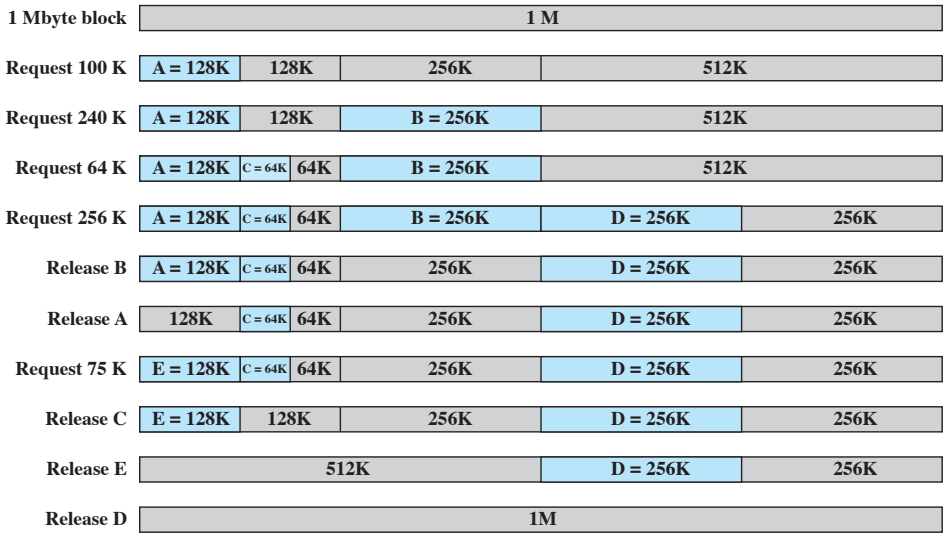
Some general comments can be made:

The first-fit algorithm is not only the simplest but usually the best and fastest as well.

The next-fit algorithm tends to produce slightly worse results than the first-fit. The largest block of free memory, which usually appears at the end of the memory space, is quickly broken up into small fragments.Compaction may be required more frequently with next-fit.

The best-fit algorithm, despite its name, is usually the **worst** performer. It guarantees that the fragment left behind is as small as possible. Main memory is quickly littered by blocks too small to satisfy memory allocation requests. Memory compaction must be done more frequently than with the other algorithms.

8.1.3 Buddy System

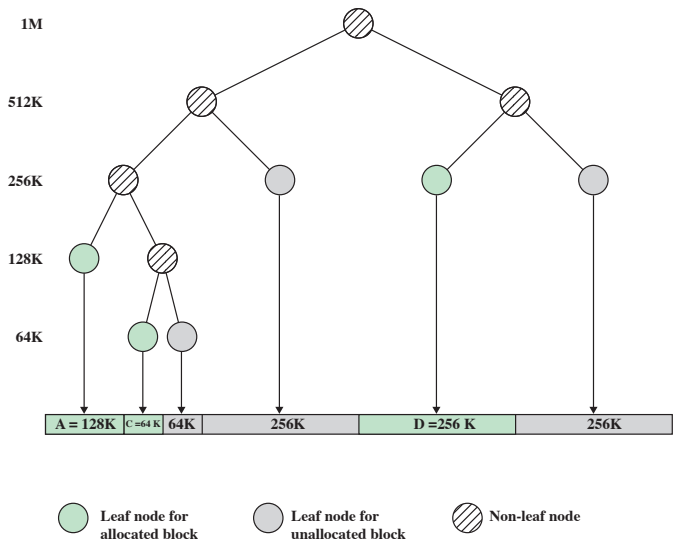


Memory blocks are available of size 2^K words, $L \leq K \leq U$, where

- 2^L = smallest size block that is allocated
- 2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation

To begin, the entire space available for allocation is treated as a single block of size 2^U. If a request of size s such that $2^{U-1} \leq s \leq 2^U$ is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size 2^{U-1}. If $2^{U-2} \leq s \leq 2^{U-1}$, then the request is allocated to one of the two buddies. Otherwise, one of the buddies is split in half again. This process continues until the smallest block greater than or equal to s is generated and allocated to the request.

At any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i. A hole may be removed from the $(i+1)$ list by splitting it in half to create two buddies of size 2^i in the i list. Whenever a pair of buddies on the i list both become unallocated, they are removed from that list and coalesced into a single block on the $(i+1)$ list.



The above figure shows a binary tree representation of the buddy allocation immediately after the Release B request. The leaf nodes represent the current partitioning of the memory.

If two buddies are leaf nodes, then at least one must be allocated; otherwise they would be coalesced into a larger block.

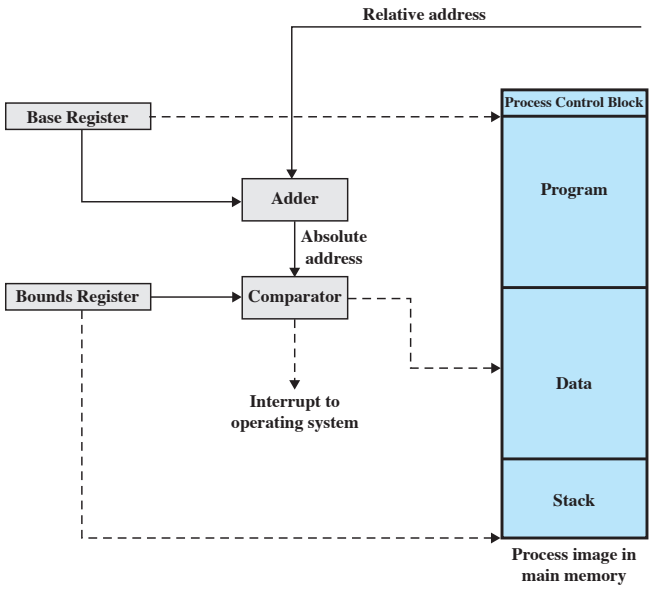
The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes: minimizes both internal and external fragmentation.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_MEM_SIZE 1024 // Maximum memory size
#define MIN_BLOCK_SIZE 32 // Minimum block size

typedef struct Block {
    int size;
    int isFree;
    struct Block *next;
    struct Block *prev;
    struct Block *buddy;
} Block;

Block *head = NULL;

// Initialize memory with a single large block
void initializeMemory() {
    head = (Block *)malloc(sizeof(Block));
    head->size = MAX_MEM_SIZE;
    head->isFree = 1;
    head->next = head->prev = head->buddy = NULL;
}

// Allocate memory using the Buddy System
void *buddySplit(int size) {
    Block *current = head;
    int adjustedSize = (int)pow(2, ceil(log(size) / log(2))); // Adjust size to next power of 2

    if (adjustedSize < MIN_BLOCK_SIZE) adjustedSize = MIN_BLOCK_SIZE;

    while (current) {
        if (current->isFree && current->size >= adjustedSize) {
            while (current->size / 2 >= size) {
                Block *buddy = (Block *)malloc(sizeof(Block));
                buddy->size = current->size / 2;
                current->size /= 2;

                buddy->next = current->next;
                buddy->prev = current;
                buddy->buddy = current;
                buddy->isFree = 1;

                if (current->next) current->next->prev = buddy;
                current->next = buddy;
                current->buddy = buddy;
            }
            current->isFree = 0;
            return current;
        }
        current = current->next;
    }
    return NULL; // No suitable block found
}

// Merge buddies if possible
```

```c
void buddyMerge(void *ptr) {
    Block *block = (Block *)ptr; // Get block metadata from memory address
    if (block->buddy && block->buddy->isFree && block->size == block->buddy->size) {
        block->isFree = 1;
        block->next = block->buddy->next;
        if (block->buddy->next) block->buddy->next->prev = block->buddy->prev;

        block->size *= 2;
        free(block->buddy);
        block->buddy = block->prev;
        block->isFree = 1;

        buddyMerge(block); // Recursively try to merge with its buddy
    }
}

void printMemoryTree() {
    Block *current = head;
    printf("Memory Blocks:\n");
    while (current != NULL) {
        printf("Block Size: %d, Status: %s\n", current->size, current->isFree ? "Free" : "Allocated");
        current = current->next;
    }
}

int main() {
    initializeMemory();

    // Example usage
    void *ptr1 = buddySplit(100);
    void *ptr2 = buddySplit(200);

    // Print the memory tree
    printMemoryTree(head, 0);

    if(ptr1 != NULL){
        buddyMerge(ptr1);
        printMemoryTree(head, 0);
    }

    if(ptr2 != NULL){
        buddyMerge(ptr2);
        printMemoryTree(head, 0);
    }
    return 0;
}
```

8.2 Paging

Typically, all of the memory references in the loaded process are relative to the origin of the program. Thus a hardware mechanism is needed for translating relative addresses to physical main memory addresses at the time of execution of the instruction that contains the reference.



When a process is assigned to the Running state, a special processor register, sometimes called the **base register**, is loaded with the starting address in main memory of the program. There is also a "bounds" register that indicates the ending location of the program; these values must be set when the program is loaded into memory or when the process image is swapped in.

During the course of execution of the process, relative addresses are encountered. These include the contents of the instruction register, instruction addresses that occur in branch and call instructions, and data addresses that occur in load and store instructions. Each such relative address goes through two steps of manipulation by the processor. First, the value in the base register is added to the relative address to produce an absolute address. Second, the resulting address is compared to the value in the bounds register. If the address is within bounds, then the instruction execution may proceed. Otherwise, an interrupt is generated to the operating system, which must respond to the error in some fashion.

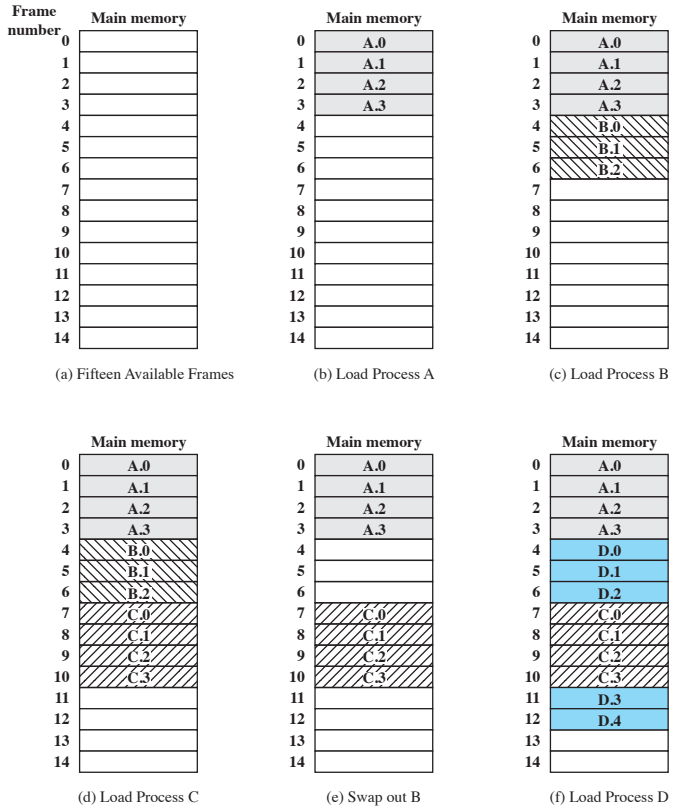
Memory Management Terms

- **Frame**: A fixed-length block of main memory.
- **Page**: A fixed-length block of data that resides in secondary memory. A page of data may temporarily be copied into a frame of main memory.
- **Segment**: A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging.)

Terms Relationships

- Frames correspond to pages in terms of size but are located in the physical RAM.
- A segment is a variable-length block of memory that is logically divided, often corresponding to program modules like functions, objects, or data arrays.
- Segmentation and paging can be combined in a segmented paging system, where the memory is first divided into segments, and then each segment is further divided into pages.

Suppose, however, that main memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of the same size. Then **the chunks of a process, known as pages**, could be assigned to **available chunks of memory, known as frames**. The wasted space in memory for each process is due to internal fragmentation consisting of only **a fraction of the last page of a process**. There is **no external fragmentation**.



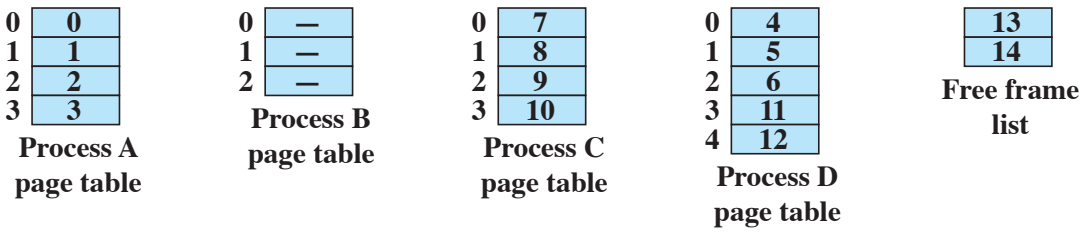
At a given point in time, some of the frames in memory are in use and some are free.

- A list of free frames is maintained by the OS.
- OS finds 4 free frames and loads the 4 pages of process A.
- Process B, consisting of three pages, and process C, consisting of four pages, are subsequently loaded.
- Then process B is suspended and is swapped out of main memory.
- The OS brings in a new process, process D, which consists of five pages.

The primary purpose of **logical addresses** is to provide an abstraction layer that allows each process to believe it is running in its own contiguous block of memory, starting from a base address, typically zero, regardless of the actual physical memory location where it is loaded.

8.2.1 Page Table

- Maintained by operating system for each process
- Contains the frame location for each page in the process. Each logical address consists of a **page number** and an **offset within the page**.
- The processor must know how to access the page table for the current process, and use it to produce a physical address.



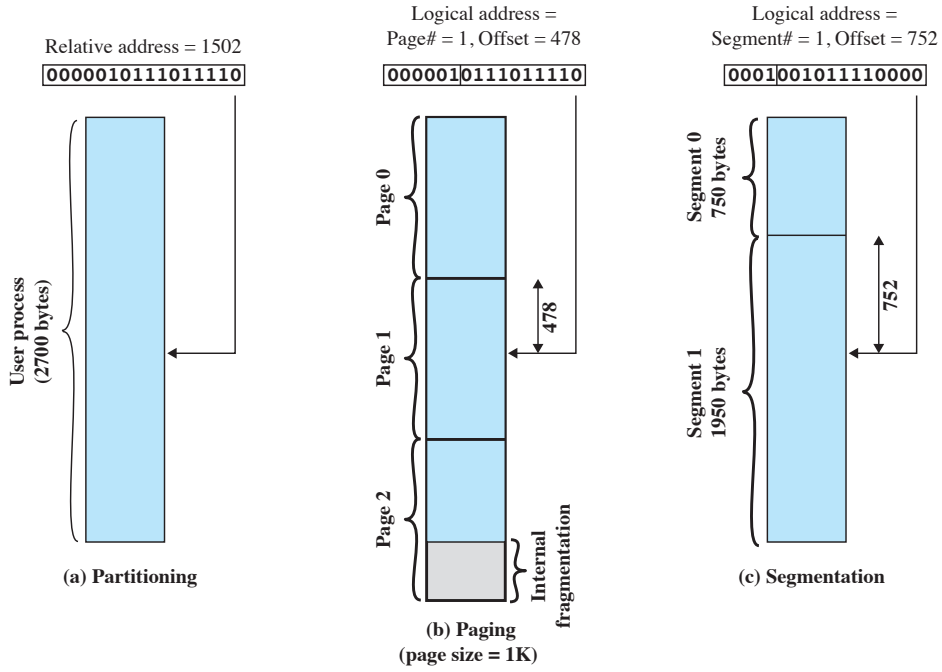
Continuing our example, the five pages of process D are loaded into frames 4, 5, 6, 11, and 12. A page table contains one entry for each page of the process, so that the table is easily indexed by the page number (starting at page 0). Each page table entry contains the number of the frame in main memory, if any, that holds the corresponding page. In addition, the OS maintains **a single free-frame list of all the frames** in main memory that are currently unoccupied and available for pages.

Thus we see that simple paging, as described here, is similar to fixed partitioning. The differences are that, **with paging, the partitions are rather small; a program may occupy more than one partition; and these partitions need not be contiguous.**

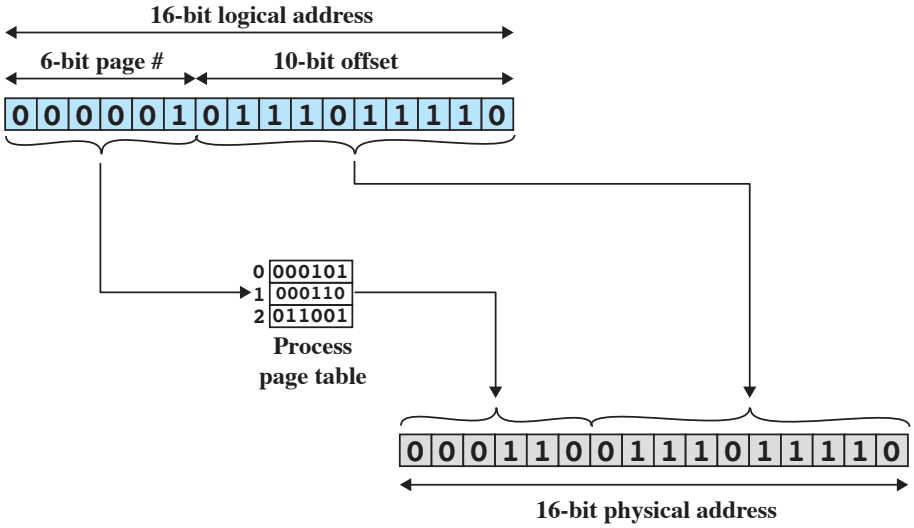
8.2.2 Logical Address

To make this paging scheme convenient, let us dictate that the page size, hence the frame size, **must be a power of 2**.

16-bit addresses are used: an offset field of 10 bits is needed, leaving 6 bits for the page number. Thus, 64 pages of 1K bytes each.



The relative address 1502, in binary form, is 0000010111011110. Relative address 1502 corresponds to an offset of 478 (0111011110) on page 1 (000001), which yields the same 16-bit number, 0000010111011110.



Consider an address of n + m bits, where the leftmost n bits are the page number and the rightmost m bits are the offset. In our example, n = 6 and m = 10. The following steps are needed for physical address translation:

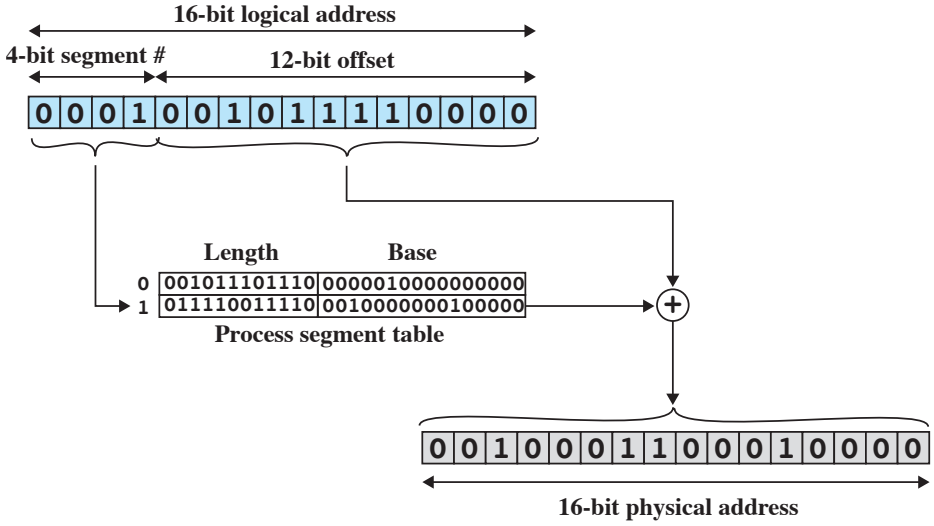
- Extract the page number as the leftmost n bits of the logical address.
- Use the page number as an index into the process page table to find the frame number, k.
- Appending the frame number to the offset.

8.3 Segmentation

The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous.

Eliminates internal fragmentation but suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses. Analogous to paging, a simple segmentation scheme would make use of a segment table for each process and a list of free blocks of main memory. Each segment table entry would have to give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware.



Consider an address of n + m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. In our example above, n = 4 and m = 12. Thus the maximum segment size is $2^{12} = 4096$. The following steps are needed for address translation:

1. Extract the segment number as the leftmost n bits of the logical address.
2. Use the segment number as an index into the process segment table to find the starting physical address of the segment.
3. Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
4. The desired physical address is the sum of the starting physical address of the segment plus the offset.

In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752. Suppose that this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is 0010000000100000 + 001011110000=0010001100010000.

To summarize, with simple segmentation, a process is divided into a number of segments that need not be of equal size. When a process is brought in, all of its segments are loaded into available regions of memory, and a segment table is set up.

9 Virtual Memory

9.1 Motivation and Definition

There are two implications of virtual memory, both leading to enhanced system utilization:

1. **More processes in memory.** Since only parts of each process need to be loaded, memory can hold more processes at once. This makes better use of the processor, as it's more likely at least one process will be ready to run.
2. **Programs can be larger than memory.** This removes a major programming limitation. Without virtual memory, programmers must carefully track available memory and create complex strategies for oversized programs. With virtual memory, the operating system handles this automatically. Programmers can work as if they have access to the full size of disk storage, while the system loads only the needed pieces into memory.

Since processes can only execute within main memory, this memory is designated as **real memory**. However, programmers and users perceive a potentially much larger memory space—one that extends to disk storage. **This extended memory is known as virtual memory**. Virtual memory enables **efficient multiprogramming and frees users from the restrictive limitations imposed by main memory constraints.**

9.2 Thrashing

Consider a large process P consisting of a long program plus a number of arrays of data; Over any short period of time:

- Execution may be confined to a small section of the program and data
- It would clearly be wasteful to load in dozens of pieces for that process when only a few pieces will be used before the program is suspended and swapped out

Only a few pieces of any given process are in memory:

- More processes can be maintained in memory
- Time is saved because unused pieces are not swapped in and out of memory

But what if the loaded pieces are inappropriate? In the steady state, practically all of main memory will be occupied with process pieces:

- If the program branches to an instruction or references a data item on a piece not in main memory, a fault is triggered
- When the operating system brings one piece in, it must throw another out
- If it throws out a piece just before it is used, then it will just have to retrieve that piece again almost immediately

Thrashing: The system spends most of its time swapping pieces rather than executing instructions.

How to select pieces of a program and further avoid thrashing?

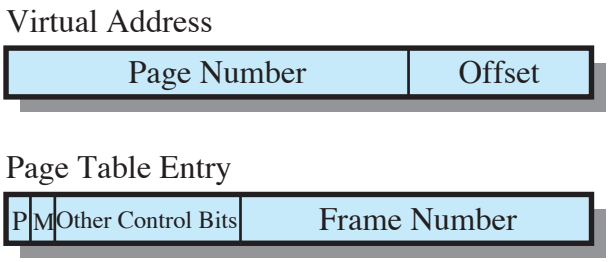
- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time

This reasoning is based on belief in the **principle of locality**. The principle of locality suggests that a virtual memory scheme may work. For virtual memory to be practical and effective, two ingredients are needed.

- There must be hardware support for the paging and/or segmentation scheme to be employed.
- The operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory.

9.3 Paging Only

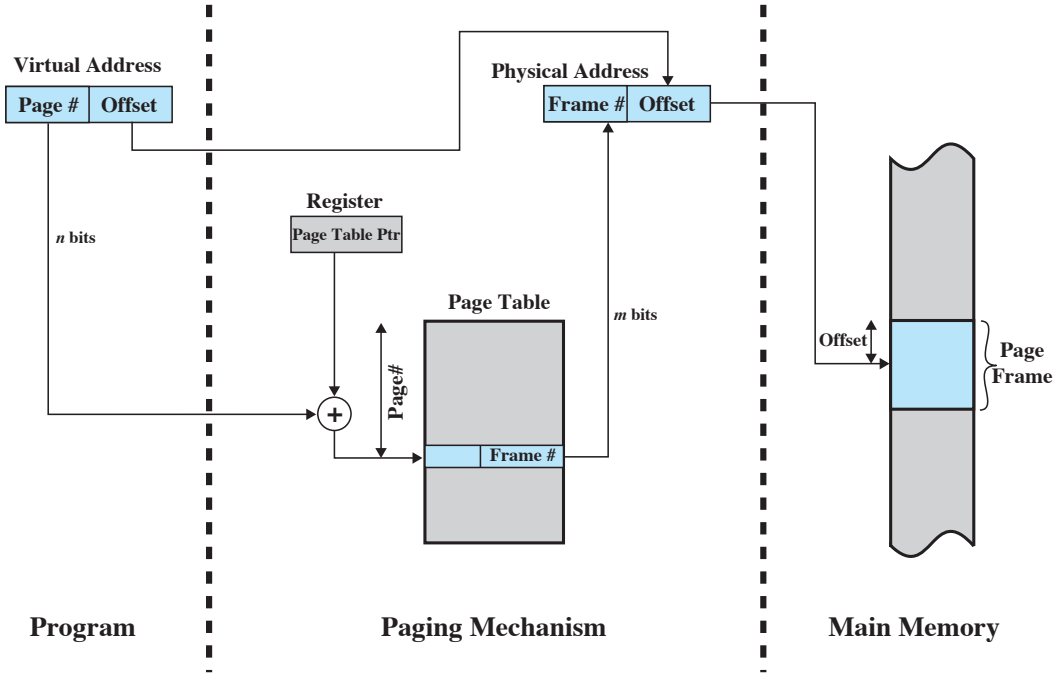
Typically, each process is associated with its own page table. Under this approach, page table entries become more complex.



(a) Paging only

Since only a subset of a process's pages may reside in main memory at any given time, each page table entry requires **a presence bit (P)** that indicates whether the corresponding page is currently in main memory. When this bit indicates the page is present, the entry also contains the frame number where that page is stored.

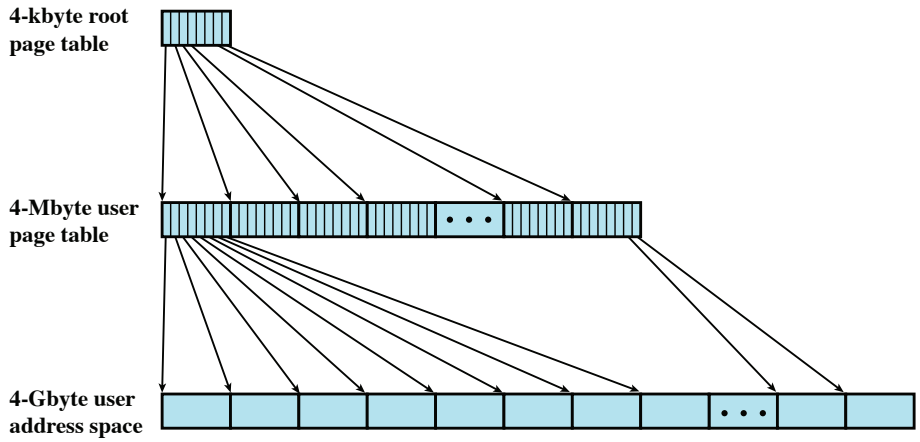
Additionally, each page table entry includes a **modify bit (M)** that tracks whether the page's contents have been changed since it was last loaded into main memory. If no modifications have occurred, the system can avoid writing the page back to secondary storage when the frame needs to be reallocated.



Since page tables vary in length based on process size, they cannot be stored in registers and must reside in main memory for access. During process execution, **a register stores the starting address of the current process's page table**. The virtual address's page number serves as an index into this table to retrieve the corresponding frame number. This frame number combines with the offset portion of the virtual address to generate the physical address.

Typically, the page number field exceeds the frame number field in bit length ($n > m$). This size difference occurs because **a process may contain more pages than the available frames in main memory**.

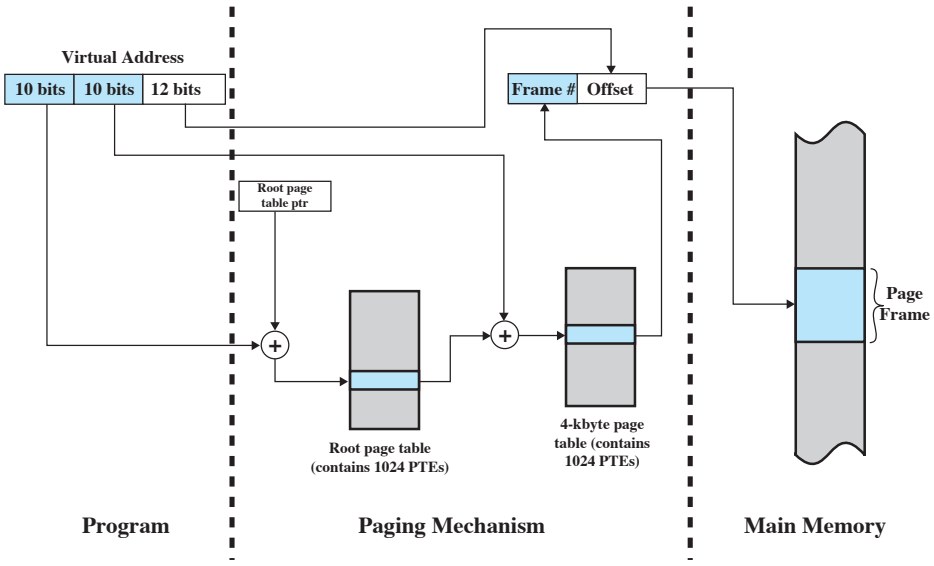
9.3.1 Two Level Hierarchical Page Table



For a 32-bit address system with hierarchical paging:

- Page size: 4-kilobyte (2^{12} bytes)
- Total number of pages: 2^{20} pages
- Each Page Table Entry (PTE) requires 4 bytes
- Level-1 page table size: 4 bytes $\times 2^{20}$ entries $= 2^{22}$ bytes (4 MB)
- Level-1 table occupies: $\frac{2^{22}}{2^{12}} = 2^{10}$ pages
- Level-2 (root) page table contains 2^{10} entries of 4 bytes each
- Level-2 table size: 4 bytes $\times 2^{10} = 2^{12}$ bytes (4 KB)
- The root page table occupies exactly one page in main memory

Thus, the two-level paging scheme maps the entire address space while keeping only the necessary page tables in memory.



The root page always remains in main memory. The first 10 bits of a virtual address are used to index into the root page to find a PTE for a page of the user page table. If that page is not in main memory, a page fault occurs. If that page is in main memory, then the next 10 bits of the virtual address index into the user PTE page to find the PTE for the page that is referenced by the virtual address.

Given information below:

- Virtual Address: `0x04D2C678`
- Root Page Table Index: First 10 bits = `0x013` (19 in decimal)

- Second-Level Page Table Index: Next 10 bits = `0x2B0` (688 in decimal)
- Offset: Last 12 bits = `0xC678` (50808 in decimal)

1. The first 10 bits (19) index into the root page table.
2. The root page table entry at index 19 points to the second-level page table at frame #50.
3. The next 10 bits (688) index into the second-level page table at frame #50.
4. The second-level page table entry indicates that the desired page is stored in frame #1024.
5. The physical address is calculated by combining frame #1024 with the offset.

The physical address is computed by:

$$\begin{aligned} \text{Physical Address} &= \text{Frame Number} \times \text{Page Size} + \text{Offset} && (1)\\ &= 1024 \times 2^{12} + 50808 && (2)\\ &= 1024 \times 4096 + 50808 && (3)\\ &= 4194304 + 50808 && (4)\\ &= 4245112 && (5) \end{aligned}$$

Advantage:

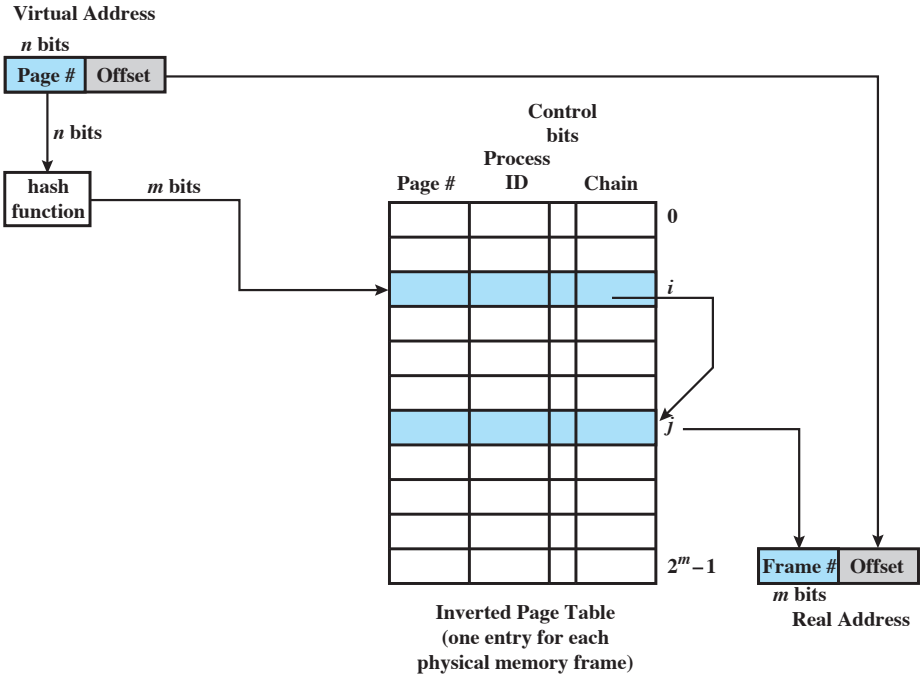
Single-level problem: A single-level page table for a 32-bit address space would require millions of entries (2^20 entries for 4KB pages), regardless of how much memory the process actually uses. This creates a massive memory overhead.

Two-level solution: Two-level page tables only allocate second-level tables for address ranges actually used by the process. For example, if a program only uses a few scattered megabytes across a 4GB address space, only a few second-level page tables need to be created rather than maintaining entries for the entire address space.

9.3.2 Inverted Page Table

A significant limitation of conventional page tables is that their size scales proportionally with the virtual address space. The inverted page table architecture offers an alternative approach:

- The page number component of a virtual address is transformed into a hash value
- This hash value references an entry in the inverted page table
- The inverted page table contains one entry per physical memory page frame, rather than one per virtual page
- This structure requires a fixed proportion of physical memory for tables, independent of the number of processes or virtual pages supported
- It is termed "inverted" because page table entries are indexed by frame number instead of virtual page number
- This architecture proves particularly advantageous for shared memory implementations



Each entry in the inverted page table includes the following elements:

- **Page number:** The page number component from the virtual address.
- **Process identifier:** Identifier of the process that owns this page.
- **Control bits:** This field contains various flags such as valid, referenced, and modified; along with protection and locking information.
- **Chain pointer:** Contains null if there are no chained entries for this entry. Otherwise, it stores the index value of the next entry in the chain.

Figure illustrates a typical implementation of the inverted page table approach. For a physical memory with 2^m frames, the inverted page table contains 2^m entries, where the i-th entry corresponds to frame i.

In this example, the virtual address includes an n-bit page number, where $n > m$. The hash function transforms the n-bit page number into an m-bit quantity, which serves as an index into the inverted page table.

Example:

- Process A requests access to virtual page 1
- Process B requests access to virtual page 5

When Process A accesses virtual page 1:

1. MMU computes $h(1) = 3$
2. Finding entry 3 empty, MMU creates new mapping:

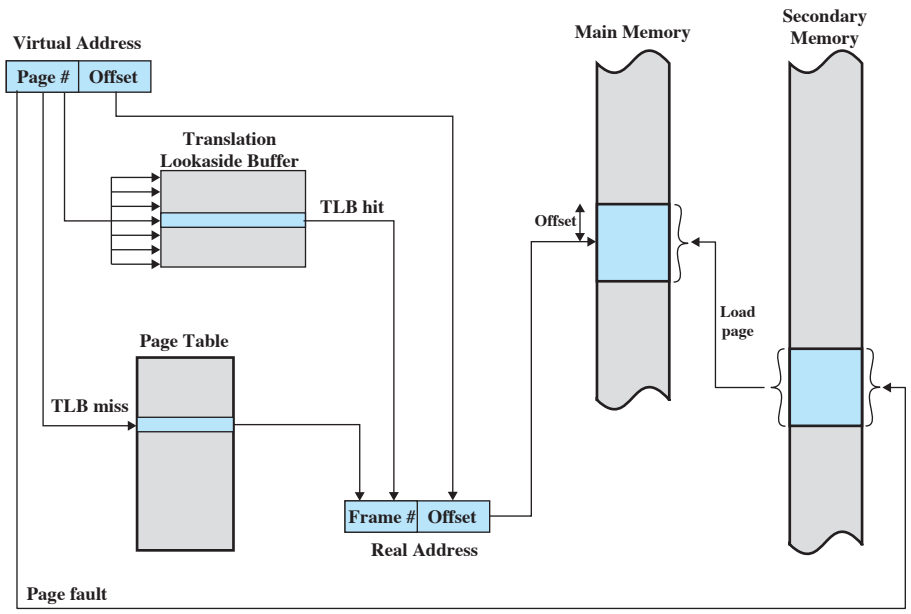
$$\text{Entry } 3 : \langle \text{VPN} = 1, \text{Frame} = X, \text{PID}_A \rangle \rightarrow \text{NULL}$$

When Process B accesses virtual page 5:

1. MMU computes $h(5) = 3$
2. Entry 3 is occupied (collision detected)
3. MMU adds new node to linked list:

$$\text{Entry } 3 : \langle \text{VPN} = 1, \text{Frame} = X, \text{PID}_A \rangle \rightarrow \langle \text{VPN} = 5, \text{Frame} = Y, \text{PID}_B \rangle \rightarrow \text{NULL}$$

9.3.3 Translation Lookaside Buffer(TLB)



In principle, every virtual memory reference can cause two physical memory accesses:

- One to fetch the appropriate page table entry
- One to fetch the desired data

Thus, **a straightforward virtual memory scheme would have the effect of doubling the memory access time.** To overcome this problem, most virtual memory implementations make use of **a special high-speed cache** for page table entries, commonly known as a **translation lookaside buffer (TLB)**. This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used.

The TLB significantly improves performance by:

1. Storing recently accessed page table entries
2. Reducing the need for accessing the full page table in memory
3. Minimizing the performance penalty of virtual-to-physical address translation

Without a TLB, each memory access would require:

$$T_{\text{access}} = T_{\text{page table lookup}} + T_{\text{physical memory access}} \tag{6}$$

With an effective TLB implementation, the common case becomes:

$$T_{\text{access with TLB hit}} \approx T_{\text{physical memory access}} + T_{\text{TLB lookup}} \tag{7}$$

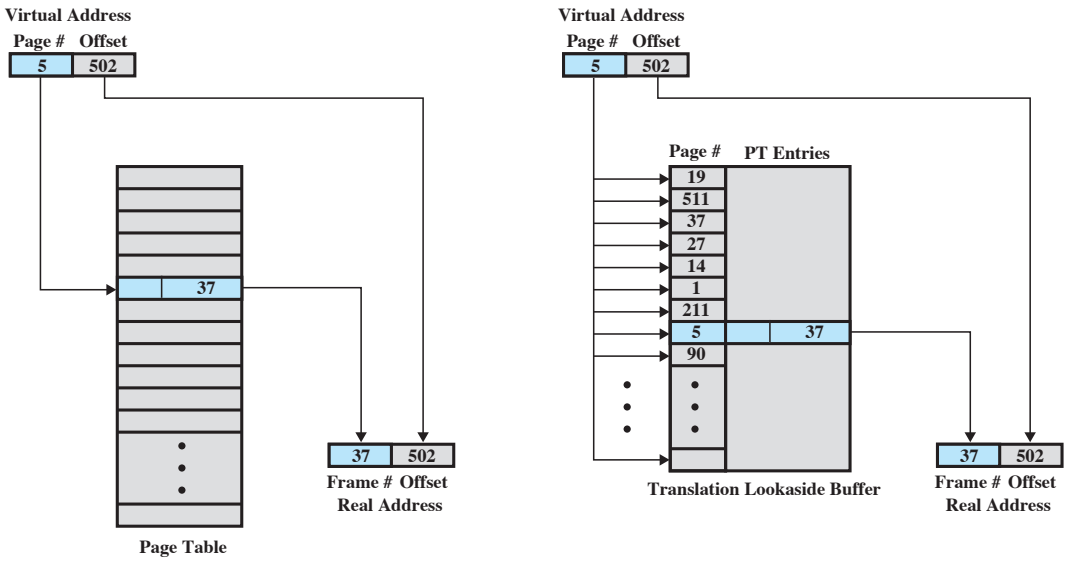
where $T_{\text{TLB lookup}} \ll T_{\text{page table lookup}}$

Unlike page tables which can be directly indexed by page number, the Translation Lookaside Buffer (TLB) requires a different approach:

- The TLB maintains only a subset of page table entries, making direct indexing impossible
- Each TLB entry must contain both the page number and the complete page table entry information
- TLB lookup employs special hardware using content-addressable memory (CAM)

TLB lookup utilizes **associative mapping** rather than direct mapping:

- The processor hardware simultaneously searches multiple TLB entries
- Each entry is checked for a matching page number
- This parallel search approach is called associative mapping
- This contrasts with direct mapping (indexing) used for page table lookups



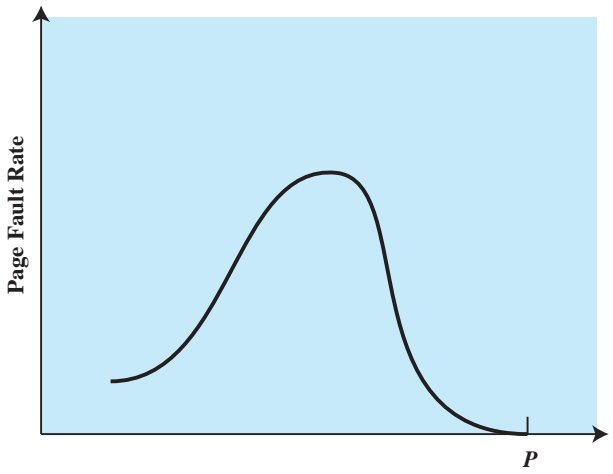
(a) Direct mapping

(b) Associative mapping

9.4 Page Size

As page size decreases, internal fragmentation is reduced. However, this necessitates more pages per process, resulting in larger page tables. In heavily multiprogrammed environments with large programs, some portions of the page tables for active processes must reside in virtual memory rather than main memory. Consequently, a single memory reference

may trigger two page faults: **the first to retrieve the necessary page table segment, and the second to fetch the actual process page.**

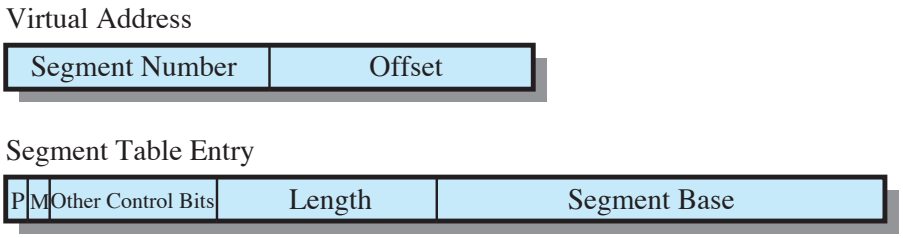


(a) Page Size

Page size significantly influences the frequency of page faults through mechanisms related to the principle of locality.

- If the page size is very small: relatively large number of pages will be available: locality.
- As the size of the page is increased: each individual page will contain locations further and further from any particular recent reference.
- Eventually: the page fault rate will begin to fall as the size of a page approaches the size of the entire process.

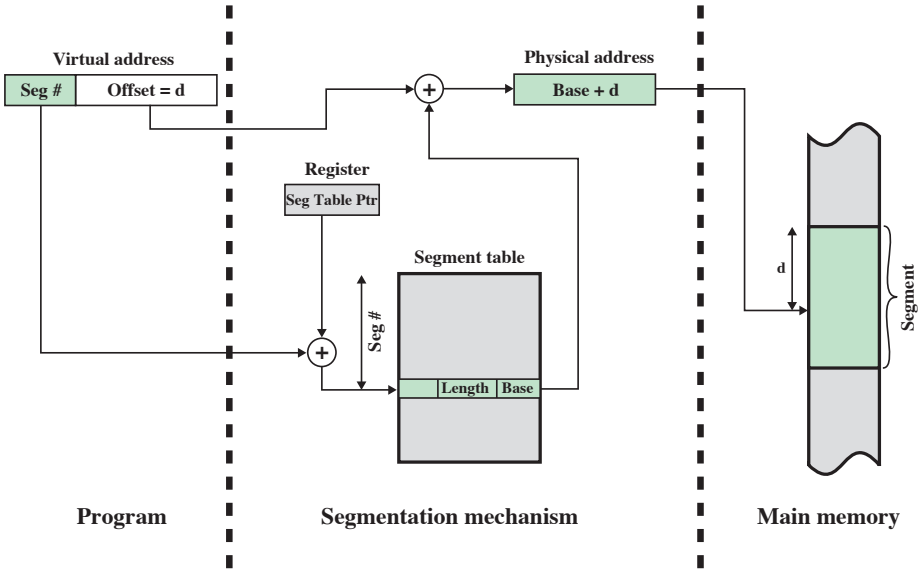
9.5 Segmentation Only



(b) Segmentation only

The components of each segment table entry:

- the starting address of the corresponding segment in main memory
- the length of the segment
- A bit is needed to determine if the segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory.



This virtual address consists of a segment number and an offset, and the conversion utilizes a segment table. Since the segment table's length varies according to process size, it cannot be stored in registers. Instead, it must reside in main memory for access.

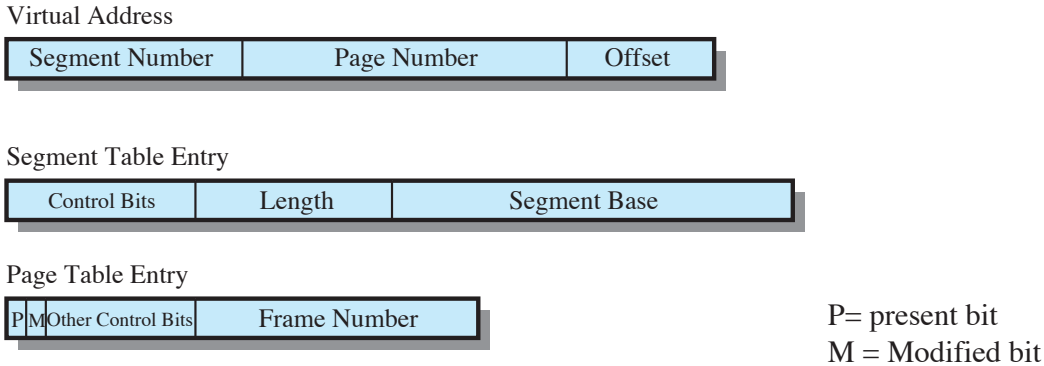
During process execution, a register stores the starting address of that process's segment table. The segment number from a virtual address serves as an index into this table, retrieving the corresponding main memory address for the segment's beginning. This address is then combined with the offset portion of the virtual address to generate the desired physical address.

9.6 Combined Paging And Segmentation

Paging and segmentation each offer distinct advantages in memory management systems.

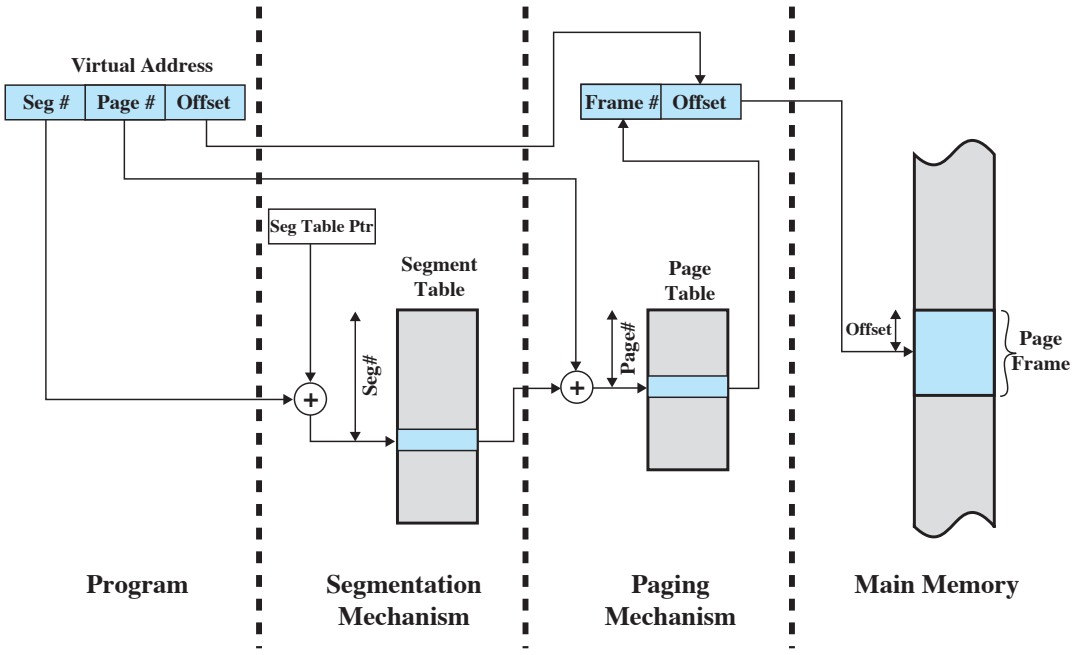
Paging, which operates transparently to the programmer, eliminates external fragmentation, thereby ensuring efficient main memory utilization. Furthermore, since the memory units transferred between main memory and secondary storage are of uniform, fixed size, this facilitates the development of sophisticated memory management algorithms that can exploit program behavior patterns.

Segmentation, which remains visible to the programmer, provides several benefits, including accommodation of dynamically growing data structures, support for modular programming, and mechanisms for sharing and protection.



(c) Combined segmentation and paging

In a combined paging/segmentation architecture, a user's address space is divided into multiple segments according to the programmer's requirements. Each segment is subsequently divided into fixed-size pages, with each page corresponding in size to a main memory frame. If a segment's length is less than a page, it occupies exactly one page. From the programmer's perspective, a logical address continues to consist of a segment number and a segment offset. From the system's perspective, however, the segment offset is interpreted as a combination of a page number and a page offset within the specified segment.



The segment number portion to index into the process segment table to find the page table for that segment.

The page number portion of the virtual address is used to index the page table and look up the corresponding frame number.

This is combined with the offset portion of the virtual address to produce the desired real address.

9.7 Fetch Policy

The fetch policy determines when a page should be brought into main memory. The two common alternatives are **demand paging** and **prepaging**.

9.7.1 Demand Paging

- Only brings pages into main memory when a reference is made to a location on the page
- Many page faults when process is first started
- Principle of locality suggests that as more and more pages are brought in, most future references will be to pages that have recently been brought in, and page faults should drop to a very low level

9.7.2 Prepaging

- Pages other than the one demanded by a page fault are brought into memory simultaneously
- Exploits the inherent characteristics of most secondary memory devices,such as disks, which have seek times and rotational latency.
- When pages of a process are stored contiguously in secondary memory, it becomes more efficient to bring in multiple pages at once
- The prepaging policy may be implemented in two scenarios:
  - At process initialization, where the programmer must designate desired pages
  - During every page fault occurrence (seem preferable because it is invisible to the programmer.)
- Becomes ineffective if the additional loaded pages remain unreferenced

Prepaging should not be confused with swapping. When a process is swapped out of memory and put in a suspended state, all of its resident pages are moved out. When the process is resumed, all of the pages that were previously in main memory are returned to main memory.

9.8 Replacement Policy

In memory management, page replacement addresses which page to remove when memory is full and a new page is needed.

- Concerns the selection of a frame in main memory for replacement when a new page must be loaded
- Goal is to remove the page least likely to be accessed soon
- More complex replacement policies require greater hardware and software implementation overhead

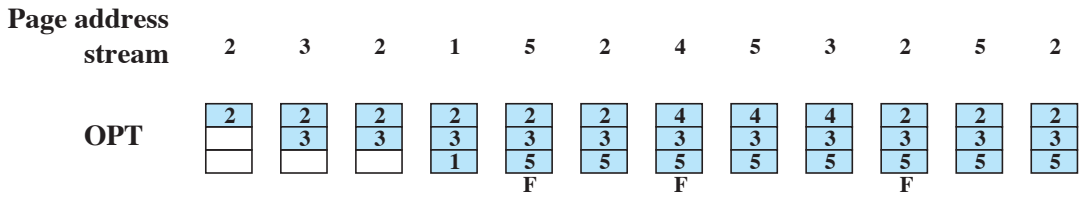
One restriction on replacement policy needs to be mentioned before looking at various algorithms: Some of the frames in main memory may be **locked**. When a frame is locked, the page currently stored in that frame may not be replaced. Much of the kernel of the OS, as well as key control structures, are held in locked frames. In addition, I/O buffers and other time-critical areas may be locked into main memory frames. Locking is achieved by associating a lock bit with each frame. This bit may be kept in a frame table as well as being included in the current page table.

There are certain basic algorithms that are used for the selection of a page to replace. Replacement algorithms that will be discussed:

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock

9.8.1 Optimal Policy

The optimal policy selects for replacement that page for which the time to the next reference is the longest. Clearly, this policy is **impossible to implement**, because it would require the operating system to have perfect knowledge of future events. However, it does **serve as a standard** against which to judge real-world algorithms.



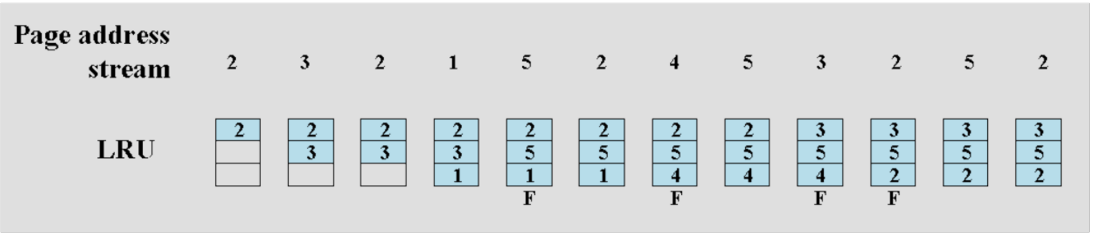
The example assumes a fixed frame allocation (fixed resident set size) for this process of three frames. The execution of the process requires reference to five distinct pages. The page address stream formed by executing the program is:

2 3 2 1 5 2 4 5 3 2 5 2

which means that the first page referenced is 2, the second page referenced is 3, and so on. The optimal policy produces three page faults after the frame allocation has been filled.

9.8.2 Least recently used (LRU)

The least recently used (LRU) policy **replaces the page in memory that has not been referenced for the longest time**. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, **the LRU policy does nearly as well as the optimal policy**. The problem with this approach is the difficulty in implementation. One approach would be to tag each page with the time of its last reference; this would have to be done at each memory reference, both instruction and data. Even if the hardware would support such a scheme, **the overhead would be tremendous**. Alternatively, one could maintain a stack of page references, again an expensive prospect.



```c
#include <stdio.h>

#define PAGE_SEQ_LEN 12
#define NUMBER_OF_FRAMES 3

// Structure for representing a page in memory
typedef struct {
    int number; // Page number
    int lastUsedTime; // The last time this page was accessed
} PageFrame;

void initializePageFrames(PageFrame frames[]) {
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        frames[i].number = -1; // -1 indicates that the frame is initially empty
        frames[i].lastUsedTime = -1; // Initialize last used time as -1
    }
}

void printFrames(PageFrame frames[]) {
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        if (frames[i].number != -1) {
            printf("%d(%d) ", frames[i].number, frames[i].lastUsedTime);
        } else {
            printf("- ");
        }
    }
    printf("\n");
}

// Function to find the least recently used page
int findLRUPageFrameIndex(PageFrame frames[]) {
    int lruIndex = 0;
    int earliestTime = frames[0].lastUsedTime;

    for (int i = 1; i < NUMBER_OF_FRAMES; i++) {
        if (frames[i].lastUsedTime < earliestTime) {
            lruIndex = i;
            earliestTime = frames[i].lastUsedTime;
        }
    }

    return lruIndex;
}

void LRUPageReplacement(int pages[]) {
    PageFrame frames[NUMBER_OF_FRAMES];
    initializePageFrames(frames);

    int pageFaults = 0;
    int time = 0; // To keep track of the time each page was last used

    for (int i = 0; i < PAGE_SEQ_LEN; i++) {
        int currentPage = pages[i];
        int found = 0;

        // Check if the page is already in one of the frames
        for (int j = 0; j < NUMBER_OF_FRAMES; j++) {
            if (frames[j].number == currentPage) {
                frames[j].lastUsedTime = time++; // Update the last used time
                found = 1;
                break;
            }
        }

        // If the page was not found in the frames
```

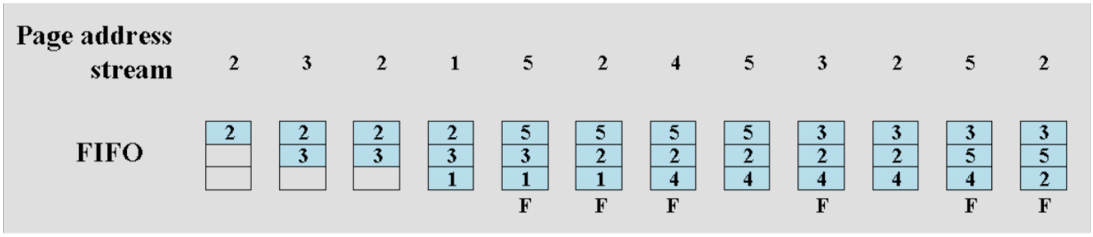
```
        if (!found) {
            int lruIndex = findLRUPageFrameIndex(frames);
            frames[lruIndex].number = currentPage; // Replace the LRU page with the new page
            frames[lruIndex].lastUsedTime = time++; // Update the last used time
            pageFaults++;
        }
        printf("Processing page %d: ", currentPage);
        printFrames(frames);
    }

    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int pages[PAGE_SEQ_LEN] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};

    printf("Starting LRU Page Replacement Simulation\n");
    LRUPageReplacement(pages);

    return 0;
}
```

9.8.3 First-in-first-out (FIFO)

- Page frames are treated as a circular buffer
- Replacement occurs in round-robin order
- Implementation requires only a pointer cycling through page frames
- The page residing in memory for the longest time is replaced

Limitation: FIFO often makes suboptimal decisions

- Frequently used program regions or data may be repeatedly paged in and out
- The algorithm ignores actual usage patterns




```
#include <stdio.h>
#include <stdbool.h>

#define PAGE_SEQ_LEN 12
#define NUMBER_OF_FRAMES 3

// Function to check if a page is already in a frame
bool isPageInFrames(int page, int frames[]) {
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        if (frames[i] == page) {
            return true;
        }
    }
    return false;
}

void printFramesFifo(int frames[]) {
    // Print the current state of frames
```

```c
    for (int j = 0; j < NUMBER_OF_FRAMES; j++) {
        if (frames[j] != -1) {
            printf("%d ", frames[j]);
        } else {
            printf("_ ");
        }
    }
    printf("\n");
}

// Function to implement FIFO page replacement
int fifo(int pages[], int frames[]) {
    int pageFaults = 0;
    int pageInsertIndex = 0;

    // Initialize frames to -1, indicating they're initially empty
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        frames[i] = -1;
    }

    for (int i = 0; i < PAGE_SEQ_LEN; i++) {
        int currentPage = pages[i];

        // If the current page is not in frames, we have a page fault
        if (!isPageInFrames(currentPage, frames)) {
            // Replace the oldest page with the current page
            frames[pageInsertIndex] = currentPage;

            // Move to the next frame index, wrapping around if necessary
            pageInsertIndex = (pageInsertIndex + 1) % NUMBER_OF_FRAMES;

            // Increase the count of page faults
            pageFaults++;
        }
        printf("Processing page %d: ", currentPage);
        printFramesFifo(frames);
    }

    return pageFaults;
}

int main() {
    int pages[PAGE_SEQ_LEN] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};
    int frames[NUMBER_OF_FRAMES];

    int pageFaults = fifo(pages, frames);

    printf("Total Page Faults: %d\n", pageFaults);
    return 0;
}
```

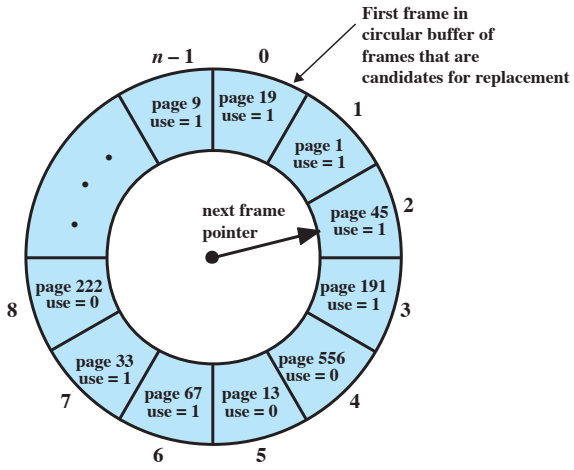
9.8.4 Clock Policy

- Associate an additional bit (use bit) with each frame
- Organize frames as a circular buffer with a pointer
- When a page is loaded or referenced, set its use bit to 1
- For replacement:
  - Examine frame at current pointer position
  - If use bit = 1, reset to 0 and advance pointer
  - If use bit = 0, replace this page and advance pointer

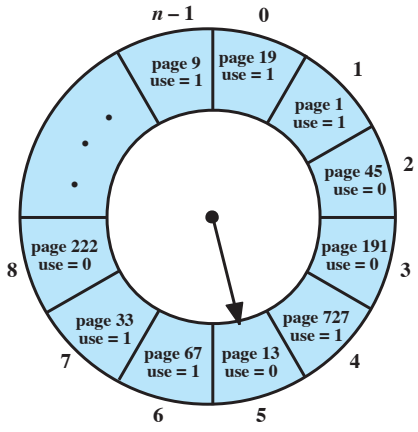
– If all frames have use bit = 1, algorithm completes one cycle

– After full cycle, all bits are reset to 0

– Replace page at original pointer position

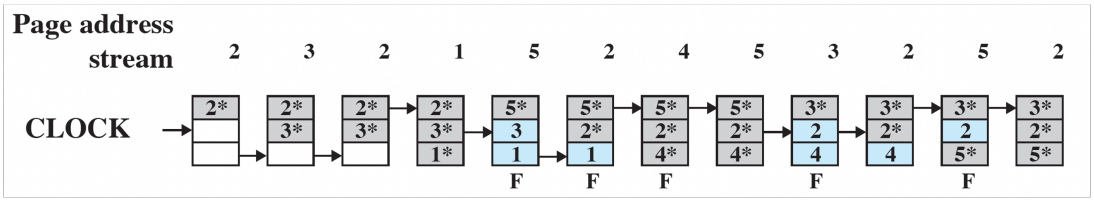


(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Above is an example of the simple clock policy mechanism. A circular buffer of n main memory frames is available for page replacement. Just prior to the replacement of a page from the buffer with incoming page 727 , the next frame pointer points at frame 2, which contains page 45 . The clock policy is now executed. Because the use bit for page 45 in frame 2 is equal to 1, this page is not replaced. Instead, the use bit is set to 0 and the pointer advances. Similarly, page 191 in frame 3 is not replaced; its use bit is set to 0 and the pointer advances. In the next frame, frame 4, the use bit is set to 0. Therefore, page 556 is replaced with page 727 . The use bit is set to 1 for this frame and the pointer advances to frame 5, completing the page replacement procedure.



The presence of an asterisk indicates that the corresponding use bit is equal to 1, and the arrow indicates the current position of the pointer. Note the clock policy is adept at protecting frames 2 and 5 from replacement.

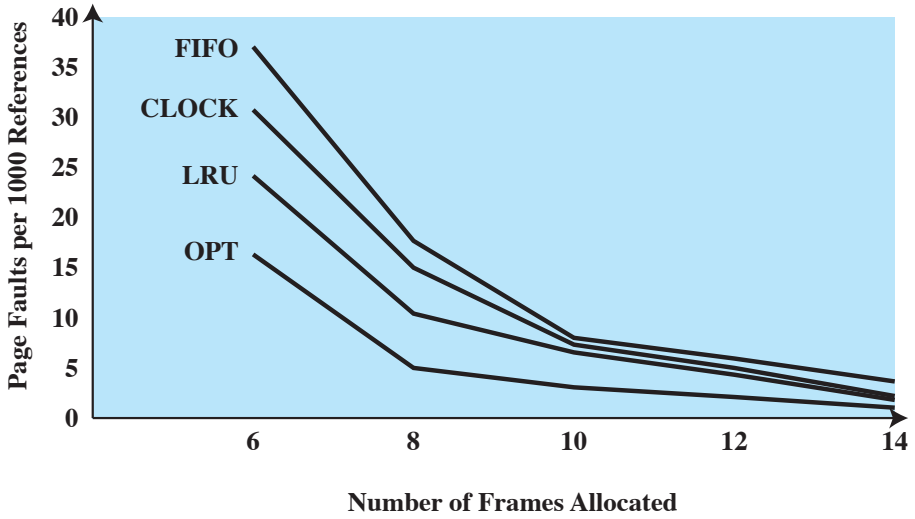
```c
#include <stdio.h>

#define PAGE_SEQ_LEN 12
#define NUMBER_OF_FRAMES 3

typedef struct {
```

```
    int pageNumber;
    int useBit;
} Frame;

void initializeFrames(Frame frames[]) {
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        frames[i].pageNumber = -1; // -1 indicates that the frame is empty
        frames[i].useBit = 0; // Initialize use bit to 0
    }
}

int findFrame(Frame frames[], int pageNumber) {
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        if (frames[i].pageNumber == pageNumber) {
            return i; // Return index of the frame if page is found
        }
    }
    return -1; // Return -1 if the page is not found in any frame
}

void displayFrames(Frame frames[]) {
    printf("Frames: ");
    for (int i = 0; i < NUMBER_OF_FRAMES; i++) {
        if (frames[i].pageNumber != -1) {
            printf("%d(%d) ", frames[i].pageNumber, frames[i].useBit);
        } else {
            printf("Empty ");
        }
    }
    printf("\n");
}

void clockPageReplacement(int pages[], Frame frames[]) {
    initializeFrames(frames);

    int pointer = 0; // Points to the oldest frame
    int pageFaults = 0;

    for (int i = 0; i < PAGE_SEQ_LEN; i++) {
        int page = pages[i];
        printf("Processing page: %d\n", page);
        int frameIndex = findFrame(frames, page);

        if (frameIndex == -1) { // Page fault
            while (frames[pointer].useBit == 1) {
                frames[pointer].useBit = 0; // Clear the use bit
                pointer = (pointer + 1) % NUMBER_OF_FRAMES; // Move the pointer to the next frame
            }
            // Replace the page at pointer with the new page
            frames[pointer].pageNumber = page;
            frames[pointer].useBit = 1; // Set the use bit
            pointer = (pointer + 1) % NUMBER_OF_FRAMES; // Move the pointer to the next frame

            pageFaults++;
        } else {
            frames[frameIndex].useBit = 1; // Set the use bit since the page was accessed
        }

        displayFrames(frames);
    }

    printf("\nTotal page faults: %d\n", pageFaults);
}

int main() {
    int pages[PAGE_SEQ_LEN] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};
```

```
    Frame frames[NUMBER_OF_FRAMES];

    printf("Clock Page Replacement Algorithm Simulation\n\n");
    clockPageReplacement(pages, frames);

    return 0;
}
```

9.8.5 Comparison

The experimental results shown in the figure compare four page replacement algorithms.



At smaller frame allocations, the differences between algorithms are most pronounced, with FIFO performing over twice as poorly as the optimal algorithm. Additional studies have shown that the clock algorithm closely approximates LRU performance when using variable allocation with either global or local replacement scope.

9.8.6 Page Buffering

While LRU and clock policies outperform FIFO, they add complexity and overhead. Additionally, replacing modified pages costs more than unmodified ones due to write-back requirements.

Page buffering improves performance while allowing simpler replacement policies:

- Basic FIFO replacement algorithm
- Two lists for replaced pages:
  - Free page list (unmodified pages)
  - Modified page list (modified pages)

When a page is replaced, it remains physically in memory, but its page table entry moves to the appropriate list. VMS maintains a small reserve of free frames. When reading a new page, the frame at the head of the free list is used. Replaced pages go to the tail of their respective lists.

The key advantage is that replaced pages remain in memory, enabling quick recovery if referenced again. Effectively, these lists serve as page caches. The modified page list provides an additional benefit by enabling cluster writing of modified pages, significantly reducing I/O operations and disk access time.

9.9 Resident Set Management

The OS must decide how many pages to bring into main memory.

- The smaller the amount of memory allocated to each process, the more processes can reside in memory
- Small number of pages loaded increases page faults
- Beyond a certain size, further allocations of pages will not effect the page fault rate

A fixed-allocation policy gives a process a fixed number of frames in main memory within which to execute. That number is decided at initial load time (process creation time) and may be determined based on the type of process (interactive, batch, type of application) or may be based on guidance from the programmer or system manager. With a fixed-allocation policy, whenever a page fault occurs in the execution of a process, one of the pages of that process must be replaced by the needed page.

A variable-allocation policy allows the number of page frames allocated to a process to be varied over the lifetime of the process. Ideally, a process that is suffering persistently high levels of page faults, indicating that the principle of locality only holds in a weak form for that process, will be given additional page frames to reduce the page fault rate; whereas a process with an exceptionally low page fault rate, indicating that the process is quite well behaved from a locality point of view, will be given a reduced allocation, with the hope that this will not noticeably increase the page fault rate.

The variable-allocation policy would appear to be the more powerful one. However, the difficulty with this approach is that **it requires the operating system to assess the behavior of active processes**. This inevitably requires software overhead in the operating system and is dependent on hardware mechanisms provided by the processor platform.

The scope of a replacement strategy can be categorized as **global or local**.

- **A local replacement policy** chooses only among the resident pages of the process that generated the page fault in selecting a page to replace.
- **A global replacement policy** considers all unlocked pages in main memory as candidates for replacement, regardless of which process owns a particular page.

Let's dicuss some potential combined policies:

- **With a fixed-allocation local scope policy**, it is necessary to decide ahead of time the amount of allocation to give to a process. This could be decided on the basis of the type of application and the amount requested by the program. The drawback to this approach is twofold:
  - If allocations tend to be too small, then there will be a high page fault rate, causing the entire multiprogramming system to run slowly.
  - If allocations tend to be unnecessarily large, then there will be too few programs in main memory and there will be either considerable processor idle time or considerable time spent in swapping.
- **With a variable-allocation global scope policy**, it is easiest to implement, the operating system maintains a list of free frames. When a page fault occurs, a free frame is added to the resident set of a process and the page is brought in. Thus, a process experiencing page faults will gradually grow in size, which should help reduce overall page faults in the system.

  The difficulty with this approach is in the replacement choice. When there are no free frames available, the operating system must choose a page currently in memory to replace. The selection is made from among all of the frames in memory, except for locked frames such as those of the kernel. Using any of the policies discussed in the preceding subsection, the page selected for replacement can belong to any of the resident processes; there is no discipline to determine which process should lose a page from its resident set. Therefore, the process that suffers the reduction in resident set size may not be optimum.

  One way to counter the potential performance problems of a variable-allocation, global-scope policy is to use page buffering. In this way, the choice of which page to replace becomes less significant, because the page may be reclaimed if it is referenced before the next time that a block of pages are overwritten.
- **With a variable-allocation local scope policy**, The variable-allocation, local-scope strategy attempts to overcome the problems with a global-scope strategy. It can be summarized as follows:
  - When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set, based on application type, program request, or other criteria. Use either prepaging or demand paging to fill up the allocation.
  - When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault.
  - From time to time, reevaluate the allocation provided to the process, and increase or decrease it to improve overall performance.

9.9.1 Working Set

The working set with parameter for a process at virtual time t, which we designate as$W(t, \Delta)$, is the set of pages of that process that have been referenced in the last Δ virtual time units.

$W(t, \Delta) = \{p \mid p \text{ was referenced in time interval } [t - \Delta, t]\}$

Sequence of Page References	Window Size, Δ			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

9.9.2 Page Fault Frequency (PFF)

An algorithm that follows variable-allocation local scope policy and working set is the page fault frequency (PFF) algorithm. It requires a use bit to be associated with each page in memory. The bit is set to 1 when that page is accessed.

For a specific Time interval T, we record page fault number in a T, if it exceeds upper bound, increase the resident set, if it below lower bound, decrease the resident set, otherwise, we keep everything unchanged. Reset page fault number for each T.

There is a major flaw in the PFF approach, which is that it does not perform well during the transient periods when there is a shift to a new locality. During interlocality transitions, the rapid succession of page faults causes the resident set of a process to swell before the pages of the old locality are expelled; the sudden peaks of memory demand may produce unnecessary process deactivations and reactivations, with the corresponding undesirable switching and swapping overheads.

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_PAGES 200
#define MAX_FRAMES 10
#define UPPER_PFF_LIMIT 7
#define LOWER_PFF_LIMIT 4
#define ADJUSTMENT_INTERVAL 10  // Interval after which the frame count can be adjusted

// Function to check if a page is in the set
int isInMemory(int page, int frames[], int frameCount) {
    for (int i = 0; i < frameCount; i++) {
        if (frames[i] == page) {
            return 1; // Page is in memory
        }
    }
    return 0; // Page is not in memory
}

// PFF algorithm simulation
```

```c
void simulatePFF(int pages[]) {
    int frames[MAX_FRAMES] = {0};
    int frameCount = 3; // Starting frame count
    int pageFaults = 0;
    int referenceCounter = 0;
    int pointer = 0;

    for (int i = 0; i < MAX_PAGES; i++) {
        referenceCounter++;
        if (!isInMemory(pages[i], frames, frameCount)) {
            // Page fault occurred
            frames[pointer] = pages[i];
            pointer = (pointer + 1) % frameCount;
            pageFaults++;
            printf("Time: %d; Page fault for page %d\n", i, pages[i]);
        }
        else{
            printf("Time %d; Page %d hit!\n", i, pages[i]);
        }


        // Check if we need to adjust the frame count after every ADJUSTMENT_INTERVAL references
        if (referenceCounter % ADJUSTMENT_INTERVAL == 0) {
            if (pageFaults >= UPPER_PFF_LIMIT && frameCount < MAX_FRAMES) {
                frameCount++; // Increase the frame count
                printf("Increasing frame count to %d due to high page fault frequency.\n", frameCount);
            } else if (pageFaults <= LOWER_PFF_LIMIT && frameCount > 1) {
                frameCount--; // Decrease the frame count
                printf("Decreasing frame count to %d due to low page fault frequency.\n", frameCount);
            }
            // Reset page fault count for the next interval
            pageFaults = 0;
        }

    }
}

int main() {
    int pages[MAX_PAGES];
    // Generate a sequence of page references to trigger both increase and decrease in frame count
    for (int i = 0; i < MAX_PAGES; i++) {
        if (i % 40 < 20) {
            // Alternate between a narrow range and a wider range of pages
            pages[i] = rand() % 5;
        } else {
            pages[i] = rand() % 10 + 5;
        }
    }

    printf("Starting PFF algorithm simulation...\n");
    simulatePFF(pages);

    return 0;
}
```

9.9.3 Variable Interval Sampled Working Set (VSWS)

At the beginning of a sampling interval, the use bits of all the resident pages for the process are reset; at the end, only the pages that have been referenced during the interval will have their use bit set; these pages are retained in the resident set of the process throughout the next interval, while the others are discarded. Thus the resident set size can only decrease at the end of an interval. During each interval, any faulted pages are added to the resident set; thus the resident set remains fixed or grows during the interval.

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_PAGES 400
#define MAX_FRAMES 20

typedef struct {
    int pageNumber;
    int useBit;
} Frame;

void initializeFramesVSWS(Frame frames[], int size) {
    for (int i = 0; i < size; i++) {
        frames[i].pageNumber = -1; // Indicates an empty frame
        frames[i].useBit = 0;
    }
}

int findPage(int pageNumber, Frame frames[], int size) {
    for (int i = 0; i < size; i++) {
        if (frames[i].pageNumber == pageNumber) {
            return i; // Page found
        }
    }
    return -1; // Page not found
}

void simulateVSWS(int pages[], int R, int Q) {
    Frame frames[MAX_FRAMES];
    initializeFramesVSWS(frames, MAX_FRAMES);

    int frameCount = 0;
    int pageFaults = 0;
    int lastSampleTime = 0;
    int currentTime = 0;

    for (currentTime = 0; currentTime < MAX_PAGES; currentTime++) {
        int pageIndex = findPage(pages[currentTime], frames, frameCount);

        if (pageIndex == -1) { // Page fault
            pageFaults++;
            if (frameCount < MAX_FRAMES) {
                frames[frameCount].pageNumber = pages[currentTime];
                frames[frameCount].useBit = 1;
                frameCount++;
            }
        } else {
            frames[pageIndex].useBit = 1; // Mark the page as used
        }

        // Check the VSWS policy conditions
        if ((currentTime - lastSampleTime == R) || (pageFaults == Q)) {
            // Scan use bits and adjust the resident set
            for (int i = 0; i < frameCount; i++) {
                if (frames[i].useBit == 0) {
                    frames[i] = frames[--frameCount]; // Remove the page
                    i--; // Adjust loop index after removal
                } else {
                    frames[i].useBit = 0; // Reset the use bit for the next interval
                }
            }


            printf("Sample at time %d, resident set size: %d\n", currentTime, frameCount);
            lastSampleTime = currentTime;
```

```
            pageFaults = 0; // Reset page faults for the next interval
        }
    }
}

int main() {
    int pages[MAX_PAGES];
    // Generate a sequence of page references
    for (int i = 0; i < MAX_PAGES; i++) {
        pages[i] = rand() % 15; // Assume 15 different pages
    }

    int R = 20; // duration of the sampling interval
    int Q = 11;  // Allowed page faults between sampling instances

    printf("Starting VSWS algorithm simulation...\n");
    simulateVSWS(pages, R, Q);

    return 0;
}
```

9.10 Cleaning Policy

Concerned with determining when a modified page should be written out to secondary memory.

- **Demand Cleaning**: A page is written out to secondary memory only when it has been selected for replacement
- **Precleaning**: Allows the writing of pages in batches

Both precleaning and demand cleaning have drawbacks.

- Precleaning allows the writing of pages in batches, but it makes little sense to write out hundreds or thousands of pages only to find that the majority of them have been modified again before they are replaced. The transfer capacity of secondary memory is limited and should not be wasted with unnecessary cleaning operations.
- With demand cleaning, the writing of a dirty page is coupled to, and precedes, the reading in of a new page. This technique may minimize page writes, but it means that a process that suffers a page fault may have to wait for two page transfers before it can be unblocked. This may decrease processor utilization.

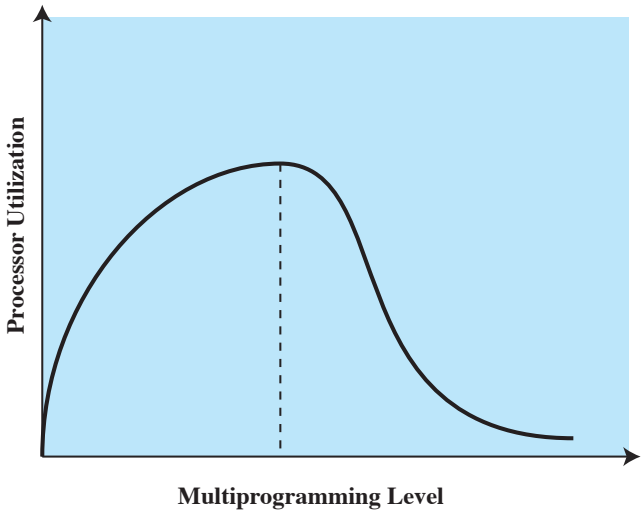
A better approach incorporates page buffering. With page buffering, replaced pages can be placed on two lists: modified and unmodified. The pages on the modified list can periodically be written out in batches and moved to the unmodified list. A page on the unmodified list is either reclaimed if it is referenced or lost when its frame is assigned to another page.

9.11 Load Control

Determines the number of processes that will be resident in main memory.

- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping
- Too many processes will lead to thrashing

As the multiprogramming level increases from a small value, one would expect to see processor utilization rise, because there is less chance that all resident processes are blocked. However, a point is reached at which the average resident set is inadequate. At this point, the number of page faults rises dramatically, and processor utilization collapses.



The Approaches:

- A working set or PFF algorithm implicitly incorporates load control. In providing the required resident set size for each active process, the policy automatically and dynamically determines the number of active programs.
- Adjusts the multiprogramming level so that the mean time between faults equals the mean time required to process a page fault. Performance studies indicate that this is the point at which processor utilization attained a maximum.

If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be swapped out, then 6 possibilities exist:

- Lowest-priority process
- Faulting process
- Last process activated
- Process with the smallest resident set
- Largest process
- Process with the largest remaining execution window

9.12 Memory Management in Old Linux Kernel

In the Intel 80X86 architecture system, the Linux kernel's memory management subsystem implements **paging mechanisms**. It utilizes **page directory** and **page table structures** to handle the allocation and deallocation of memory requested by other kernel components. Memory management operates in units called memory pages, where each page represents **4K bytes of contiguous physical memory addresses**. Page directory entries and page table entries enable addressing and management of specific memory pages. There are three files in the memory management directory of Linux 0.12:

- The `page.s` assembly file contains only the memory page exception interrupt service program (INT 14). It primarily implements page fault and page write protection handling.
- The `memory.c` file serves as the core program for memory page management. It performs memory initialization operations, manages page directories and page tables, and processes memory allocation requests from other kernel components.
- The `swap.c` program handles memory page swapping operations, including swap mapping bitmap management functions and swap device access functions.

9.12.1 `mem_init()`

Its primary purpose is to initialize the memory map that tracks which physical memory pages are in use and which are free for allocation.

```
1 void mem_init(long start_mem, long end_mem)
```

- `start_mem`: The starting address of the main memory area (MMA) that can be allocated.
- `end_mem`: The ending address of physical memory (typically 16MB in early Linux systems).

The `mem_init()` function works as follows:

1. First, it marks all memory pages in the range 1MB-16MB as used (`USED` = 100).
2. Then, it identifies which subset of these pages constitute the main memory area (MMA).
3. It marks just those main memory area pages as free (0) for future allocation.
4. Typically, this would make the range from around 4MB to 16MB available for dynamic allocation.

This approach reserves the lower portion of memory (below `start_mem`, typically in the range of 1MB to 4MB) for the kernel itself and other fixed system structures. In early Linux systems:

- 0-1MB: Reserved for system operations, BIOS, and other low-level functions.
- 1MB-`start_mem` (typically around 4MB): Used for kernel code, data structures, and statically allocated components.
- `start_mem` to `end_mem` (typically 4MB-16MB): The main memory area (MMA) available for dynamic allocation.

Here is the function implementation:

```
void mem_init(long start_mem, long end_mem)
{
    int i;
    /*
     * set the memory top (16MB).
     * The 16MB reference in this code represents a historical limitation from very early Linux kernels,
     * not the maximum addressable memory on the x86 architecture.
     */

    HIGH_MEMORY = end_mem;

    /*
     * The function first sets the memory mapped byte array items corresponding to all pages in
     * the range of 1MB to 16MB to the occupied state, that is, all bytes are set to USED (100).
     * PAGING_PAGES is defined as (PAGING_MEMORY>>12), which is the number of all physical memory
     * pages above 1MB (15MB/4KB = 3840).

     * 0-1MB region: The first megabyte of memory (addresses 0x00000000 to 0x000FFFFF) was special. It contained:
     * The interrupt vector table
     * BIOS data and code
     * Memory-mapped I/O for hardware devices
     * The real-mode operating environment

     * Above 1MB: Memory starting from address 0x00100000 (1MB) and extending upward
     */
    for (i=0 ; i<PAGING_PAGES ; i++) // PAGING_PAGES = 3840.
        mem_map[i] = USED;

    /*
     * MAP_NR() is a macro that converts a physical address to a page number,
     * typically by dividing by the page size (4KB) and potentially applying an offset.
     * Find the number 'i' in the byte array corresponding to the page at the start address 'start_mem'.
     */
    i = MAP_NR(start_mem);
    /*
     * calculates the size of the main memory area by subtracting the start address from the end address.
     */
    end_mem -= start_mem;

    /*
     * converts the memory size from bytes to the number of pages.
     * Right-shifting by 12 is equivalent to dividing by £2^{12} = 4096£ (4KB, the standard page size).
     */
    end_mem >>= 12;
```

```
    /*
     * Mark just those main memory area pages as free (0) for future allocation.
     */
    while (end_mem-->0)
        mem_map[i++]=0; // bytes of MMA pages are reset.
}
```

9.12.2 get_empty_page()

The get_empty_page() function serves a dual purpose in memory management:

1. It allocates a physical memory page from the main memory area
2. It maps this physical page to a specified virtual address in the memory space

```
void get_empty_page(unsigned long address)
```

address: An unsigned long that represents the virtual address where the newly allocated page should be mapped.

```
void get_empty_page(unsigned long address)
{
    /*
     * tmp is defined to store the physical page address that will be allocated.
     */
    unsigned long tmp;

    /*
     * tmp=get_free_page() calls a function to allocate a free physical page in main memory and assigns its address to tmp.
     * If get_free_page() returns 0, indicating failure to allocate a page
     * put_page(tmp,address) attempts to map this physical page to the specified virtual address.
     * If put_page() returns 0, indicating failure to map the page to the specified address.
    */
    if (!(tmp=get_free_page()) || !put_page(tmp,address)) {

        /*
         * Calls \texttt{free_page(tmp)} to release the allocated page (if any)
         */
        free_page(tmp);

        /*
         * Calls oom() which stands for "out of memory"
         * a function that handles the out-of-memory condition,
         * likely by displaying an error message and possibly terminating the process
        */
        oom();
    }
}
```

9.12.3 put_page()

This function maps a physical memory page to a specific virtual address in the kernel.

```
static unsigned long put_page(unsigned long page, unsigned long address)
```

- page: Physical address of a page frame in main memory
- address: Desired virtual address where the page should be mapped

```c
/*
 * This function puts a page in memory at the wanted address.
 * It returns the physical address of the page gotten, 0 if
 * out of memory (either when trying to access page-table or
 * page.)
 *
 * The main job of implementing this function is to set the information of the specified
 * page in the relevant page directory entry and page table entry.
 */
static unsigned long put_page(unsigned long page, unsigned long address)
{
    unsigned long tmp, *page_table;

    /*
     * Check if the physical page address is within valid memory range
     * and verify that the page has been properly allocated.
     */
    if (page < LOW_MEM || page >= HIGH_MEMORY)
        printk("Trying to put page %p at %p\n", page, address);

    // In early Linux kernels, a value of 1 in mem_map typically indicated a free page
    if (mem_map[(page-LOW_MEM)>>12] != 1)
        printk("mem_map disagrees with %p at %p\n", page, address);

    /*
     * Calculate the page directory entry pointer from the virtual address.
     * The top 10 bits of the virtual address form the directory index.
     * Shifting right by 20 bits and masking with 0xffc produces a 4-byte aligned pointer.
     * The mask 0xffc equals 0b1111 1111 1100 in binary.
     * This means it keeps the upper bits and zeros out the bottom 2 bits.
     * The purpose of this masking is to ensure 4-byte alignment of the pointer.
     * In a 32-bit system:
     * Each page directory entry is 4 bytes (32 bits) in size
     * The page directory itself is an array of these 4-byte entries
     * To access a specific entry, we need to multiply the index by 4
     * This creates a properly aligned pointer to the correct page directory entry
     */
    /*
     * address>>20 shifts the virtual address to the right by 20 bits. In x86 architecture with 4KB pages:
     * Bits 0-11 typically represent the offset within a page
     * Bits 12-21 typically index into the page table
     * Bits 22-31 typically index into the page directory
     * By shifting right 20 bits, the code is extracting the page directory index portion of the address,
     * but keeping some bits that will be used for alignment.
     * & 0xffc masks off all but bits 2-11 of the shifted value:

     * 0xffc is 1111 1111 1100 in binary
     * This preserves bits 2-11 and clears bits 0-1
     * This ensures the resulting value is a multiple of 4 (word-aligned)
     */
    page_table = (unsigned long *)((address>>20) & 0xffc);

    /*
     * Check if the page directory entry is valid (bit 0 = Present flag).
     * If valid, extract the page table address from the entry.
     * If not valid, allocate a new page for the page table and update the directory entry.
     */
    /*
     * checks if the least significant bit of the page table entry is set to 1.
     * In x86 architecture, bit 0 of a page table entry is the "Present" bit
     * When this bit is 1, it means the page table entry points to a valid page or another page table
     */
    if ((*page_table) & 1) {
        /* Page table exists - extract its address */
        page_table = (unsigned long *)(0xfffff000 & *page_table);
```

```
    } else {
        /* Page table doesn't exist - create a new one */
        if (!(tmp = get_free_page()))
            return 0;  /* Out of memory */

        /* Set directory entry with flags 7 (Present, Read/Write, User) */
        *page_table = tmp | 7;
        page_table = (unsigned long *)tmp;
    }

    /*
     * Set the page table entry for the specified linear address.
     * page_table is a pointer to the current page table being modified.
     * (address>>12) & 0x3ff calculates which entry in the page table to modify:

     * address>>12 shifts the virtual address right by 12 bits, removing the page offset portion (assuming 4KB pages)
     * & 0x3ff masks it to keep only the lower 10 bits (0x3ff = 1023 in decimal, or binary 11 1111 1111)
     * This gives an index between 0 and 1023, representing which of the 1024 entries in the page table to modify

     * page | 7 constructs the actual page table entry value:

     * page is the physical address of the page being mapped
     * | 7 sets the three least significant bits (111 in binary):

     * Bit 0: Present (1 = page is in physical memory)
     * Bit 1: Read/Write (1 = page is writable)
     * Bit 2: User/Supervisor (1 = page is accessible from user mode)
     * The assignment puts this constructed value into the calculated index of the page table.
     */
    return page;
}
```

9.12.4 do_no_page()

The `do_no_page()` function is a critical part of Linux's page fault handling mechanism. It's called when a process attempts to access memory that isn't present in physical RAM, triggering a page fault exception.

```
void do_no_page(unsigned long error_code, unsigned long address)
```

This function receives two parameters:

- error_code: CPU-generated error type
- address: virtual address that caused the page fault

The function follows a systematic approach to resolve page faults:

1. Validate the address range
2. Check if the page is in swap space
3. Determine the location of required data (executable, library, or dynamic memory)
4. Attempt page sharing where possible
5. Allocate new pages and load content as needed

```
void do_no_page(unsigned long error_code, unsigned long address)
{
    /* Variable declarations */
    int nr[4];
    unsigned long tmp;
    unsigned long page;
    int block, i;
    struct m_inode *inode;

```

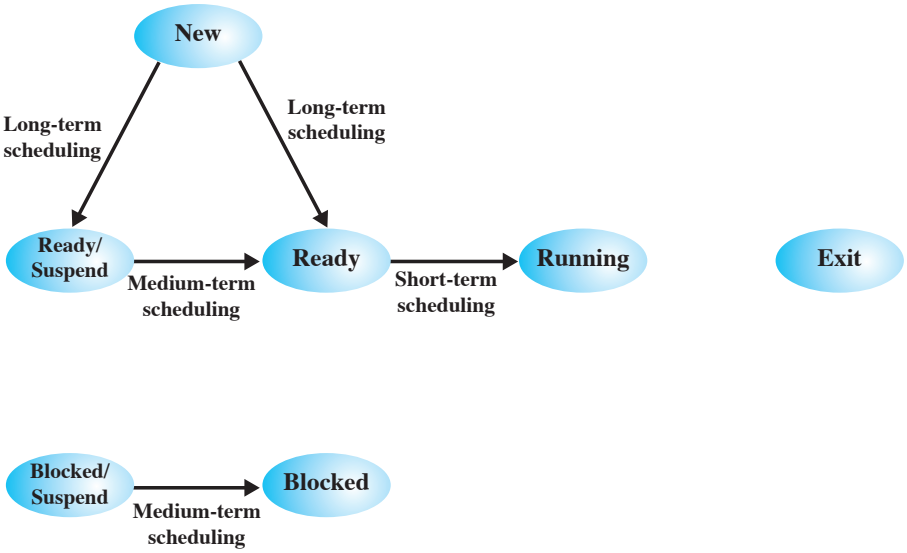
```
    // The function first validates whether the fault address is within acceptable ranges:
    // Error - kernel page missing
    if (address < TASK_SIZE)
        printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");

    // Error - address outside process space
    if (address - current->start_code > TASK_SIZE) {
        printk("Bad things happen: nonexistent page error in do_no_page\n\r");
        do_exit(SIGSEGV);
    }

    /*
     * The method is to first obtain the content of the page directory item corresponding to the
     * specified virtual address 'address', and then take out the address of the page table there,
     * and add the page table entry offset to obtain the corresponding page table entry pointer,
     * thereby obtaining the content of the page table entry. If the content of the entry is not
     * 0, but bit P=0, it indicates that the physical page specified by the page table entry should
     * be in the swap device. The function then exits after the specified page is loaded from the
     * swap device.

     * The swap device is a designated area on disk that the operating system uses
     * as virtual memory when physical RAM is insufficient.
     * A swap device serves as an extension of physical memory by temporarily storing
     * memory pages that aren't actively being used.
     * This allows the system to free up RAM for more immediate tasks and processes.
     */

    page = *(unsigned long *)((address >> 20) & 0xffc);  // page directory entry
    if (page & 1) {
        page &= 0xfffff000;                              // page table address
        page += (address >> 10) & 0xffc;                 // page table entry pointer
        tmp = *(unsigned long *)page;                    // page table entry content
        if (tmp && !(1 & tmp)) {
            swap_in((unsigned long *)page);              // read from swap device
            return;
        }
    }
    /*
     * By clearing these 12 bits, you get the starting address of the page that
     * contains the original address
     */
    address &= 0xfffff000;

    /*
     * current->start_code is the starting linear address of the current process's code segment
     * The subtraction gives you the offset from the start of the process's code segment
     */
    tmp = address - current->start_code;

    /*
     * Executable: This is the main program file that the process is running. It contains the
     * program's code, initial data, and headers.
     * Library: This refers to shared libraries that are mapped into the process's address space.

     * Early Linux used a memory layout where:
     * The executable was loaded at the beginning of the process's address space
     * Shared libraries were loaded starting at LIBRARY_OFFSET (typically higher in memory)
     * Dynamic memory (heap, stack) was allocated in between
     */

    /*
     Library Area Check: If the logical address tmp is at or beyond LIBRARY_OFFSET, then:

     * The data should come from the shared library file
     * block calculation: 1 + (tmp - LIBRARY_OFFSET) / BLOCK_SIZE

```

```
     * Subtracts LIBRARY_OFFSET to get the offset within the library
     * Divides by BLOCK_SIZE (1KB) to get the block number
     * Adds 1 to skip the file header block
     */
    if (tmp >= LIBRARY_OFFSET) {
        /* Page in library area */
        inode = current->library;
        block = 1 + (tmp - LIBRARY_OFFSET) / BLOCK_SIZE;
    }

    /*
     * Executable Area Check: If tmp is less than the end of the process's data segment:

     * The data should come from the executable file
     * block calculation: 1 + tmp / BLOCK_SIZE

     * Divides by BLOCK_SIZE to get the block number
     * Adds 1 to skip the file header block
     */
    else if (tmp < current->end_data) {
        /* Page in executable area */
        inode = current->executable;
        block = 1 + tmp / BLOCK_SIZE;
    }
    /*
     * Dynamic Memory: If neither of the above, this is dynamically allocated memory:

     * No file backing (inode = NULL)
     * No block to read (block = 0)
     * Will be handled by simply allocating an empty page
     */
    else {
        /* Dynamically requested memory */
        inode = NULL;
        block = 0;
    }

    /* For dynamic memory: Directly allocate empty page */
    if (!inode) {
        get_empty_page(address);
        return;
    }

    /* For file-backed pages: Attempt page sharing first */
    if (share_page(inode, tmp))
        return;

    /* If sharing fails: Allocate new page and load from disk */
    if (!(page = get_free_page()))
        oom();

    /* If allocation is successful, load the data from the disk */
    for (i = 0; i < 4; block++, i++)
        nr[i] = bmap(inode, block);

    bread_page(page, inode->i_dev, nr);

    /* This code segment is handling an edge case where the page being loaded might
     * contain data beyond the end of the process's data segment.
     * First, it calculates i = tmp + 4096 - current->end_data:
     * tmp is the logical address of the page in the process's space
     * 4096 is the page size in bytes
     * current->end_data is the end address of the process's data segment
     * So i represents how many bytes of the page extend beyond the end of the data segment

     * If i > 4095, it means the entire page is beyond the data segment,
```

```
     * which shouldn't happen in this context, so it sets i = 0 (no bytes to clear).
     * It sets tmp = page + 4096, which points to the byte just after the end of the newly loaded page.
     * Then it enters a loop that runs i times:

     * tmp-- moves backward through the page
     * *(char *)tmp = 0 zeros out each byte
    */

    i = tmp + 4096 - current->end_data;
    if (i > 4095)
        i = 0;

    tmp = page + 4096;
    while (i-- > 0) {
        tmp--;
        *(char *)tmp = 0;
    }

    /* Map page to address space, means update the page table information */
    if (put_page(page, address))
        return;

    /* if put_page() fails (out-of-memory condition)
     * Calls oom() which stands for "Out Of Memory" -
     * this is a function that typically kills the process or takes other drastic action
    */
    free_page(page);
    oom();
}
```

10 Uniprocessor Scheduling

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency. In many systems, this scheduling activity is broken down into three separate functions: long-, medium-, and short-term scheduling. The names suggest the relative time scales with which these functions are performed.



- Long-term scheduling is performed when a new process is created. This is a decision whether to add a new process to the set of processes that are currently active. Controls the degree of multiprogramming:
  - The more processes that are created, the smaller the percentage of time that each process can be executed.
  - Each time a job terminates, the scheduler may decide to add one or more new jobs.
  - If the fraction of time that the processor is idle exceeds a certain threshold, the long-term scheduler may be invoked.
- Medium-term scheduling is a part of the swapping function. This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution.
  - Memory Management: If the system is running low on memory, less active processes can be swapped out to free up space.
  - Priority Adjustment: Processes with lower priority or those waiting for certain events (like I/O completion) might be swapped out temporarily.
  - Load Balancing: It helps in balancing the load on the system by controlling the number of active processes.
- Short-term scheduling is the actual decision of which ready process to execute next, also known as the dispatcher
  - Executes most frequently
  - Makes the fine-grained decision of which process to execute next
  - Invoked when an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another (interrupts or signals).

  The primary goal of short-term scheduling is to maximize CPU utilization and system throughput by minimizing CPU idle time. It also aims to ensure fairness among processes, providing a reasonable response time to interactive users.

Between Long-term and Short-term: While long-term scheduling controls the admission of new processes into the system from jobs or user programs, short-term scheduling makes rapid decisions on which of the ready processes in memory to execute next on the CPU.

Medium-term: Medium-term scheduling provides a dynamic adjustment mechanism to the composition of processes, based on current system conditions and workload, which can change throughout the system's operation.

10.1 Short Term Scheduling

The main objective of short-term scheduling is to **allocate processor time in such a way as to optimize one or more aspects of system behavior**.

The commonly used criteria can be categorized along two dimensions.

- **User-oriented criteria** relate to the behavior of the system as perceived by the individual user or process. An example is response time in an interactive system. Response time is the elapsed time between the submission of a request until the response begins to appear as output. This quantity is visible to the user and is naturally of interest to the user. We would like a scheduling policy that provides "good" service to various users. In the case of response time, a threshold may be defined, say two seconds. Then a goal of the scheduling mechanism should be to maximize the number of users who experience an average response time of two seconds or less.
- Other criteria are **system oriented**. That is, the focus is on effective and efficient utilization of the processor. An example is throughput, which is the rate at which processes are completed. This is certainly a worthwhile measure of system performance and one that we would like to maximize. However, it focuses on system performance rather than service provided to the user. Thus, throughput is of concern to a system administrator but not to the user population.

Whereas user-oriented criteria are important on virtually all systems, system-oriented criteria are generally of minor importance on single-user systems. On a single-user system, it probably is not important to achieve high processor utilization or high throughput as long as the responsiveness of the system to user applications is acceptable.

Another dimension along which criteria can be classified is those that are performance related and those that are not directly performance related.

- Performance-related criteria are quantitative and generally can be readily measured. Examples include **response time and throughput**.
- Criteria that are not performance related are either qualitative in nature or do not lend themselves readily to measurement and analysis. An example of such a criterion is **predictability**. We would like for the service provided to users to exhibit the same characteristics over time, independent of other work being performed by the system. To some extent, this criterion can be measured by calculating variances as a function of workload. However, this is not nearly as straightforward as measuring throughput or response time as a function of workload.

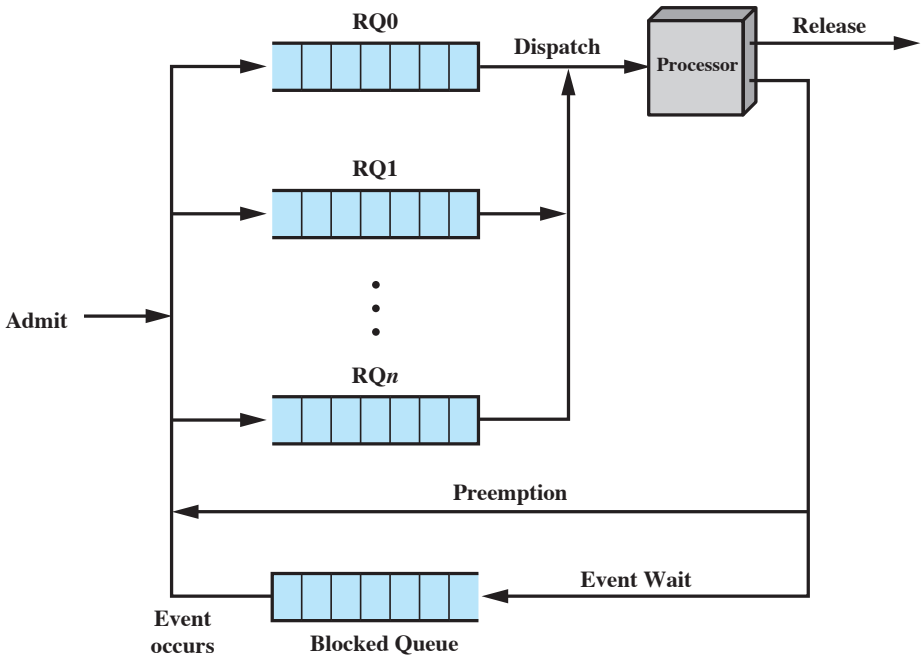
Predictability in operating system (OS) scheduling refers to the ability of the scheduling system to provide a consistent and deterministic timing for task execution.

Predictability implies that a job or task should have a consistent response time and cost every time it runs, independent of changes in system workload. For instance, if you submit a print job to a server, you would expect it to be completed in roughly the same amount of time on Monday morning (when system loads might be high) as on Saturday afternoon (when loads might be lower).

Hard Real-Time Systems: These systems require absolute predictability. Missing a deadline could lead to catastrophic failures. For example, in an airbag control system, failing to deploy the airbag in time could have dire consequences.

Soft Real-Time Systems: These systems also aim for predictability but can tolerate occasional deadline misses without catastrophic consequences. For example, in a video streaming application, missing a frame deadline might result in a temporary glitch but is not disastrous.

In many systems, each process is assigned a priority and the scheduler will always choose a process of higher priority over one of lower priority. Figure illustrates the use of priorities. For clarity, the queuing diagram is simplified, ignoring the existence of multiple blocked queues and of suspended states.



Instead of a single ready queue, we provide a set of queues, in descending order of priority: $RQ_0, RQ_1, \ldots, RQ_n$, with priority$[RQ_i]$ > priority$[RQ_j]$ for $i < j$. When a scheduling decision is to be made, the scheduler will start at the highest-priority ready queue (RQ_0). If there are one or more processes in the queue, a process is selected using some scheduling policy. If RQ_0 is empty, then RQ_1 is examined, and so on.

One problem with a pure priority scheduling scheme is that lower-priority processes may suffer **starvation**. This will happen if there is always a steady supply of higher-priority ready processes. If this behavior is not desirable, the priority of a process can change with its age or execution history.

- Non-preemptive: In this case, once a process is in the Running state, it continues to execute until it terminates or it blocks itself to wait for I/O or to request some OS service.
- Preemptive: The currently running process may be interrupted and moved to the Ready state by the OS.

  The decision to preempt may be performed:

  – when a new process arrives;

  – when an interrupt occurs that places a blocked process in the Ready state;

  – Periodically, based on a clock interrupt.

10.2 Short Term Scheduling Policies

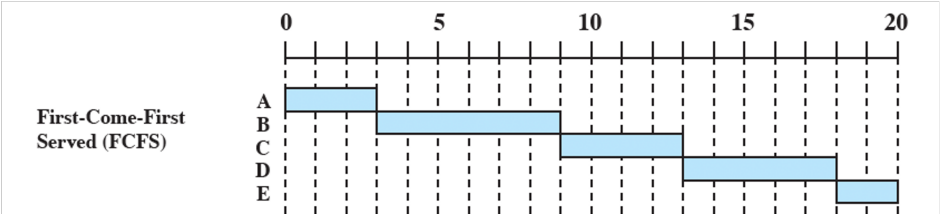
10.2.1 First Come First Served (FCFS)

Simplest scheduling policy.

Also known as first-in-first-out (FIFO) or a strict queuing scheme.

- As each process becomes ready, it joins the ready queue
- When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56



Performs much better for long processes than short ones: when short processes arrive behind longer processes in the queue, they have to wait for one or more longer processes to complete before they can start. This leads to higher average waiting times and turnaround times, especially for short processes.

Tends to favor processor-bound processes over I/O-bound processes: If an I/O-bound process follows a CPU-bound process in the FIFO queue, it must wait until the preceding process completes its execution. CPU-bound processes, which perform extensive computations, will occupy the CPU for longer periods, delaying the I/O-bound process unnecessarily.

Selection function	**Decision mode**	**Throughput**	**Response time**	**Overhead**	**Effect on processes**	**Starvation**
max[w]	Non-preemptive	Not emphasized	May be high, especially if there is a large variance in process execution times	Minimum	Penalizes short processes; penalize I/O bound processes	No

w is the waiting time.

10.2.2 Round Robin

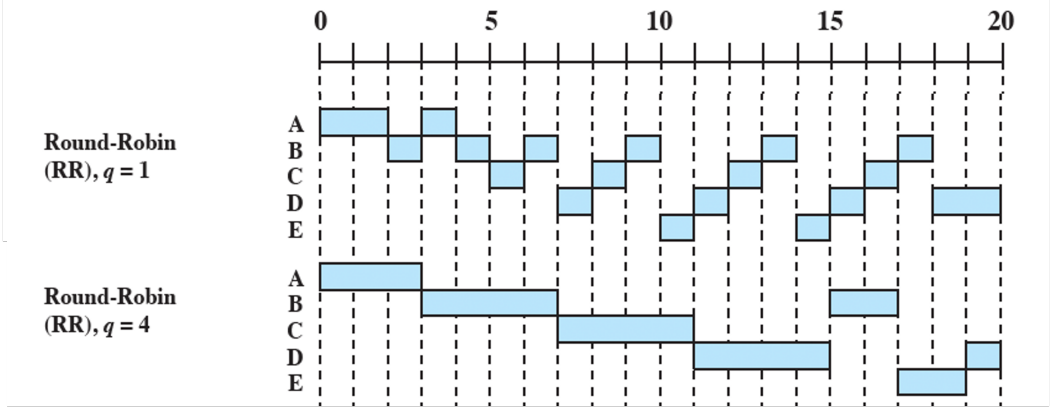
Uses preemption based on a clock.

Also known as time slicing because each process is given a slice of time before being preempted.

The principal design issue is the length of the time quantum, or slice, to be used.

A time quantum that is longer than the longest-running process, round robin degenerates to FCFS.

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_S)	3	6	4	5	2	Mean
RR q = 1						
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_S	1.33	2.67	3.25	2.80	3.50	2.71
RR q = 4						
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_S	1.00	2.5	1.75	2.80	5.50	2.71



With round robin, the principal design issue is the length of the time quantum, or slice, to be used:

- If the quantum is very short, then short processes will move through the system relatively quickly.
- On the other hand, there is processing overhead involved in handling the clock interrupt and performing the scheduling and dispatching function.

One drawback, consider a mix of processor-bound and I/O-bound processes.

- An I/O-bound process uses a processor for a short period and then is blocked for I/O; it waits for the I/O operation to complete and then joins the ready queue.
- On the other hand, a processor-bound process generally uses a complete time quantum while executing and immediately returns to the ready queue.

Thus, processor-bound processes tend to receive an unfair portion of processor time, which results in poor performance for I/O-bound processes, inefficient use of I/O devices, and an increase in the variance of response time.

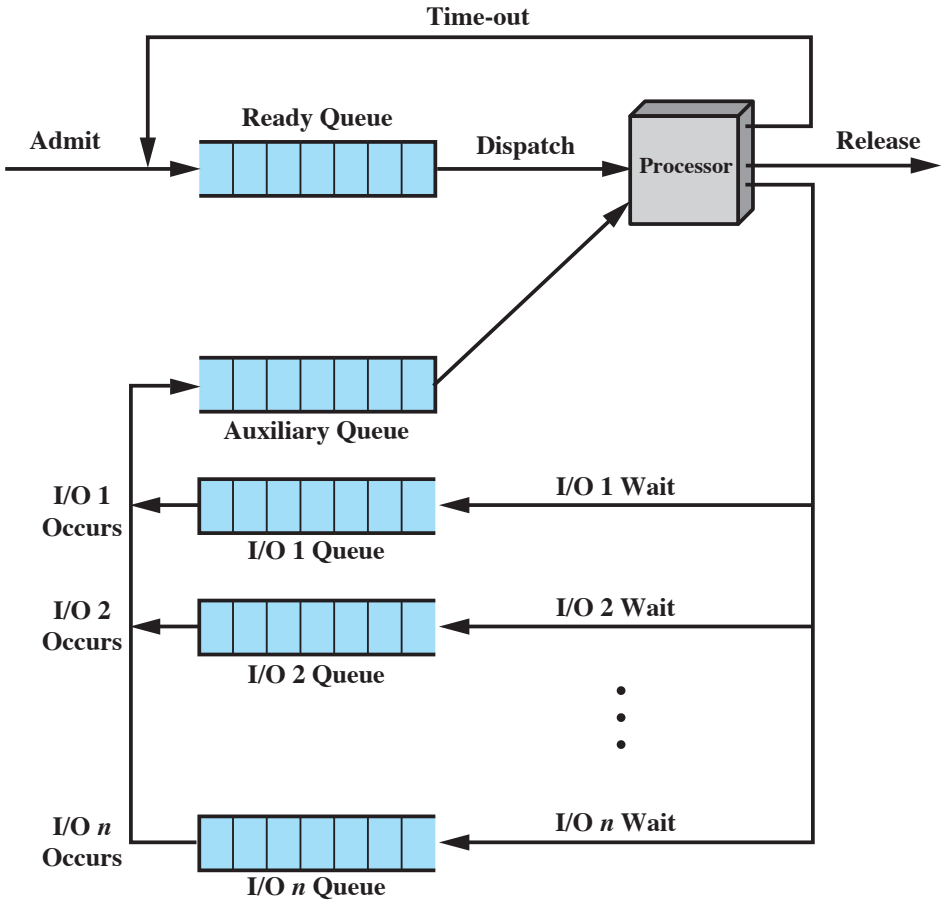
Selection function	**Decision mode**	**Throughput**	**Response time**	**Overhead**	**Effect on processes**	**Starvation**
Constant	Preemptive (at time quantum)	Low if quantum is too short	Provides good response time for short processes	Minimum	Fair treatment	No

Table 2: Round Robin Scheduling Algorithm Characteristics

The new feature is an FCFS auxiliary queue to which processes are moved after being released from an I/O block.

When a dispatching decision is to be made, processes in the auxiliary queue get preference over those in the main ready queue.

When a process is dispatched from the auxiliary queue, it runs no longer than a time equal to the basic time quantum minus the total time spent running since it was last selected from the main ready queue.



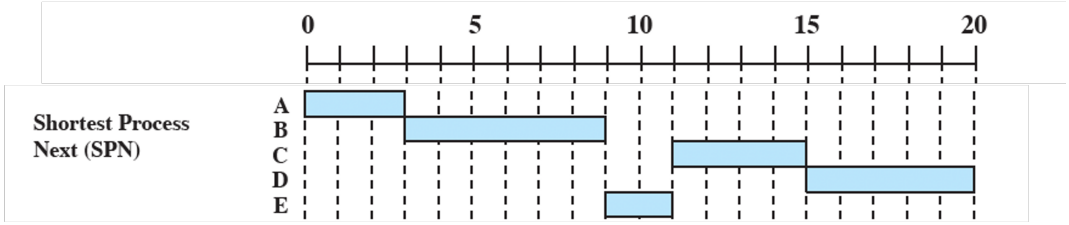
10.2.3 Shortest Process Next (SPN)

Nonpreemptive policy in which the process with the shortest expected processing time is selected next.

A short process will jump to the head of the queue and there is a possibility of starvation for longer processes.

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean

SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84



One difficulty is the need to know, or at least estimate, the required processing time of each process.

In a production environment, the same jobs run frequently:

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i \tag{9.1}$$

where

T_i = processor execution time for the ith instance of this process (total execution time for batch job; processor burst time for interactive job),
S_i = predicted value for the ith instance, and
S_1 = predicted value for first instance; not calculated.

To avoid recalculating the entire summation each time, we can rewrite Equation (9.1) as

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n \tag{9.2}$$

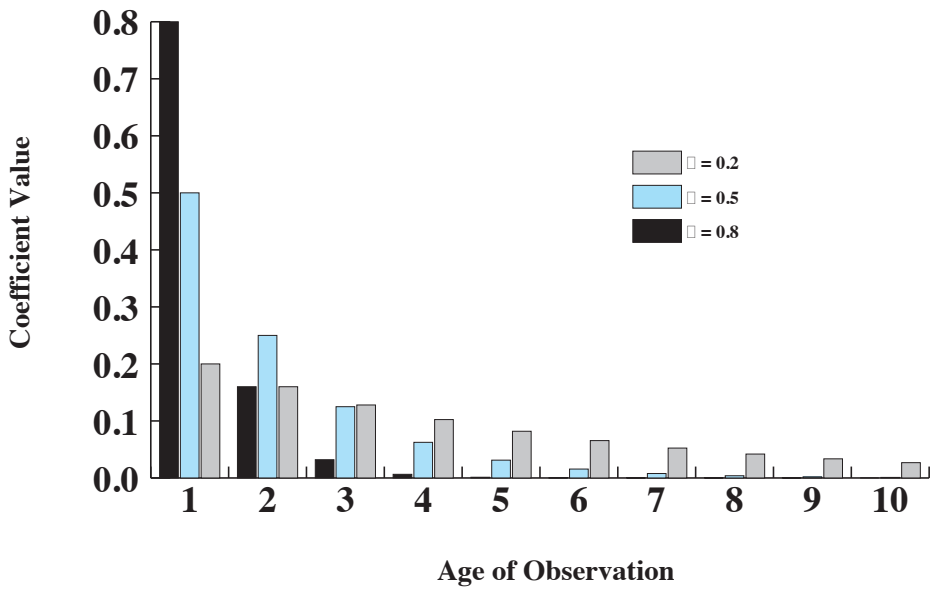
Note each term in this summation is given equal weight; that is, each term is multiplied by the same constant $1/n$.

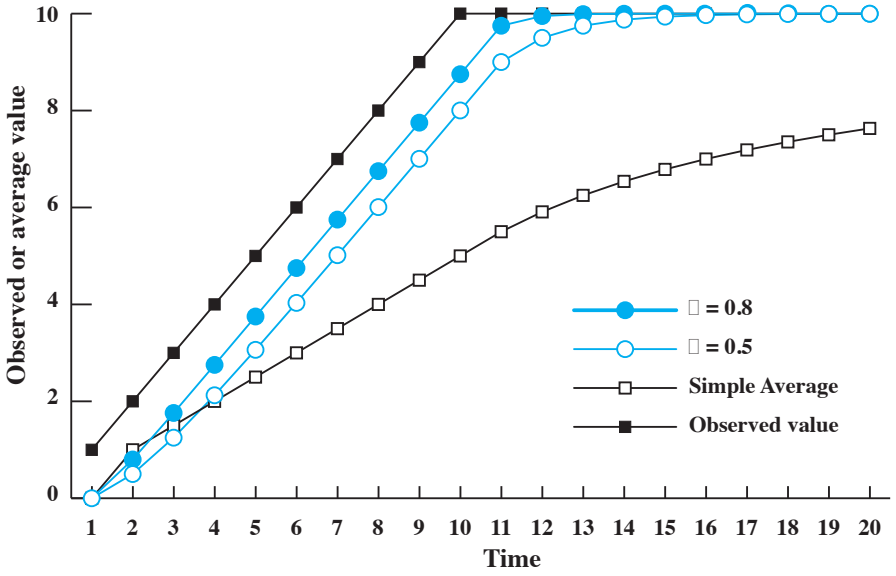
Typically, we would like to give **greater weight to more recent instances**, because these are more likely to reflect future behavior. A common technique for predicting a future value on the basis of a time series of past values is **exponential averaging**:

$$S_{n+1} = \alpha T_n + (1-\alpha)S_n \tag{9.3}$$

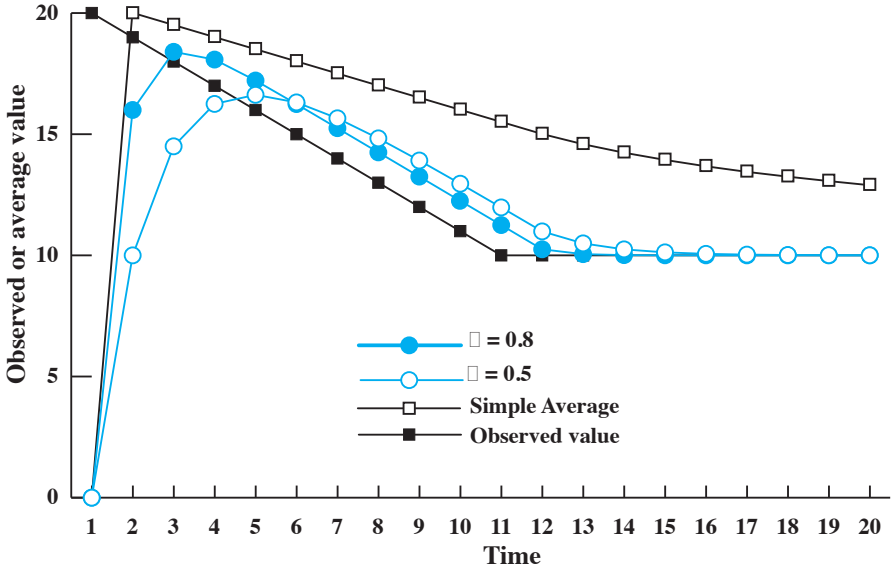
where α is a constant weighting factor ($0 < \alpha < 1$) that determines the relative weight given to more recent observations relative to older observations. Compare with Equation (9.2). By using a constant value of α, independent of the number of past observations, Equation (9.3) considers all past values, but the less recent ones have less weight. To see this more clearly, consider the following expansion of Equation (9.3):

$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1 \tag{9.4}$$





(a) Increasing function



(b) Decreasing function

Selection function	Decision mode	Throughput	Response time	Overhead	Effect on processes	Starvation
min[s]	Non-preemptive	High	Provides good response time for short processes	Can be high	Penalizes long processes	Possible

s is the total service time required by the process

10.2.4 Shortest Remaining Time (SRT)

Preemptive version of SPN.

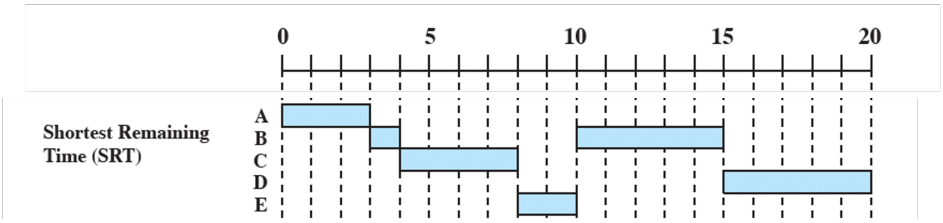
Scheduler always chooses the process that has the shortest expected remaining processing time.

Risk of starvation of longer processes.

When a new process joins the ready queue, it may in fact have a shorter remaining time than the currently running process. Accordingly, the scheduler may preempt the current process when a new process becomes ready.

Should give superior turnaround time performance to SPN because a short job is given immediate preference to a running longer job.

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59



Selection function	Decision mode	Throughput	Response time	Overhead	Effect on processes	Starvation
$\min[s - e']$	Preemptive (at arrival)	High	Provides good response time	Can be high	Penalizes long processes	Possible

e is the time spent in execution so far.

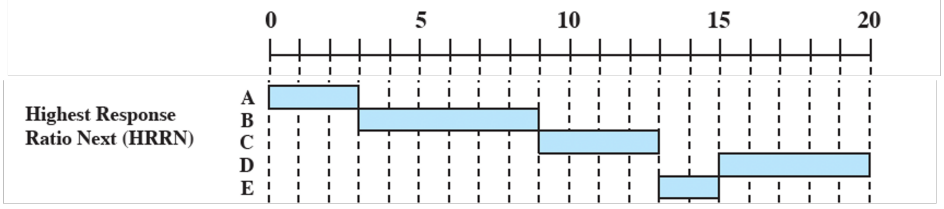
10.2.5 Highest Response Ratio Next (HRRN)

Chooses the next process with the greatest ratio:

$$\text{Ratio} = \frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}} \tag{8}$$

Attractive because it accounts for the age of the process. While shorter jobs are favored, aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs.

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14

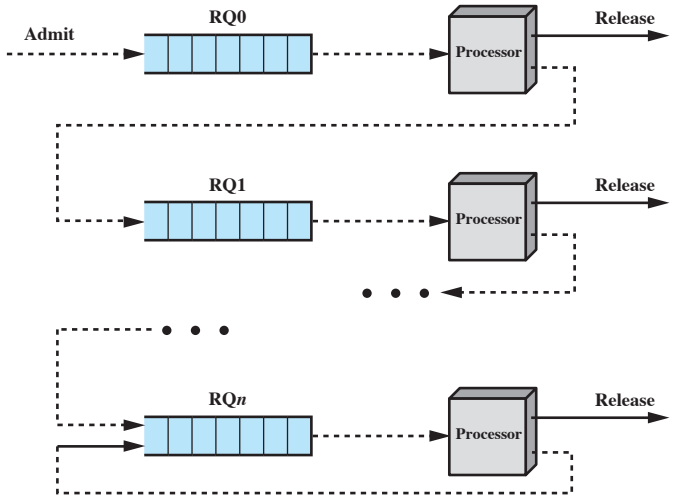


Selection function	Decision mode	Throughput	Response time	Overhead	Effect on processes	Starvation
$\max\left(\frac{w+s}{s}\right)$	Non-preemptive	High	Provides good response time	Can be high	Good balance	No

10.2.6 Feedback

If we have no indication of the relative length of various processes, then none of SPN, SRT, and HRRN can be used. Another way of establishing a preference for shorter jobs is to penalize jobs that have been running longer.

When a process first enters the system, it is placed in RQ0. After its first preemption, when it returns to the Ready state, it is placed in RQ1. Each subsequent time that it is preempted, it is demoted to the next lower-priority queue. Thus, newer, shorter processes are favored over older, longer processes.



I/O-bound processes typically use the CPU for short bursts, quickly performing computations before issuing I/O requests. After making such a request, they often enter a waiting state, yielding the CPU to other processes. Feedback schedulers can detect this pattern and may increase the priority of these processes because they use less CPU time and yield often, allowing more processes to run efficiently.

Selection function	**Decision mode**	**Throughput**	**Response time**	**Overhead**	**Effect on processes**	**Starvation**
-	Preemptive (at time quantum)	Not emphasized	Not emphasized	Can be high	May favor I/O bound processes	Possible

10.3 Performance Comparison

Any scheduling discipline that chooses the next item to be served independent of service time obeys the relationship:

$$\frac{T_r}{T_s} = \frac{1}{1-\rho} \tag{9}$$

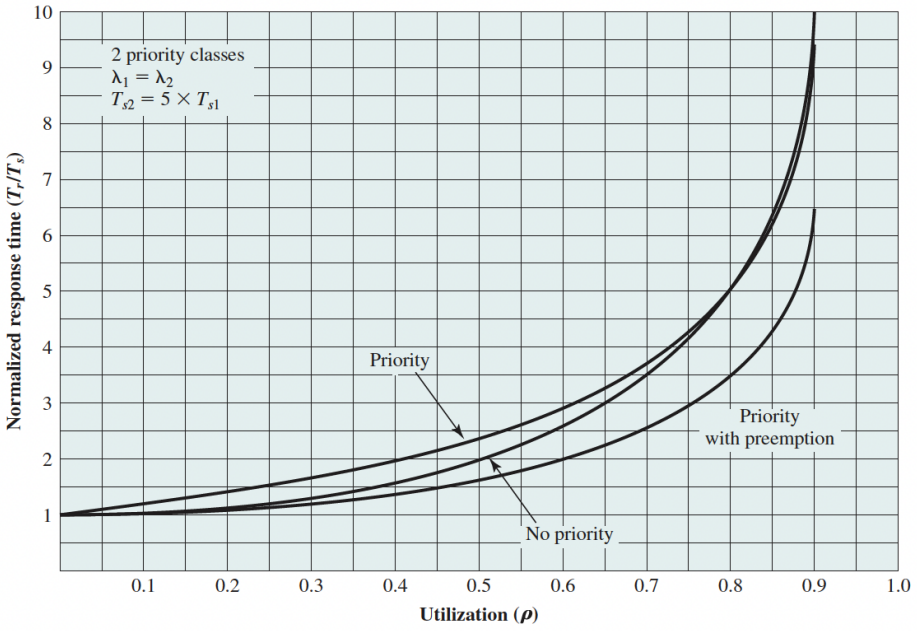
where

$$T_r = \text{turnaround time or residence time; total time in system, waiting plus execution} \tag{10}$$
$$\tag{11}$$
$$T_s = \text{average service time; average time spent in Running state} \tag{12}$$
$$\rho = \text{processor utilization} \tag{13}$$

This means that as the CPU gets busier, the turnaround time increases compared to the service time. This is because, with a high processor utilization, processes are likely to spend more time waiting in the queue.

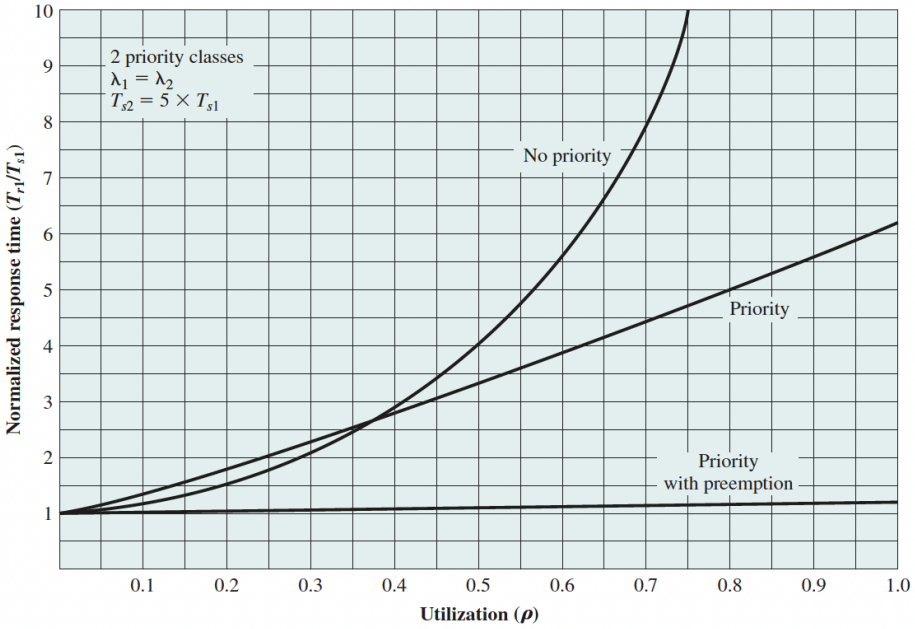
On the other hand, if the processor is not very utilized, the turnaround time will be closer to the actual service time, indicating that processes don't have to wait as long to be processed.

For all the scenarios we discuss below, priority 1 items are serviced before priority 2 items.(higher-priority, shorter processes)



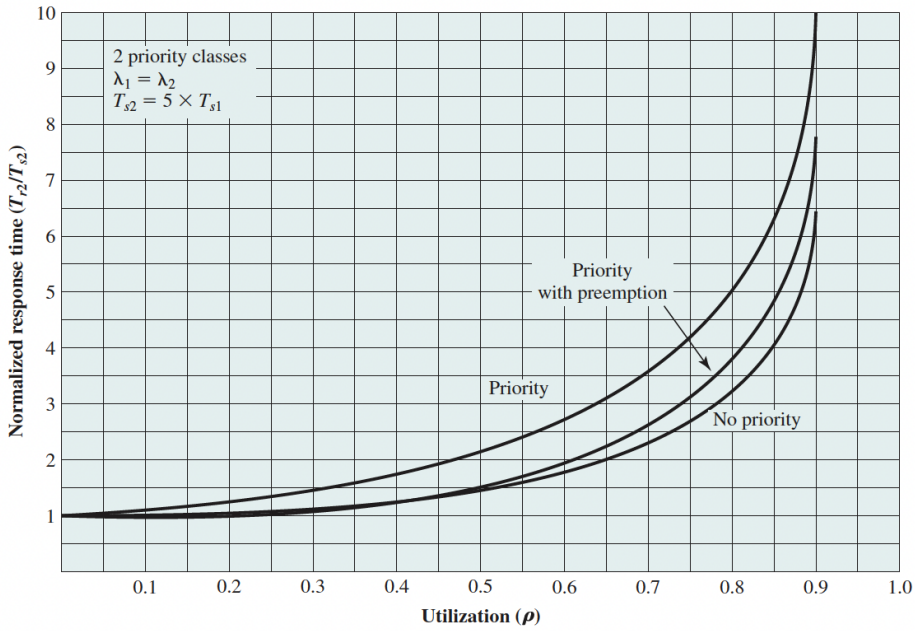
As utilization increases, the normalized response time increases as well due to more contention for the system resources.

Introducing preemption further improves response times for higher-priority processes because they can interrupt lower-priority processes and get serviced sooner.



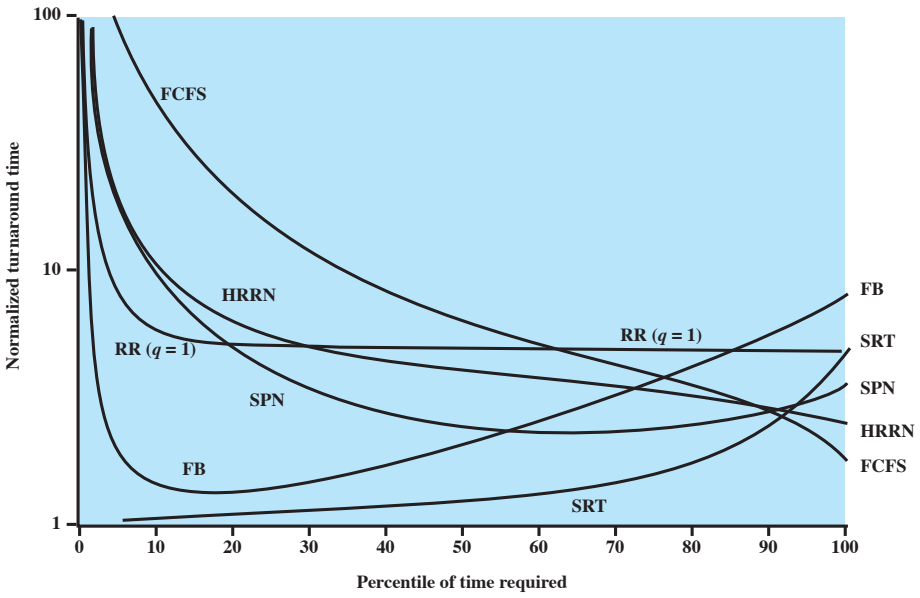
This is the figure of normalized response time for shorter processes. The normalized response time for priority 1 processes increases with utilization but generally at a slower rate than in the no-priority scenario. This is because priority 1 processes are given precedence, which should theoretically reduce their waiting time.

With preemption, this improvement occurs because priority 1 processes can interrupt priority 2 processes, thereby reducing their waiting time significantly.



This is the figure of normalized response time for longer processes. Priority 2 processes are deprioritized in favor of priority 1 processes, leading to increased waiting times for priority 2 processes as the system gets busier.

Impact of Preemption: better than non preemption priority. Because preemption increases the over all system performance, thus P2 processes benefit from it.

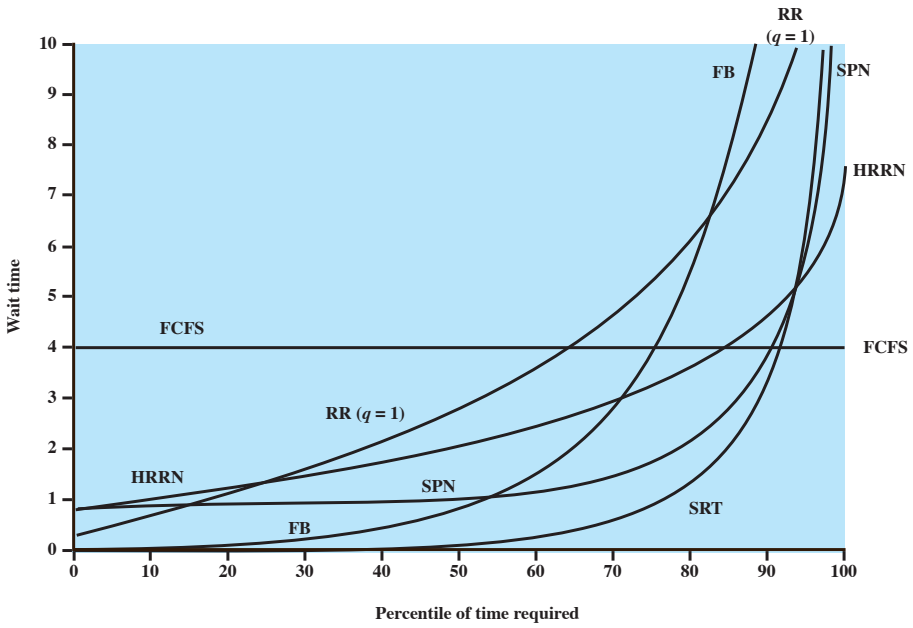


The simulation for normalized turnaround time. The simulation involved 50,000 processes with the processor utilization is 0.8. Note, therefore, that we are only measuring one utilization point.

To present the results, processes are grouped into service-time percentiles, each of which has 500 processes. Thus, the 500 processes with the shortest service time are in the first percentile; with these eliminated, the 500 remaining processes with the shortest service time are in the second percentile; and so on. This allows us to view the effect of various policies on processes as a function of the length of the process.

- **First Come First Served (FCFS)**: Short tasks get stuck behind long tasks, leading to increased overall wait times, especially for short processes.
- **Highest Response Ratio Next (HRRN)**: tends to favor processes that have waited a long time relative to their service time, which can benefit longer processes that have experienced significant wait times, while still providing reasonable wait times for shorter processes.
- **Shortest Process Next (SPN)**: can significantly reduce the wait time for short processes but can lead to starvation of longer processes, as they might get perpetually delayed behind shorter ones.

- **Round Robin (RR)**: RR ensures that no process starves and that each gets regular CPU time, but the frequent context switching can increase overhead, particularly affecting processes that could have completed quickly under other policies.
- **Feedback (FB)**: by adjusting their priorities based on execution history, potentially offering a balanced approach but with the complexity of managing dynamic priorities.
- **Shortest Remaining Time (SRT)**: preemptive and processor always select SRT process to execute. Thus, it can minimize wait times for short processes and is particularly effective in reducing response times. However, like SPN, it risks starving longer processes.



The figure shows the average waiting time.

- **FCFS (First-Come, First-Served):**
- The absolute waiting time is uniform, as is to be expected because scheduling is independent of service time.
- **Round Robin** (quantum = 1 time unit): Treats all processes fairly. Short has shorter waiting time and longer has longer waiting time.
- **SPN (Shortest Process Next):** Performs better than round robin, except for the shortest processes.
- **SRT (Shortest Remaining Time):**Performs better than SPN except for the longest 7% of all processes.
- **HRRN (Highest Response Ratio Next):** Intended to be a compromise between FCFS (which favors long processes) and SPN (which favors short ones).
- **Feedback**: Performs quite well for short processes.

10.4 Fair-Share Scheduling

Scheduling algorithm we've discussed so far:

- view all processes as a single homogeneous pool.
- Selection is based solely on process readiness and, occasionally, priority levels.

Challenges in Multiuser Systems:

- Multiuser systems complicate scheduling with multiple processes per user.
- Traditional schedulers fail to recognize structured collections of user-specific processes.

From the user's view:

- Users are more concerned with the performance of their entire set of processes.
- Scheduling decisions should ideally consider the collective performance of these process sets.

Fair-Share Scheduling:

- Fair-share scheduling aims to allocate system resources based on user or group process sets.
- Can extend to groups, ensuring users from the same department are treated as a collective.

The following formulas apply for process j in group k:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} \tag{14}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2} \tag{15}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k} \tag{16}$$

where

$$CPU_j(i) = \text{measure of processor utilization by process } j \text{ through interval } i, \tag{17}$$

$$GCPU_k(i) = \text{measure of processor utilization of group } k \text{ through interval } i, \tag{18}$$

$$P_j(i) = \text{priority of process } j \text{ at beginning of interval } i; \text{ lower values equal} \tag{19}$$

$$\text{higher priorities,} \tag{20}$$

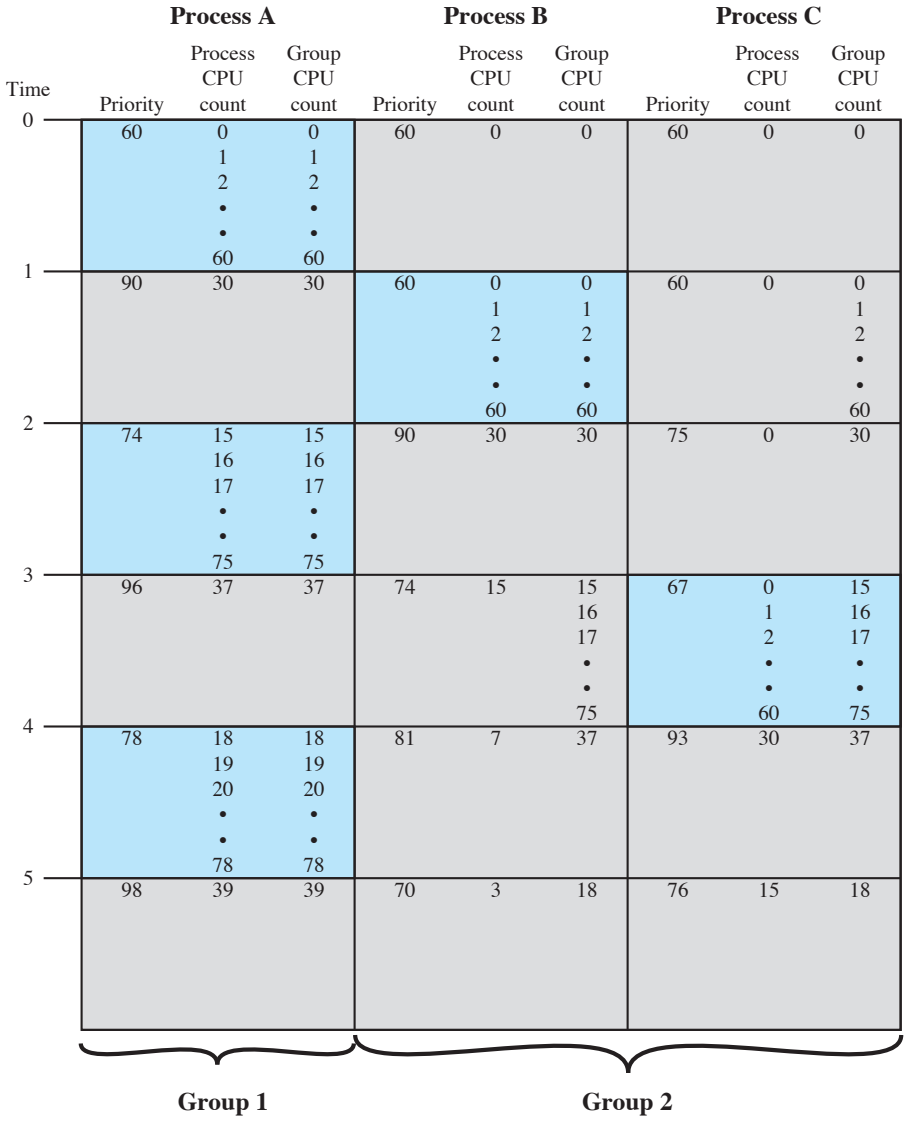
$$Base_j = \text{base priority of process } j, \text{ and} \tag{21}$$

$$W_k = \text{weighting assigned to group } k, \text{ with the constraint that and} \tag{22}$$

$$0 < W_k \leq 1 \text{ and } \sum_k W_k = 1. \tag{23}$$

The priority of a process drops as the process uses the processor and as the group to which the process belongs uses the processor.

In the case of the group utilization, the average is normalized by dividing by the weight of that group. The greater the weight assigned to the group, the less its utilization will affect its priority.



process A is in one group and processes B and C are in a second group, with each group having a weighting of 0.5. Assume that all processes are processor bound and are usually ready to run. All processes have a base priority of 60.

11 Multiprocessor Scheduling

We begin with a concise overview of multiprocessor architectures before examining the distinct considerations involved in scheduling at both process and thread levels.

Multiprocessor systems can be categorized as follows:

- **Loosely coupled or distributed multiprocessor (cluster):** Comprises a collection of relatively independent systems, with each processor having dedicated main memory and I/O channels.
- **Functionally specialized processors:** Exemplified by I/O processors. In this architecture, a master general-purpose processor controls specialized processors that provide specific services.
- **Tightly coupled multiprocessor:** Consists of processors that share common main memory and operate under the integrated control of a single operating system.

A good way of characterizing multiprocessors and placing them in context with other architectures is to consider the synchronization granularity, or frequency of synchronization, between processes in a system.

1. **Independent Parallelism**: In loosely coupled multiprocessors or distributed cluster systems, explicit synchronization between processes is often unnecessary. Time-sharing systems represent a typical application of this parallelism model, characterized by:
   - Concurrent execution of independent tasks
   - Absence of inter-task dependencies
   - Decomposition of workloads into smaller computational units
   - Minimal requirement for synchronization or coordination

   In such environments, each user independently performs distinct applications (e.g., word processing or spreadsheet calculations). The multiprocessor environment effectively delivers functionality comparable to a multiprogrammed uniprocessor system, with the significant advantage of reduced average response times due to the availability of multiple processing units.
2. **Coarse And Very Coarse Grained Parallelism**: Coarse-grained and very coarse-grained parallelism refer to approaches where tasks or units of execution in a parallel program are relatively large, encompassing significant portions of the overall computation.

   The researchers have developed a program that takes a specification of files needing recompilation to rebuild a piece of software and determines which of these compiles (usually all of them) can be run simultaneously. The program then spawns one process for each parallel compile. The authors report that the speedup on a multiprocessor actually exceeds what would be expected by simply adding up the number of processors in use, due to synergies in the disk buffer caches and sharing of compiler code, which is loaded into memory only once.

   These approaches simplify parallel program design and management but may not fully exploit the available parallelism, potentially leading to underutilization of computational resources.
3. **Medium-Grained Parallelism**: Single application can be effectively implemented as a collection of threads within a single process Programmer must explicitly specify the potential parallelism of an application. There needs to be a high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization. Because the various threads of an application interact so frequently, scheduling decisions concerning one thread may affect the performance of the entire application.
4. **Fine-Grained Parallelism**: Fine-grained parallelism enables a high degree of concurrency, as many tasks can be executed simultaneously. Since tasks are small, there are often numerous tasks available for execution at any given time, allowing for efficient utilization of available processing resources. Excessive fine-grained parallelism may lead to increased overhead and decreased performance, particularly if tasks are too small or if synchronization overhead outweighs the benefits of parallelism.

11.1 Processes assignment

In a uniform multiprocessor architecture (where no processor has privileged access to memory or I/O devices), processors can be treated as a pooled resource with processes assigned on demand. This raises the question of static versus dynamic assignment:

`Two Types of Assignment`

- **Static Assignment:**
  - Process is permanently assigned to one processor from activation until completion

  - Each processor maintains its own dedicated short-term queue
  - **Advantages:**
    * Potentially less scheduling overhead (assignment occurs only once)
    * Enables group/gang scheduling strategies
  - **Disadvantage:** Processor load imbalance (one processor may be idle while another has a backlog)
- **Dynamic Assignment:**
  - **Common Queue Approach:**
    * All processes enter one global queue
    * Processes are scheduled to any available processor
    * A process may execute on different processors throughout its lifetime
    * In tightly coupled shared-memory architectures, context information is available to all processors
    * Scheduling cost is independent of processor identity

`Two Ways of Assigning A Process to A Processor`

Both dynamic and static methods require some way of assigning a process to a processor:

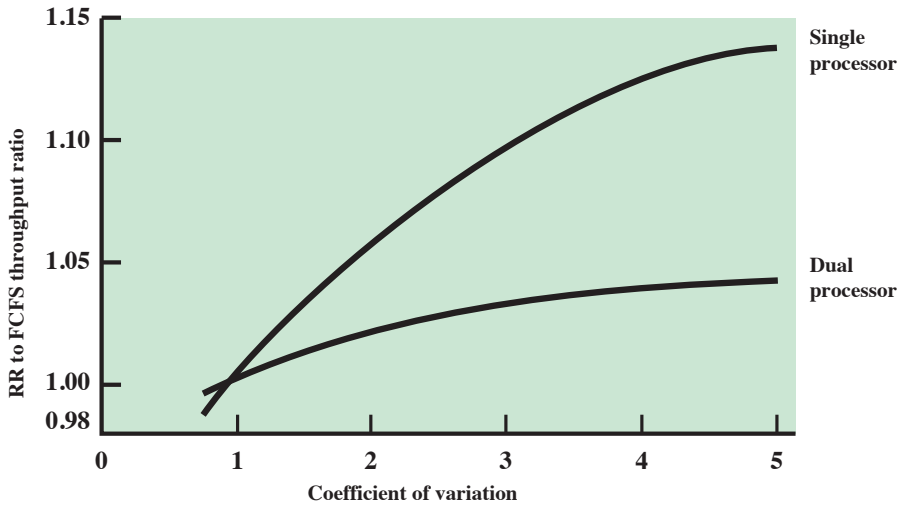
1. In a **master/slave architecture** for multiprocessor systems:
   - Essential kernel functions execute exclusively on a designated master processor
   - Slave processors are restricted to running user programs only
   - The master processor handles all scheduling responsibilities
   - When a slave requires system services (such as I/O operations):
     - It must submit a request to the master
     - It must wait until the master completes the requested service
   - This design offers simplicity, requiring minimal modifications to existing uniprocessor multiprogramming operating systems
   - Resource conflict resolution is streamlined since one processor controls all memory and I/O resources

   However, this architecture presents two significant **disadvantages**:
   - Any failure of the master processor causes complete system failure
   - The master processor can become a performance bottleneck as system demands increase
2. In a **peer architecture** for multiprocessor systems:
   - The kernel can execute on any processor
   - Each processor performs self-scheduling from the available process pool
   - This approach introduces complexity to the operating system design:
     - Must prevent multiple processors from selecting the same process
     - Must ensure processes aren't lost from the queue
     - Requires techniques to resolve and synchronize competing resource claims
   - Between master-slave and peer architectures exists a spectrum of intermediate approaches:
     - Dedicating a subset of processors to kernel processing instead of just one
     - Managing differences between kernel and user processes based on priority and execution history

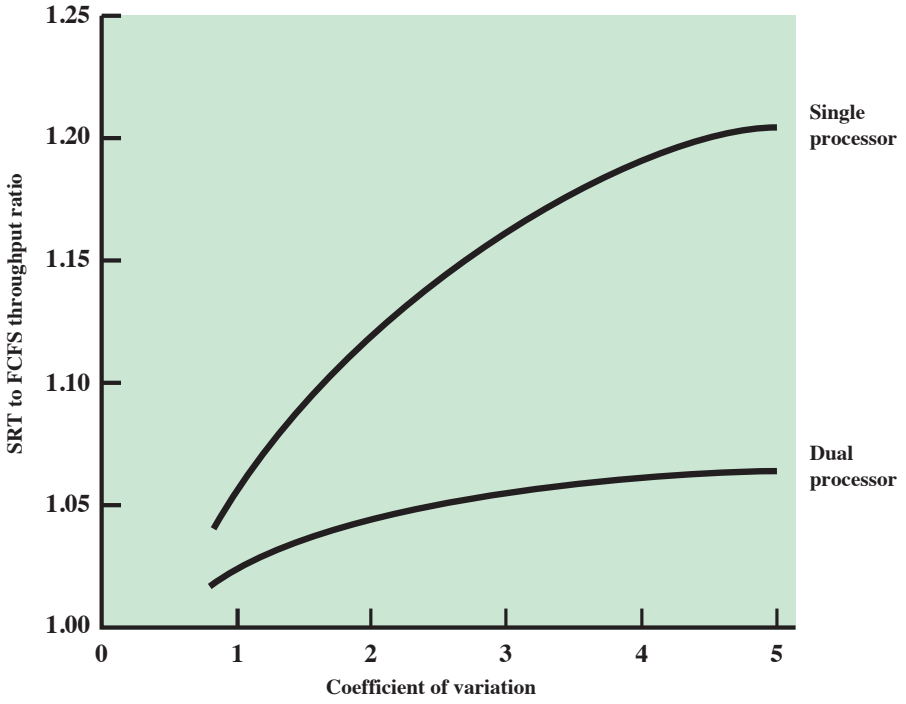
11.2 Process Scheduling

- In most traditional multiprocessor systems, processes are not dedicated to specific processors.
- A single queue serves all processors in the system.
- When priority schemes are implemented, multiple queues based on priority levels feed into the common processor pool.

- The overall system can be conceptualized as a multi-server queuing architecture.



(a) Comparison of RR and FCFS



(b) Comparison of SRT and FCFS

- Each processor in the dual system operates at half the processing rate of a single-processor system.
- The coefficient of variation corresponds to processes' service times:
  - A value of 0 indicates no variability
  - Higher values indicate greater variability among service times
- The y-axis represents the RR to FCFS throughput ratio:
  - Larger numbers indicate RR performs better than FCFS
- Key conclusions:
  - The specific scheduling discipline is much less important with two processors than with one
  - This conclusion becomes even stronger as the number of processors increases
  - A simple FCFS discipline or FCFS within a static priority scheme may suffice for a multiple-processor system

11.3 Thread Scheduling

- Thread execution is decoupled from the overall process definition
- Applications can be structured as sets of concurrent, cooperating threads sharing a common address space
- In uniprocessor systems, threads serve as:
  - A program organization technique
  - A means to overlap I/O operations with processing
- In multiprocessor environments, threads enable exploitation of true application parallelism
- Multiprocessor systems can achieve substantial performance improvements through threading
- For thread-intensive applications with high inter-thread communication:
  - Even minor differences in thread management can significantly impact performance
  - Thread scheduling optimizations become particularly important

11.3.1 Load Sharing

Load sharing is perhaps the simplest approach that carries over most directly from a uniprocessor environment. Its **advantages** include:

- Even distribution of load across processors, ensuring no processor remains idle while work is available
- No requirement for a centralized scheduler; the operating system's scheduling routine runs on any available processor

Three versions of load sharing:

- **First-come-first-served (FCFS):**
  - When a job arrives, all its threads are placed consecutively at the shared queue's end
  - Idle processors select the next ready thread, executing it until completion or blocking
- **Smallest number of threads first:**
  - The shared ready queue functions as a priority queue
  - Highest priority goes to threads from jobs with the fewest unscheduled threads
  - Jobs of equal priority are ordered by arrival time
  - Scheduled threads run until completion or blocking
- **Preemptive smallest number of threads first:**
  - Highest priority given to jobs with the fewest unscheduled threads
  - An arriving job with fewer threads than an executing job will preempt threads of the scheduled job

Disadvantages of Load Sharing:

- Central queue requires memory region that must be accessed with mutual exclusion enforcement
- Potential bottlenecks when multiple processors simultaneously seek work
- Preemptive threads typically resume execution on different processors
- Reduced caching efficiency
- Common thread pool approach decreases likelihood of simultaneous processor access for all threads of a program
- Performance degradation due to process switches when high coordination is required between program threads

A refinement of the load-sharing technique is used in the Mach operating system:

- The operating system maintains two types of run queues:
  - A local run queue for each processor
  - A shared global run queue
- The local run queue is used by threads that have been temporarily bound to a specific processor
- A processor examines the local run queue first to give bound threads absolute preference over unbound threads

- Example application: one or more processors could be dedicated to running operating system processes

11.3.2 Gang Scheduling

The concept of scheduling multiple processes simultaneously on multiple processors predates thread usage.This is described as "group scheduling" with these benefits:

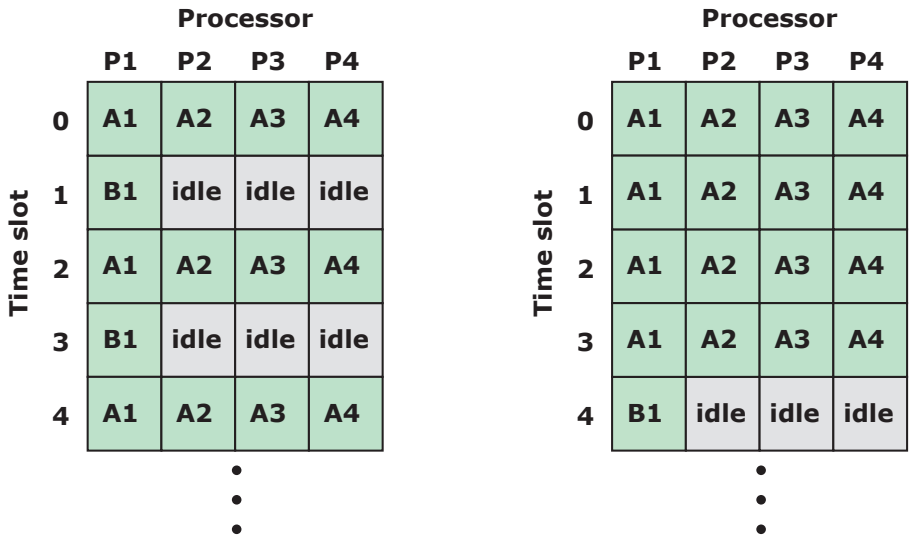
- Reduced synchronization blocking and process switching for related processes, improving performance
- Lower scheduling overhead as a single decision affects multiple processors and processes

"Gang scheduling" refers to simultaneously scheduling all threads within a single process. Gang scheduling is valuable for:

- Medium-grained to fine-grained parallel applications whose performance degrades when parts are not running while others are ready
- Any parallel application, even less performance-sensitive ones

Gang scheduling improves performance in several ways:

- Minimized process switches: When one thread needs to synchronize with another thread in the same process, no delay occurs waiting for the other thread to be scheduled
- Reduced resource allocation overhead: For example, multiple gang-scheduled threads can access a file without the additional locking overhead during seek and read/write operations



(a) Uniform scheduling **(b) Weighted scheduling**

Suppose that we have N processors and M applications, each of which has N or fewer threads. Then each application could be given $1/M$ of the available time on the N processors, using time slicing.

- Consider an example with two applications:
  - Application 1: four threads
  - Application 2: one thread
- Using uniform time allocation:
  - Each application gets 50% of processing time
  - Wastes 37.5% of the processing resource
  - Note: If there are several one-thread applications, these could all be fit together to increase processor utilization
- Alternative: scheduling weighted by number of threads
  - Application 1 (four threads): gets four-fifths (80%) of the time
  - Application 2 (one thread): gets one-fifth (20%) of the time

- Reduces processor waste to 15%

11.3.3 Dedicated Processor Assignment

When an application is scheduled, each thread receives a dedicated processor that remains assigned exclusively to that thread until the application completes execution.

- If a thread becomes blocked due to I/O operations or synchronization requirements with other threads, its assigned processor remains idle.
- The system does not implement multiprogramming of processors.
- Justification for this approach:
  - In highly parallel systems with dozens or hundreds of processors, processor utilization becomes less critical as a performance metric.
  - The complete elimination of process switching during program execution can yield significant performance improvements for that program.

The following experiment supports the statement:

The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program

The authors ran two applications simultaneously (executing concurrently), a matrix multiplication and a fast Fourier transform (FFT) calculation, on a system with 16 processors.

Number of threads per application	**Matrix multiplication**	**FFT**
1	1	1
2	1.8	1.8
4	3.8	3.8
8	6.5	6.1
12	5.2	5.1
16	3.9	3.8
20	3.3	3
24	2.8	2.4

The table shows that the performance of both applications worsens considerably when the number of threads in each application exceeds eight and thus the total number of processes in the system exceeds the number of processors. Furthermore, the larger the number of threads the worse the performance gets, because there is a greater frequency of thread preemption and rescheduling.

Both dedicated processor assignment and gang scheduling tackle scheduling by focusing on how processors are allocated to programs. **The challenge of assigning processors on a multiprocessor system is more like allocating memory on a single-processor system than scheduling tasks on a single processor.**

The key question is how many processors to give a program at any moment, which resembles deciding how many page frames to assign to a process. An "activity working set" is similar to a memory working set. It represents the minimum number of threads that need to run at the same time for an application to function properly.

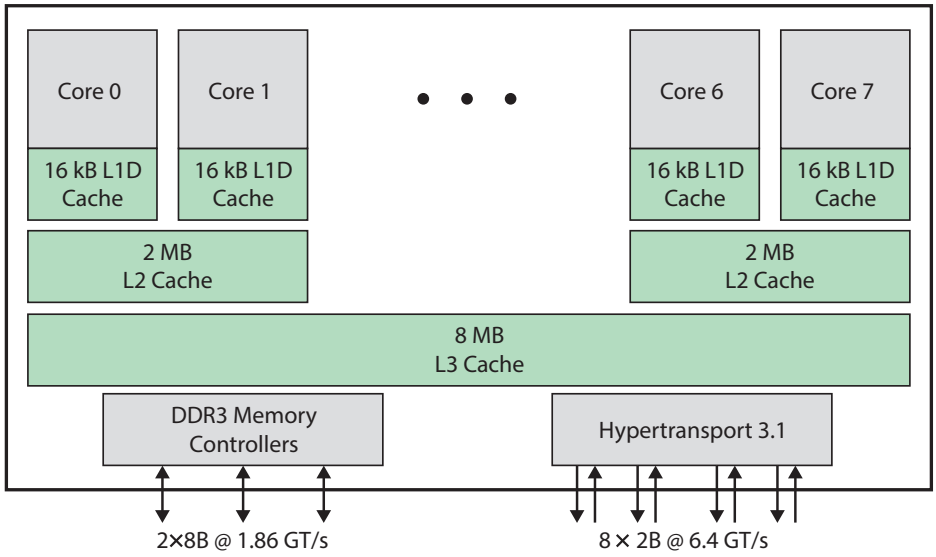
Just as insufficient memory can cause memory thrashing, failing to schedule all threads in an activity working set can lead to processor thrashing. "Processor fragmentation" happens when some processors remain unused after others are allocated, and these leftover processors are either too few or inappropriately configured to meet the needs of waiting applications. This is similar to memory fragmentation.

11.3.4 Dynamic Scheduling

In certain applications, it is possible to implement language and system tools that allow dynamic adjustment of thread count within a process. This capability enables the operating system to optimize load distribution and improve utilization.

- The operating system partitions processors among jobs
- Each job utilizes its allocated processors to execute runnable tasks by mapping them to threads
- Applications make decisions about which task subset to run and which threads to suspend when preempted

This approach may not suit all applications. Some applications might default to single-thread execution while others could be programmed to leverage this operating system feature.



In this model, the operating system's scheduling responsibility primarily focuses on processor allocation following these policies:

When a job requests processors:

- If idle processors exist, use them to fulfill the request
- If the job is a new arrival and no idle processors exist, allocate it one processor by reassigning from any job currently using multiple processors
- Any unsatisfied portion of the request remains pending until processors become available or the job rescinds the request

Upon processor release:

- Scan the queue of unsatisfied requests and assign one processor to each job with no current allocation
- Scan again, distributing remaining processors on a First-Come-First-Served basis

This approach outperforms gang scheduling and dedicated processor assignment for applications supporting dynamic scheduling. However, potential overhead might offset these performance advantages.

11.4 Multicore Thread Scheduling

Cache sharing involves two distinct aspects that must be considered:

- **Cooperative Resource Sharing**
  - Occurs when multiple threads access the same main memory locations
  - Common scenarios include:
    - Multithreaded applications
    - Producer-consumer thread interactions
  - Data brought into cache by one thread benefits cooperating threads
  - Scheduling strategy: Place cooperating threads on adjacent cores
- **Resource Contention**
  - Occurs when threads on adjacent cores compete for cache memory
  - Under any replacement policy (e.g., LRU), allocating more cache to one thread reduces available space for competing threads
  - Performance degradation results from reduced cache availability
  - Contention-aware scheduling aims to maximize shared cache effectiveness while minimizing off-chip memory access

11.5 Real-Time System

Real-time computing is an increasingly crucial field in computer science with the following characteristics:

- The operating system, particularly the scheduler, represents the most critical component of any real-time system.
- Current applications include:
  - Laboratory experiment control
  - Industrial plant process control
  - Robotics
  - Air traffic control
  - Telecommunications
  - Military command and control systems
- Next-generation systems will incorporate:
  - Autonomous land rovers
  - Controllers for elastic-joint robots
  - Intelligent manufacturing systems
  - Space station systems
  - Undersea exploration technologies

Real-time computing can be defined and characterized as follows:

- System correctness depends on:
  - The logical result of computation
  - The time at which results are produced
- A real-time system contains tasks with varying degrees of urgency
- Real-time tasks control or react to external world events
- Since these events occur in "real time," corresponding tasks must maintain synchronization with their relevant events

Task deadlines in real-time systems can be categorized as follows:

- Tasks are typically associated with deadlines that specify either:
  - A start time
  - A completion time
- Tasks are classified based on deadline requirements:
  - **Hard real-time tasks:** Must meet their deadlines
    - Missing a deadline causes unacceptable damage
    - Can result in fatal system errors
  - **Soft real-time tasks:** Have desirable but not mandatory deadlines
    - Tasks remain meaningful even after deadlines pass
    - Scheduling and completion still make sense beyond deadline

Real-time tasks can be classified also based on their recurrence pattern:

- **Aperiodic tasks:** These tasks have specific timing constraints:
  - They must finish by a defined deadline
  - They must start by a specific time
  - In some cases, both start and finish times are constrained
- **Periodic tasks:** These tasks follow a regular pattern:
  - They must execute "once per period T"

– Alternatively, they must execute "exactly T units apart"

11.5.1 Characteristics of Real Time Systems

Real-time operating systems can be characterized as having unique requirements in five general areas:

- **Determinism**
  - An operating system's deterministic quality is measured by its ability to execute operations at fixed, predetermined times or within specified time intervals (Complete determinism becomes impossible when multiple processes compete for resources and processor time.)
  - In real-time operating systems, deterministic response capability depends on:
    * Speed of interrupt response
    * System capacity to handle all requests within required timeframes
  - A key metric for operating system determinism is the maximum delay between:
    * Arrival of a high-priority device interrupt
    * Beginning of service for that interrupt
- **Responsiveness** A related but distinct characteristic from determinism is responsiveness. While determinism concerns how long an operating system delays before acknowledging an interrupt, responsiveness deals with how long it takes to service the interrupt after acknowledgment.
  - **Initial Handling Time**: The time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR). This delay increases if the ISR execution requires a process switch rather than execution within the current process context.
  - **ISR Execution Time**: The amount of time needed to perform the interrupt service routine itself. This factor generally depends on the hardware platform specifications.
  - **Interrupt Nesting Effect**: When an ISR can be interrupted by another interrupt's arrival, the service experiences additional delays.

  Determinism and responsiveness collectively constitute the response time to external events. For real-time systems, these response time requirements are critical because such systems must meet specific timing requirements imposed by individuals, devices, and data flows external to the system.
- **User control**

  User control capabilities are significantly more extensive in real-time operating systems compared to conventional operating systems:
  - **Granularity of Control:** While non-real-time operating systems typically offer limited scheduling control or only broad guidance (such as basic priority classes), real-time systems provide fine-grained control over task priorities.
  - **Task Classification:** Real-time systems enable users to explicitly distinguish between hard and soft tasks, with the ability to specify relative priorities within each category.
  - **System Configuration:** Users of real-time systems may have additional control capabilities including:
    * Specification of paging or process swapping mechanisms
    * Designation of processes that must remain resident in main memory
    * Selection of disk transfer algorithms
    * Definition of privileges for processes in various priority bands
- **Reliability**

  Real-time systems require significantly higher reliability standards than their non-real-time counterparts:
  - While non-real-time systems can often recover from transient failures through simple rebooting mechanisms
  - And processor failures in multiprocessor non-real-time systems might only cause reduced service levels until repair
  - Real-time systems must continuously respond to and control events as they occur
  - Performance degradation or system loss in real-time contexts may result in:
    * Financial losses

    - Major equipment damage
    - Potential loss of human life
  - The time-critical nature of these systems means failures cannot be tolerated as they might be in conventional systems

- **Fail-soft operation**

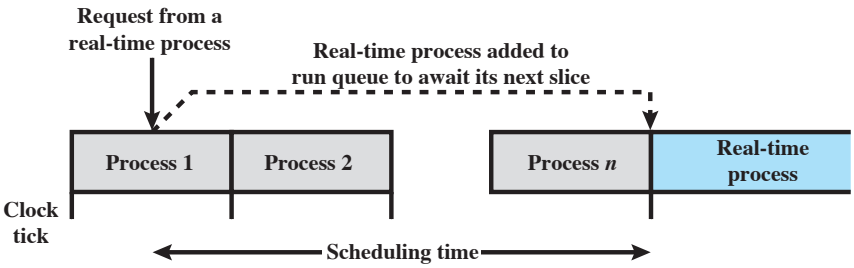
  Fail-soft operation refers to a system's capacity to preserve maximum functionality and data during failures. Unlike traditional UNIX systems that may terminate upon detecting kernel data corruption, real-time systems attempt to:

  - Correct problems or minimize their effects while continuing operation
  - Notify users or processes to initiate corrective measures
  - Continue functioning, potentially with reduced service levels
  - Maintain file and data consistency if shutdown becomes necessary

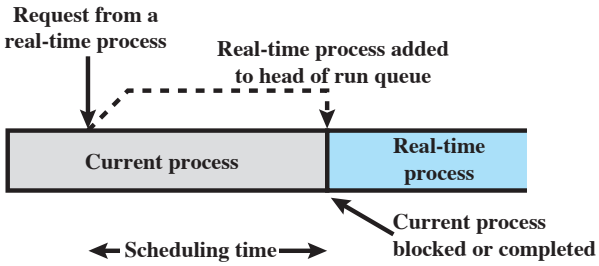
  Stability constitutes a crucial aspect of fail-soft operation, ensuring that when meeting all task deadlines becomes impossible, the system prioritizes completing deadlines for critical, high-priority tasks, even if less critical tasks experience delays.

  Common features across most real-time operating systems include:

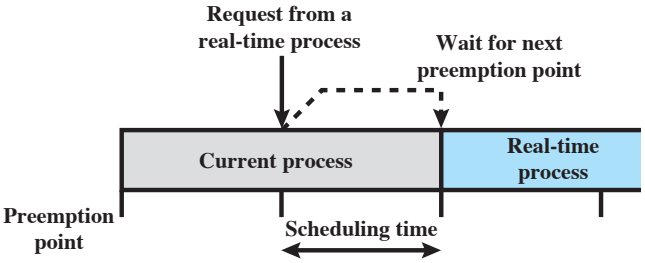
  - Stricter priority implementation than ordinary operating systems, with preemptive scheduling designed specifically for real-time requirements
  - Bounded and relatively short interrupt latency (time between device interrupt generation and service)
  - More precise and predictable timing characteristics compared to general-purpose operating systems



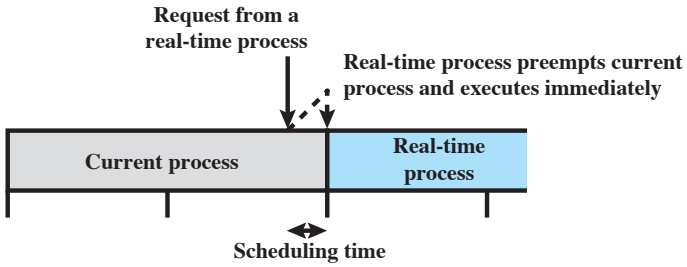
(a) Round-robin Preemptive Scheduler



(b) Priority-Driven Nonpreemptive Scheduler



(c) Priority-Driven Preemptive Scheduler on Preemption Points



(d) Immediate Preemptive Scheduler

Different scheduling approaches offer varying levels of responsiveness for real-time applications:

- The scheduling approach in scenario (a) introduces timing delays that are generally unacceptable for real-time applications.
- In scenario (b), execution of a slow, low-priority task could potentially cause delays of several seconds when a critical operation is needed. This approach fails to meet real-time requirements.
- Scenario (c) implements preemption points at regular intervals using clock-based interrupts. When a preemption point is reached, the currently running task can be preempted if a higher-priority task is waiting. This mechanism typically results in delays on the order of several milliseconds.
- The approach in scenario (d) allows the operating system to respond to interrupts almost immediately, with the exception of critical-code lockout sections. This method can reduce scheduling delays for real-time tasks to 100 μs or less.

11.6 Real-Time Scheduling

Based on these considerations, the authors identify the following classes of algorithms:

- **Static table-driven approaches:** These perform a static analysis of feasible schedules of dispatching. The result of the analysis is a schedule that determines, at run time, when a task must begin execution.
- **Static priority-driven preemptive approaches:** Again, a static analysis is performed, but no schedule is drawn up. Rather, the analysis is used to assign priorities to tasks, so that a traditional priority-driven preemptive scheduler can be used.
- **Dynamic planning-based approaches:** Feasibility is determined at run time (dynamically) rather than offline prior to the start of execution (statically). An arriving task is accepted for execution only if it is feasible to meet its time constraints. One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this task.

  Scheduling occurs after task arrival but before execution begins and attempts to create a new schedule incorporating both:

  - Previously scheduled tasks
  - The newly arrived task

  Schedule revision occurs only when:

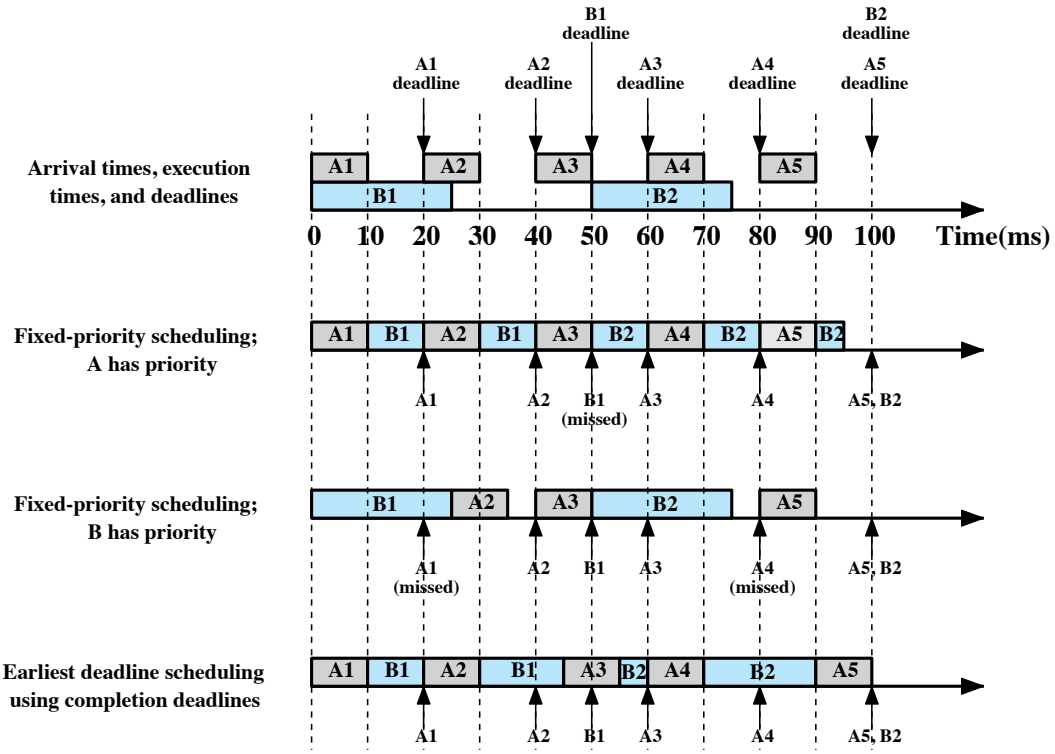
  - The new task's deadlines can be satisfied
  - No currently scheduled tasks will miss deadlines
- **Dynamic best effort approaches:** No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.
  - Commonly implemented in commercial real-time systems
  - Process:
    - System assigns priority based on task characteristics upon arrival
    - Typically employs deadline-based scheduling (e.g., earliest-deadline scheduling)
  - Characteristics:
    - Primarily designed for aperiodic tasks
    - Static scheduling analysis not possible
    - Timing constraint satisfaction unknown until deadline arrives or task completes
  - Trade-offs:
    - Advantage: Simple implementation
    - Disadvantage: Unpredictable timing constraint satisfaction

Contemporary real-time operating systems prioritize different aspects than conventional systems:

- Real-time applications typically prioritize task completion at optimal times—neither too early nor too late—despite
- Preemption design is crucial in real-time systems:

11.6.1 Example: Periodic Tasks with Completion Deadlines

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•



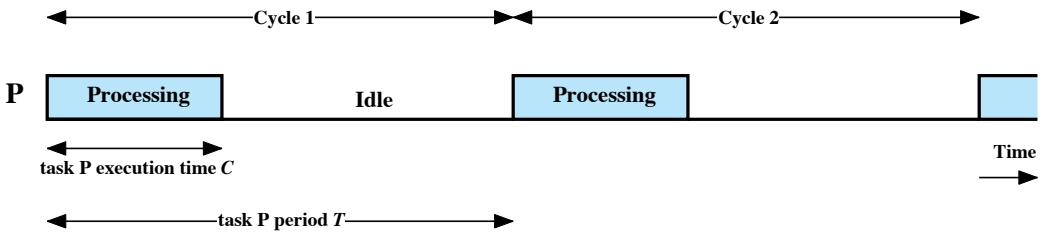
If task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume X to completion.

11.6.2 Example: Aperiodic Tasks with Starting Deadlines

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

Earliest deadline with unforced idle times, operates as follows: Always schedule the eligible task with the earliest deadline and let that task run to completion. An eligible task may not be ready, and this may result in the processor remaining idle even though there are ready tasks.

11.6.3 Rate Monotonic Scheduling (Preemptive)

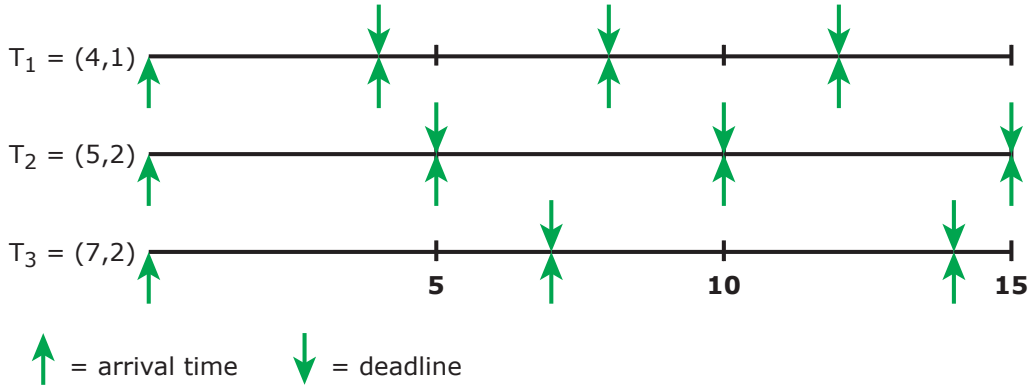


Rate monotonic scheduling (RMS) is one of the more promising methods for resolving multitask scheduling conflicts for periodic tasks.

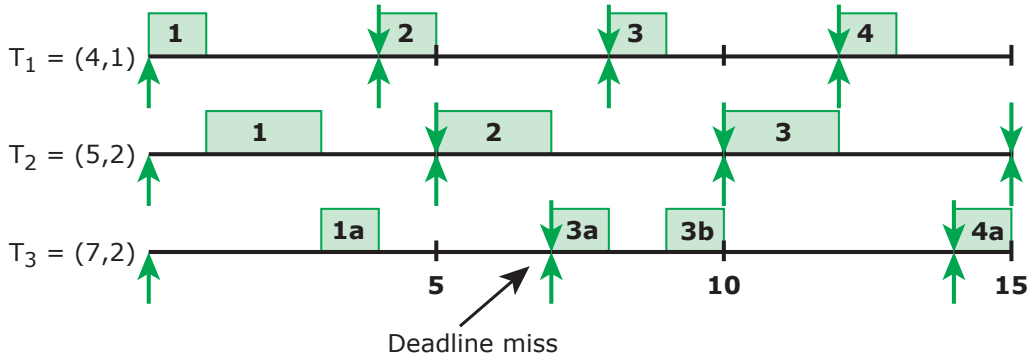
- RMS assigns priorities to tasks based on their periods
- The highest-priority task is the one with the shortest period
- The second highest-priority task has the second shortest period, and so on
- When multiple tasks are available for execution, the one with the shortest period is serviced first

The above figure illustrates the relevant parameters for periodic tasks:

- Task's period (T): The amount of time between the arrival of one instance of the task and the arrival of the next instance
- Task's rate: The inverse of its period (in seconds), measured in hertz (Hz)
  - Example: A task with a period of 50 ms occurs at a rate of 20 Hz
- Deadline: Typically, the end of a task's period is also the task's hard deadline, although some tasks may have earlier deadlines
- Execution (or computation) time (C): The amount of processing time required for each occurrence of the task
- Constraints:
  - In a uniprocessor system, the execution time must be less than the period ($C < T$)
  - If a periodic task always runs to completion, its processor utilization is $U = C/T$
  - Example: A task with period 80 ms and execution time 55 ms has processor utilization $U = 55/80 = 0.6875$



(a) Arrival times and deadines for task $T_i = (P_i, C_i)$; P_i = period, C_i = processing time



(b) Scheduling results

As can be seen, for task 3, the second instance is not executed because the deadline is missed. The third instance experiences a preemption but is still able to complete before the deadline.

One measure of the effectiveness of a periodic scheduling algorithm is whether or not it guarantees that all hard deadlines are met. Suppose we have n tasks, each with a fixed period and execution time. Then for it to be possible to meet all deadlines, the following inequality must hold:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq 1 \tag{24}$$

The sum of the processor utilizations of the individual tasks cannot exceed a value of 1, which corresponds to total utilization of the processor. The equation provides a bound on the number of tasks that a perfect scheduling algorithm can successfully schedule. For any particular algorithm, the bound may be lower. For RMS, it can be shown that the following inequality holds:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \tag{10.2}$$

The above bound for RMS is generated based on worst case, in practice, utilization as high as 90% is often achieved.

In a static priority assignment approach, one only needs to ensure that essential tasks have relatively high priorities. This can be done in RMS by structuring essential tasks to have short periods or by modifying the RMS priorities to account for essential tasks. With earliest-deadline scheduling, a periodic task's priority changes from one period to another. This makes it more difficult to ensure that essential tasks meet their deadlines.

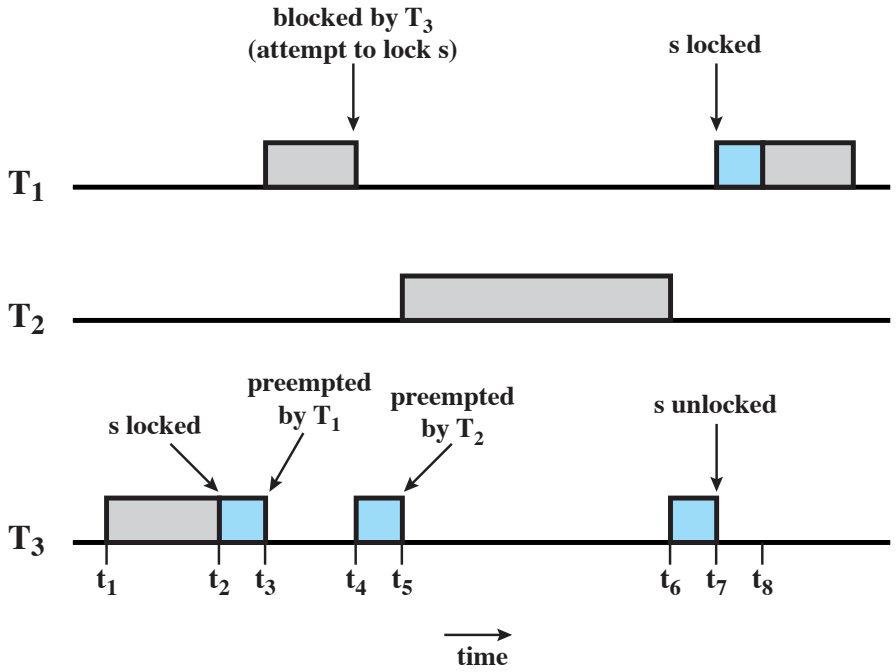
11.7 Priority inversion

Priority inversion is a phenomenon that can occur in any priority-based preemptive scheduling scheme but is particularly relevant in the context of real-time scheduling.

- **Basic Definition:** Priority inversion occurs when circumstances within the system force a higher-priority task to wait for a lower-priority task.

- **Resource Locking Scenario:** A common example occurs when a lower-priority task has locked a resource (such as a device or a binary semaphore) and a higher-priority task attempts to lock that same resource.
- **Task Blocking:** The higher-priority task will be put in a blocked state until the resource becomes available.
- **Limited Impact Case:** If the lower-priority task quickly finishes with the resource and releases it, the higher-priority task may resume promptly, potentially avoiding violations of real-time constraints.

A more serious condition is referred to as an unbounded priority inversion, in which the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks. The priority inversion experienced in the Pathfinder software was unbounded and serves as a good example of the phenomenon.



(a) Unbounded priority inversion

The Pathfinder software included the following three tasks, in decreasing order of priority:

- T1 : Periodically checks the health of the spacecraft systems and software
- T2 : Processes image data
- T3 : Performs an occasional test on equipment status

After T1 executes, it reinitializes a timer to its maximum value. If this timer ever expires, it is assumed that the integrity of the lander software has somehow been compromised. The processor is halted, all devices are reset, the software is completely reloaded, the spacecraft systems are tested, and the system starts over. This recovery sequence does not complete until the next day. T1 and T3 share a common data structure, protected by a binary semaphore *s*.

1. t1 : T3 begins executing.
2. t2 : T3 locks semaphore s and enters its critical section.
3. t3 : T1, which has a higher priority than T 3 , preempts T3 and begins executing.
4. t4 : T1 attempts to enter its critical section but is blocked because the semaphore is locked by T3 ; T3 resumes execution in its critical section.
5. t5 : T2 , which has a higher priority than T3 , preempts T3 and begins executing.
6. t6 : T2 is suspended for some reason unrelated to T1 and T3 ; T3 resumes.
7. t7 : T3 leaves its critical section and unlocks the semaphore. T1 preempts T3, locks the semaphore and enters its critical section.

In this set of circumstances, T1 must wait for both T3 and T2 to complete and fails to reset the timer before it expires.

11.8 Priority Inheritance

In practical systems, priority inheritance protocol is used to avoid unbounded priority inversion. The basic idea of priority inheritance is that a lower-priority task inherits the priority of any higher-priority task pending on a resource they share. This priority change takes place as soon as the higher-priority task blocks on the resource; it should end when the resource is released by the lower-priority task.

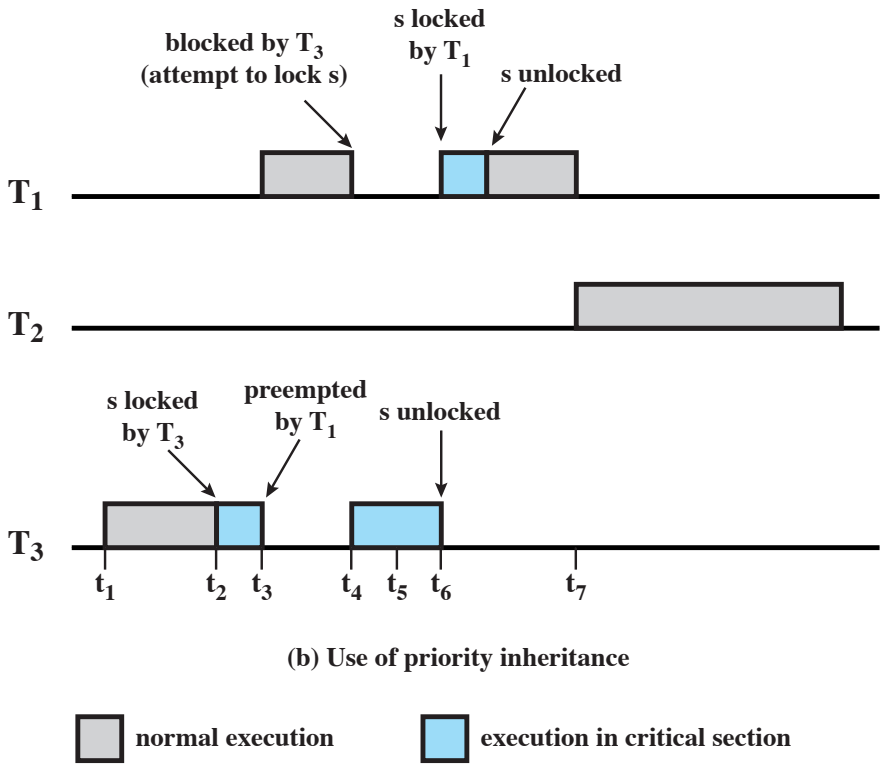


Figure 21: Caption

- t1 : T3 begins executing.
- t2 : T3 locks semaphore s and enters its critical section.
- t3 : T1 , which has a higher priority than T3 , preempts T3 and begins executing.
- t4 : T1 attempts to enter its critical section but is blocked because the semaphore is locked by T3 . T3 is immediately and temporarily assigned the same priority as T1 . T3 resumes execution in its critical section.
- t5 : T2 is ready to execute but, because T3 now has a higher priority, T2 is unable to preempt T3 .
- t6 : T3 leaves its critical section and unlocks the semaphore: its priority level is downgraded to its previous default level. T1 preempts T3 , locks the semaphore, and enters its critical section.
- t7 : T1 is suspended for some reason unrelated to T2 , and T2 begins executing.

12 IO Management And Disk Scheduling

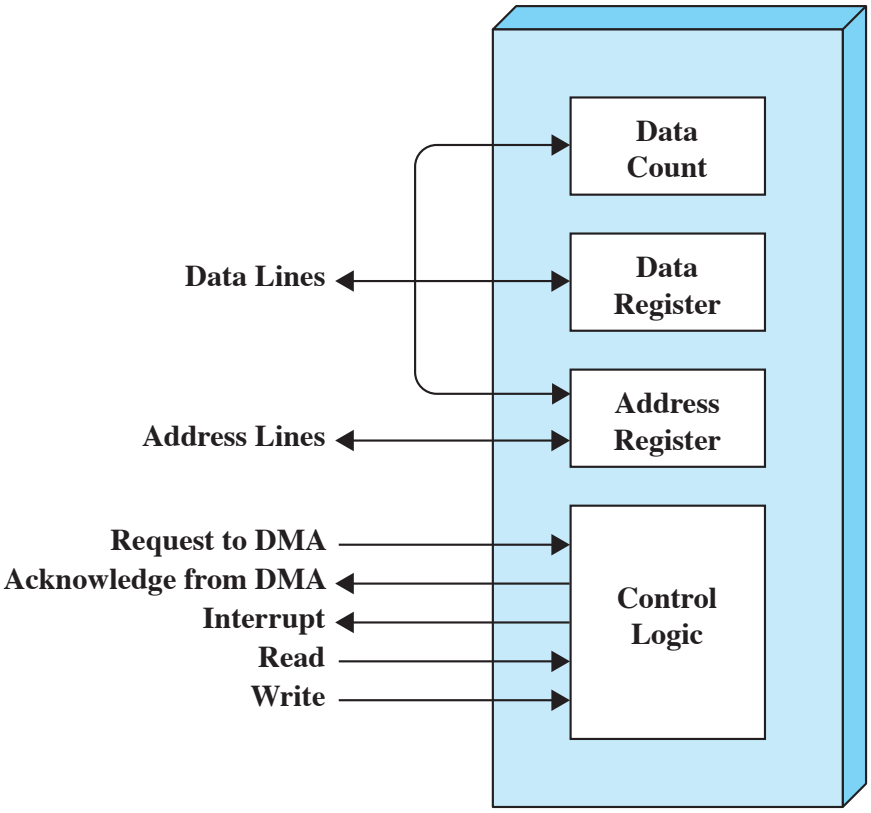
12.1 IO Devices

External devices that engage in I/O with computer systems can be grouped into three categories:

- **Human readable**: Suitable for communicating with the computer user. Examples include printers and terminals, the latter consisting of video display, keyboard, and perhaps other devices such as a mouse.
- **Machine readable**: Suitable for communicating with electronic equipment. Examples are disk drives, USB keys, sensors, controllers, and actuators.
- **Communication**: Suitable for communicating with remote devices. Examples are digital line drivers and modems.

Three techniques for performing I/O are:

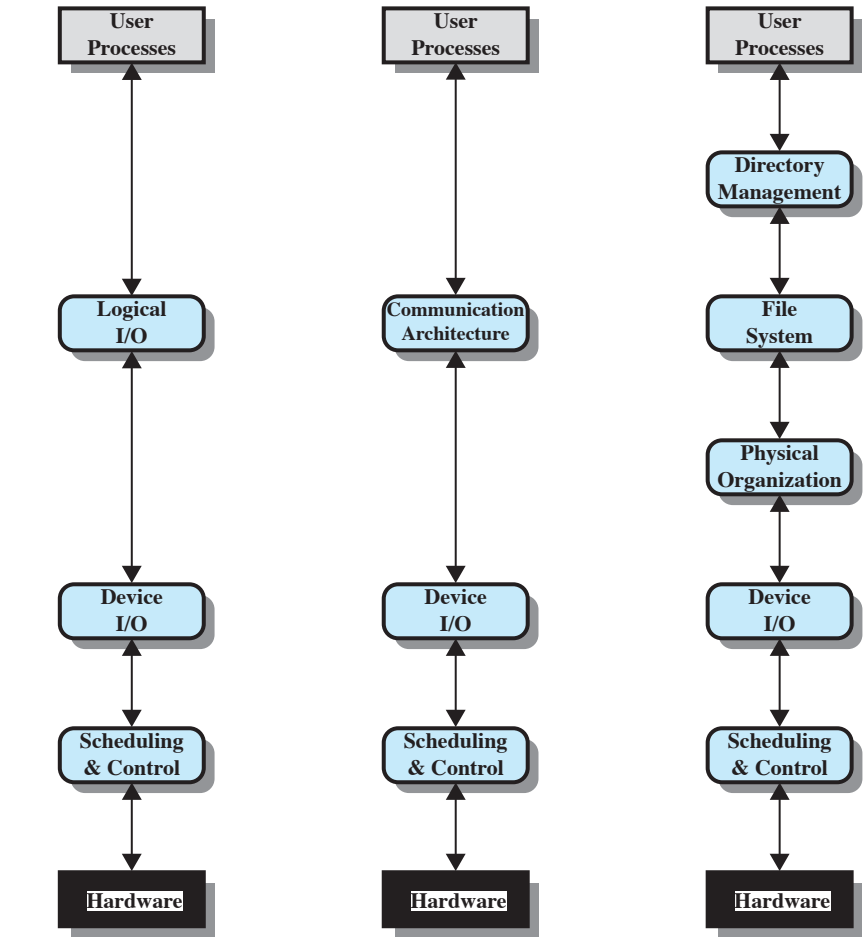
- **Programmed I/O**
  - The processor issues an I/O command on behalf of a process to an I/O module
  - That process then busy waits for the operation to be completed before proceeding
- **Interrupt-driven I/O**
  - The processor issues an I/O command on behalf of a process
  - If non-blocking – processor continues to execute instructions from the process that issued the I/O command
  - If blocking – the next instruction the processor executes is from the OS, which will put the current process in a blocked state and schedule another process
- **Direct Memory Access (DMA)**
  - A DMA module controls the exchange of data between main memory and an I/O module



The DMA unit possesses the capability to emulate processor behavior and assume control of the system bus similar to the processor. This functionality is essential for transferring data to and from memory via the system bus.

When the processor needs to read or write a data block, it delegates this task to the DMA module through the following procedure:

- The processor communicates with the DMA module, providing these essential parameters:
  - Operation type (read or write) via the dedicated control line between processor and DMA module
  - I/O device address transmitted through the data lines
  - Memory starting location (for reading from or writing to) sent via data lines and stored in the DMA module's address register
  - Word count for transfer, communicated through data lines and recorded in the data count register
- After command issuance, the processor resumes other operations, having delegated the I/O task to the DMA module
- The DMA module independently transfers the entire data block word by word directly between memory and the I/O device, bypassing the processor
- Upon completion of the data transfer, the DMA module signals the processor with an interrupt
- Processor involvement occurs only at the initiation and conclusion of the transfer process



(a) Local peripheral device **(b) Communications port** **(c) File system**

The figure presents three important logical structures which we will examine in sequence.

1. For simple **local peripheral devices** communicating with streams of bytes or records, the following layers are involved:
   - **Logical I/O:** Manages devices as logical resources using basic commands like open, close, read, and write.
   - **Device I/O:** Converts operations into hardware instructions and may use buffering for efficiency.
   - **Scheduling and Control:** Handles operation queuing, processes interrupts, and interfaces directly with hardware.
2. Communication devices use a similar structure but replace logical I/O with a communications architecture like TCP/IP.

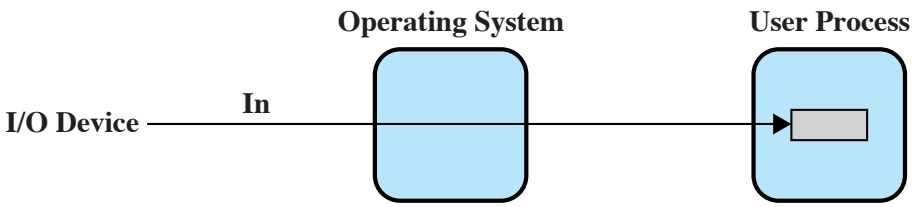
3. For storage devices with file systems, additional layers include:
    - **Directory Management:** Converts file names to identifiers and manages directory operations.
    - **File System:** Handles file structures and operations while managing access rights.
    - **Physical Organization:** Converts logical references to physical addresses and manages storage allocation.

12.2 Buffering

To avoid overheads and inefficiencies, it is sometimes convenient to perform input transfers in advance of requests being made, and to perform output transfers some time after the request is made.

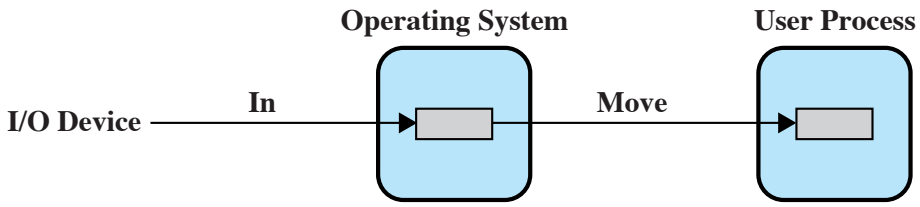
- **Block-oriented device**
    - Stores information in blocks that are usually of fixed size
    - Transfers are made one block at a time
    - Possible to reference data by its block number
    - Disks and USB keys are examples
- **Stream-oriented device**
    - Transfers data in and out as a stream of bytes
    - No block structure
    - Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are examples

Without a buffer, the OS directly accesses the device when it needs.



(a) No buffering

With single buffer, which is the simplest type of support that the operating system can provide, when a user process issues an I/O request, the OS assigns a buffer in the system portion of main memory to the operation.



(b) Single buffering

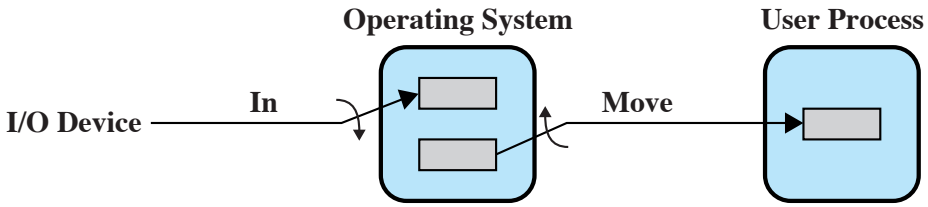
Input transfers in system buffering work as follows:

- Input transfers are made to the system buffer
- Reading ahead/anticipated input:
    - Is performed with the expectation that the block will eventually be needed
    - When the transfer completes, the process moves the block into user space
    - The process immediately requests another block afterward
- Advantages:

  - Generally provides a speedup compared to systems without buffering
  - User process can process one block of data while the next block is being read in
  - OS can swap the process out since input operations occur in system memory rather than user process memory
- Disadvantages:
  - Complicates the operating system's logic
  - Affects swapping logic as well

Stream-oriented I/O with single buffering can operate in two distinct modes:

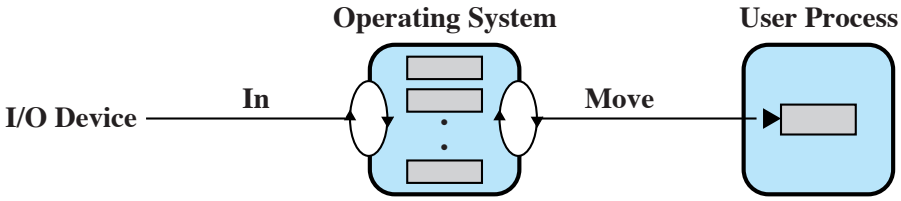
- **Line-at-a-time operation:**
  - Appropriate for scroll-mode terminals (sometimes called dumb terminals)
  - User input occurs one line at a time
  - Carriage return signals the end of a line
  - Terminal output similarly processes one line at a time
  - Line printers exemplify this type of device
  - The buffer holds a single line
  - During input, the user process is suspended while awaiting the entire line
  - For output, the user process can:
    * Place a line of output in the buffer and continue processing
    * Need not suspend unless a second line of output is ready before the buffer empties from the first operation
- **Byte-at-a-time operation:**
  - Used on forms-mode terminals where each keystroke is significant
  - Applied to various peripherals such as sensors and controllers
  - Interaction between the operating system and user process follows the producer/consumer model



(c) Double buffering

An improvement over single buffering can be achieved by assigning two system buffers to the operation. This technique, known as *double buffering* or *buffer swapping*, works as follows:

- A process transfers data to (or from) one buffer while the operating system simultaneously empties (or fills) the other
- This creates an efficient producer-consumer relationship between the process and I/O system



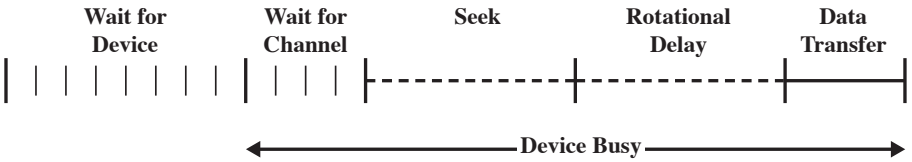
(d) Circular buffering

- When more than two buffers are used, the collection is referred to as a circular buffer.
- Each individual buffer constitutes one unit in the circular buffer.

12.3 Disk Performance

The actual details of disk I/O operation depend on the:

- Computer system
- Operating system
- Nature of the I/O channel and disk controller hardware

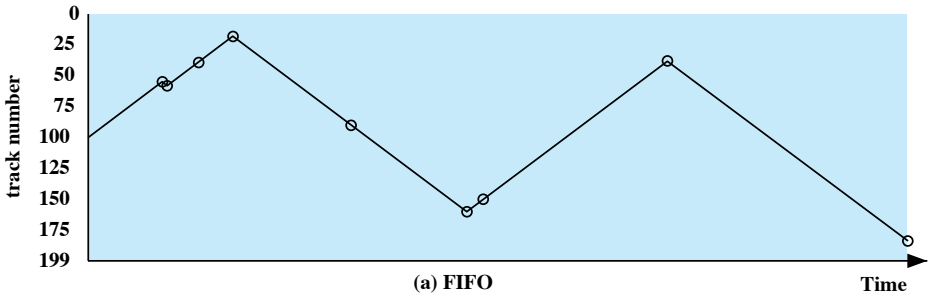


- When the disk drive is operating, the disk is rotating at constant speed To read or write the head must be positioned at the desired track and at the beginning of the desired sector on that track
- Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system
- On a movable-head system the time it takes to position the head at the track is known as **seek time**
- The time it takes for the beginning of the sector to reach the head is known as **rotational delay**
- The sum of the seek time and the rotational delay equals the **access time**

12.4 Disk Scheduling

12.4.1 First In First Out (FIFO)

FIFO (starting at track 100)	
Next track accessed	**Number of tracks traversed**
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
Average seek length	55.3



- Processes in sequential order
- Fair to all processes

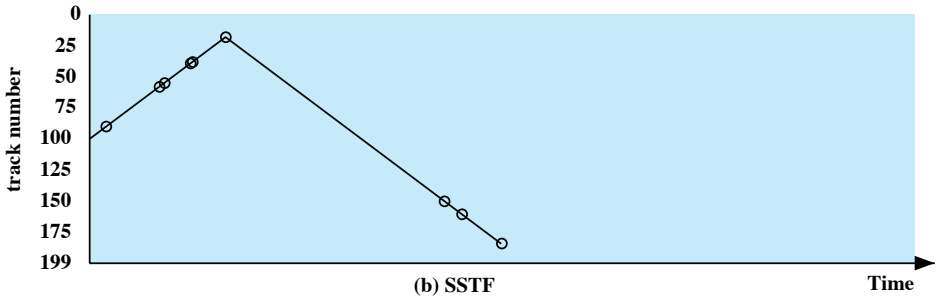
- Approximates random scheduling in performance if there are many processes competing for the disk

12.4.2 Priority (PRI)

- Control of the scheduling is outside the control of disk management software
- Goal is not to optimize disk utilization but to meet other objectives
- Short batch jobs and interactive jobs are given higher priority
- Provides good interactive response time
- Longer jobs may have to wait an excessively long time
- A poor policy for database systems (such a policy could lead to countermeasures on the part of users, who split their jobs into smaller pieces to beat the system.)

12.4.3 Shortest Service Time First (SSTF)

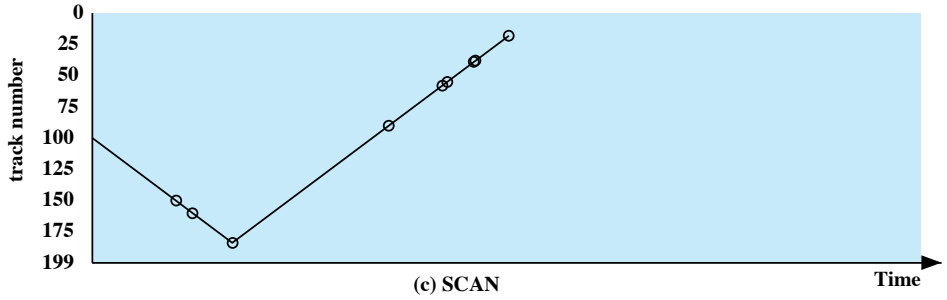
SSTF (starting at track 100)	
Next track accessed	**Number of tracks traversed**
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
Average seek length	27.5



- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Always choose the minimum seek time

12.4.4 SCAN

SCAN (starting at track 100,in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20
Average seek length	27.8

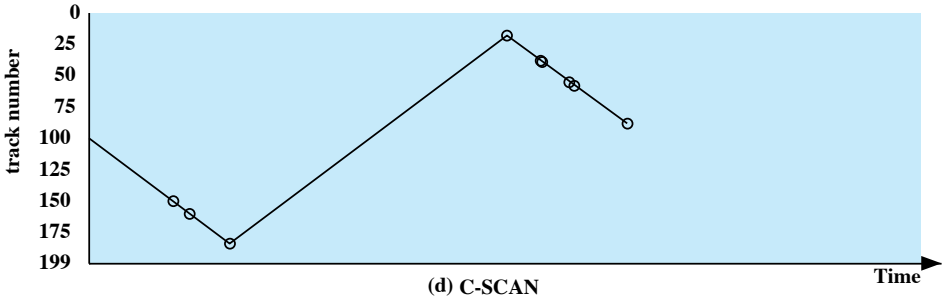


(c) SCAN

- Also known as the elevator algorithm
- Arm moves in one direction only: Satisfies all outstanding requests until it reaches the last track in that direction then the direction is reversed
- Favors jobs whose requests are for tracks nearest to both innermost and outermost tracks and favors the latest-arriving jobs

12.4.5 Circular SCAN

C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
Average seek length	35.8



(d) C-SCAN

- Restricts scanning to one direction only
- When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again

12.4.6 N-Step-SCAN

The SSTF, SCAN, and C-SCAN algorithms may experience periods where the disk arm remains stationary for extended durations. This "arm stickiness" can occur when:

- One or few processes have high access rates to a single track
- These processes can monopolize the entire device through repeated requests to that track
- High-density multisurface disks are particularly susceptible to this issue compared to lower-density disks or disks with only one or two surfaces

To mitigate the arm stickiness problem, segmentation of the disk request queue becomes necessary, with complete processing of one segment at a time. Two notable implementations of this approach are:

N-step-SCAN: This policy divides the disk request queue into subqueues of length N

- Subqueues are processed sequentially using the SCAN algorithm
- New requests arriving during processing must be directed to a different subqueue
- If fewer than N requests remain at scan completion, all are processed in the next scan
- Performance characteristics:
  - With large N values, N-step-SCAN approaches SCAN performance
  - When N = 1, FIFO is adopted

12.4.7 FSCAN

- Uses two subqueues
- When a scan begins, all of the requests are in one of the queues, with the other empty
- During scan, all new requests are put into the other queue
- Service of new requests is deferred until all of the old requests have been processed

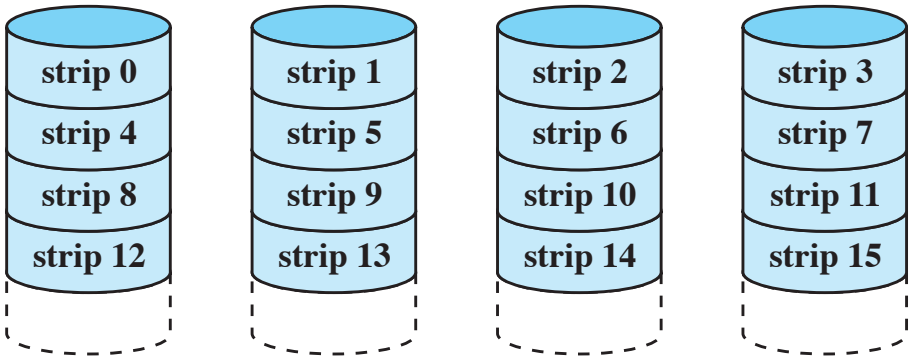
12.5 RAID

RAID (Redundant Array of Independent Disks) provides standardized approaches for organizing data across multiple disks, ensuring compatibility across platforms and operating systems. The RAID architecture includes levels 0 through 6, which share common characteristics but differ in implementation details:

- RAID functions as a set of physical disk drives that appear to the operating system as a single logical drive.
- Data distribution occurs across physical drives using a technique called striping.
- Most RAID levels utilize redundant disk capacity to store parity information, enabling data recovery after disk failures.

Note that RAID 0 and RAID 1 do not implement the parity-based redundancy described in the third characteristic.

12.5.1 RAID Level 0



(a) RAID 0 (non-redundant)

- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array: If two different I/O requests are pending for two different blocks of data, then there is a good chance that the requested blocks are on different disks. Thus, the two requests can be issued in parallel, reducing the I/O queuing time.
- Logical disk is divided into strips

Think of a logical disk as one large storage unit that users interact with. Here's how data is organized across physical disks:

- **Strip**: A small unit of data storage (could be a block, sector, etc.)
- **Striping**: The method of distributing strips across multiple disks

- **Stripe**: A complete set of strips that includes exactly one strip on each disk in the array

Imagine we have 4 physical disks in our RAID array. When data is written:

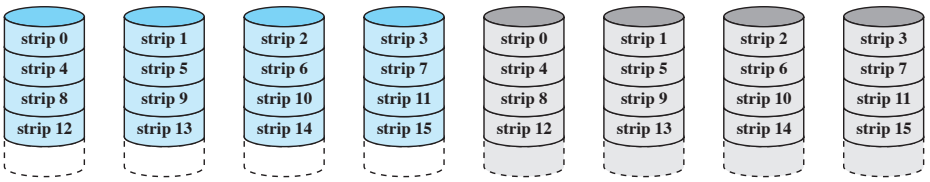
1. The first 4 logical strips (strips 0-3) are written as the first strip on each of the 4 disks, forming Stripe 0
2. The next 4 logical strips (strips 4-7) become the second strip on each disk, forming Stripe 1
3. This pattern continues for all data

```
Logical strips: 0 1 2 3 4 5 6 7 8 9 10 11 ...
                | | | | | | | | | | |  |
                v v v v v v v v v v v  v
Disk 0:     [0]   [4]   [8]    ...
Disk 1:     [1]   [5]   [9]    ...
Disk 2:     [2]   [6]   [10]   ...
Disk 3:     [3]   [7]   [11]   ...
            ---   ---   ---
          Stripe Stripe Stripe
            0     1     2
```

When your computer needs to read multiple consecutive strips (like strips 0-3), it can access all 4 disks simultaneously instead of waiting for a single disk to deliver all the data sequentially. This parallel access dramatically reduces I/O wait times, much like how multiple checkout lanes at a grocery store serve customers faster than a single lane.

This is why RAID improves performance - **it distributes the workload across multiple physical disks that can operate independently and simultaneously.**

12.5.2 RAID Level 1



(b) RAID 1 (mirrored)

RAID 1 achieves redundancy through data duplication, unlike RAID levels 2-6 which use parity calculations. In RAID 1, each logical strip is mapped to two separate physical disks, creating mirror copies of all data. While commonly implemented with data striping, RAID 1 can also function without it.

Advantages:

- **Read Performance:** Read requests can be serviced by either of the two disks containing the requested data, using whichever involves minimum seek time plus rotational latency.
- **Write Efficiency:** Write requests update both corresponding strips in parallel. Performance depends on the slower of the two writes, but there is no "write penalty" as with parity-based RAID levels.
- **Simple Recovery:** When a drive fails, data remains accessible from the second drive.

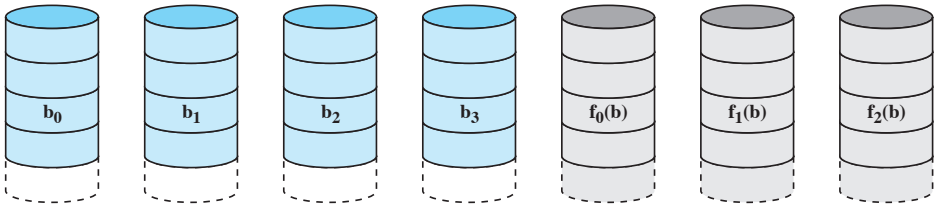
Disadvantages:

- **Cost:** RAID 1 requires twice the disk space of the logical disk it supports, making it suitable primarily for system software, critical data, and real-time backup applications.

Performance Characteristics:

- In transaction-oriented environments with mostly read requests, RAID 1 performance can approach double that of RAID 0.
- With substantial write requests, performance may not significantly exceed RAID 0.
- May improve performance for data transfer intensive applications with high percentage of reads, especially when read requests can be split between both disk members.

12.5.3 RAID Level 2



(c) RAID 2 (redundancy through Hamming code)

RAID levels 2 and 3 utilize parallel access arrays where:

- All disks participate simultaneously in every I/O request
- Drive spindles are synchronized, keeping disk heads aligned
- Data is striped in very small units (often single bytes or words)
- As in the other RAID schemes, data striping is used.
- In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word.
- With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks. Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.

RAID 2 would only be an effective choice in an environment in which many disk errors occur. Given the high reliability of individual disks and disk drives, RAID 2 is overkill and is not implemented.

Hamming code is a family of linear error-correcting codes developed by Richard Hamming in the 1940s. These codes are designed to detect and correct bit errors that can occur during data transmission or storage.

- Hamming codes add redundant parity bits to data bits in specific positions
- Each parity bit checks the correctness of particular data bits based on their positions
- The positions of parity bits are powers of 2 (1, 2, 4, 8, etc.)
- All other positions contain data bits

Error Detection Mechanism:

- Each parity bit monitors a specific subset of data bits
- The subsets are determined by the binary representation of bit positions
- Parity bit at position 2^k checks all bits whose positions have the k-th bit set to 1
- For example:
  - Parity bit at position 1 (2^0) checks bits 1, 3, 5, 7, 9, ...
  - Parity bit at position 2 (2^1) checks bits 2, 3, 6, 7, 10, 11, ...
  - Parity bit at position 4 (2^2) checks bits 4, 5, 6, 7, 12, 13, ...

Error Correction Process:

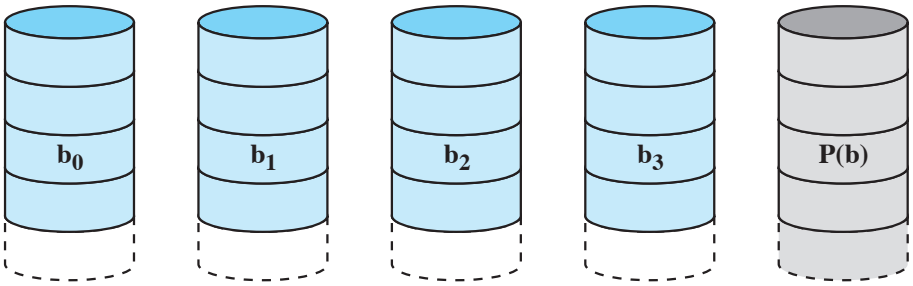
- When reading data, parity bits are recalculated
- If all parity checks pass, data is assumed error-free
- If one or more parity checks fail, their positions identify the error location
- The binary sum of failed parity positions indicates the position of the erroneous bit
- The erroneous bit can then be flipped to correct the error

Below is an example:

- The Hamming(7,4) code encodes 4 data bits using 7 total bits (3 parity bits + 4 data bits) $\oplus$ is XOR operation.
- **Bit positions:**
  - Position 1: Parity bit (P1)
  - Position 2: Parity bit (P2)

  - Position 3: Data bit (D1)
  - Position 4: Parity bit (P3)
  - Position 5: Data bit (D2)
  - Position 6: Data bit (D3)
  - Position 7: Data bit (D4)
- **Encoding process:** Suppose we want to encode the data bits 1011
  - Place data bits in positions 3, 5, 6, 7: [-, -, 1, -, 0, 1, 1]
  - Calculate parity bit P1 (position 1):
    - P1 checks positions 1, 3, 5, 7 (odd-numbered positions)
    - Data in these positions: [P1, 1, 0, 1]
    - For even parity: $P1 = 1 \oplus 0 \oplus 1 = 0$
    - Updated code: [0, -, 1, -, 0, 1, 1]
  - Calculate parity bit P2 (position 2):
    - P2 checks positions 2, 3, 6, 7 (positions with '1' in the second bit of binary representation)
    - Data in these positions: [P2, 1, 1, 1]
    - For even parity: $P2 = 1 \oplus 1 \oplus 1 = 1$
    - Updated code: [0, 1, 1, -, 0, 1, 1]
  - Calculate parity bit P3 (position 4):
    - P3 checks positions 4, 5, 6, 7 (positions with '1' in the third bit of binary representation)
    - Data in these positions: [P3, 0, 1, 1]
    - For even parity: $P3 = 0 \oplus 1 \oplus 1 = 0$
    - Final encoded message: [0, 1, 1, 0, 0, 1, 1]
- **Error detection example:** Suppose bit 5 gets flipped during transmission
  - Received message: [0, 1, 1, 0, **1**, 1, 1] (bit 5 changed from 0 to 1)
  - Recalculate parity bits:
    - P1 should check [P1, 1, **1**, 1] → Expected: $1 \oplus 1 \oplus 1 = 1$, Actual: 0 → Error detected
    - P2 should check [P2, 1, 1, 1] → Expected: $1 \oplus 1 \oplus 1 = 1$, Actual: 1 → No error detected
    - P3 should check [P3, **1**, 1, 1] → Expected: $1 \oplus 1 \oplus 1 = 1$, Actual: 0 → Error detected
  - Error location calculation:
    - Failed parity checks: P1 (position 1) and P3 (position 4)
    - Binary sum: 001 + 100 = 101 (binary) = 5 (decimal)
    - This indicates the error is at position 5
    - Correcting action: Flip bit 5 back from 1 to 0
    - Restored message: [0, 1, 1, 0, 0, 1, 1]

12.5.4 RAID Level 3



(d) RAID 3 (bit-interleaved parity)

RAID 3 is organized similarly to RAID 2 but with some key differences:

- RAID 3 requires only a single redundant disk, regardless of array size
- It employs parallel access with data distributed in small strips
- Instead of an error-correcting code, it uses a simple parity bit computed for corresponding bit positions across all data disks

Let's consider a simple RAID 3 system with 4 data disks and 1 parity disk:

Assume we want to store the following byte of data across our 4 data disks:

- Disk 1: 1011
- Disk 2: 0110
- Disk 3: 1100
- Disk 4: 0101

To calculate the parity that gets stored on the parity disk, we perform a bit-by-bit XOR operation across all data disks:

```
Disk 1:   1 0 1 1
Disk 2:   0 1 1 0
Disk 3:   1 1 0 0
Disk 4:   0 1 0 1
-----------------
Parity:   0 1 0 0  (XOR of all bits in each column)
```

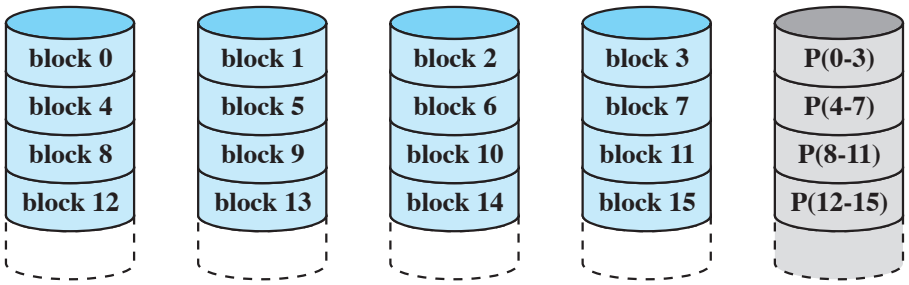
Now let's say Disk 3 fails. Here's how we would recover its data:

1. We know the data on the working disks:
   - Disk 1: 1011
   - Disk 2: 0110
   - Disk 4: 0101
   - Parity Disk: 0100
2. To recover Disk 3's data, we XOR all remaining disks including the parity disk:

```
Disk 1:         1 0 1 1
Disk 2:         0 1 1 0
Disk 4:         0 1 0 1
Parity Disk:    0 1 0 0
-----------------
Recovered D3:   1 1 0 0  (XOR of all bits in each column)
```

And we've successfully recovered the original data from Disk 3 (1100)!

12.5.5 RAID Level 4

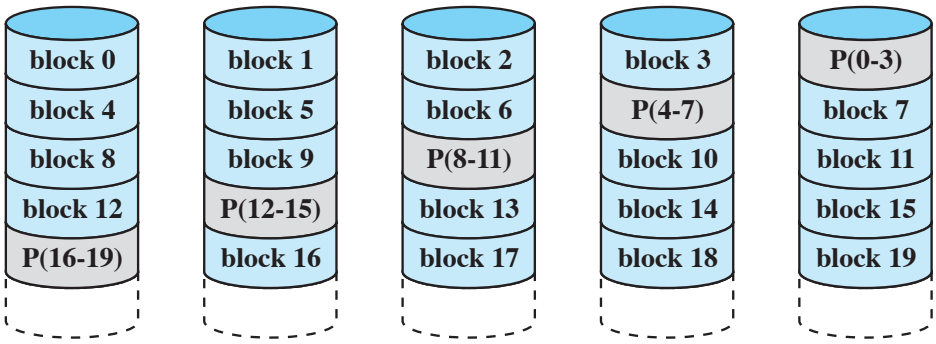


(e) RAID 4 (block-level parity)

RAID levels 4-6 employ an independent access technique, with the following characteristics:

- Member disks operate independently, allowing parallel satisfaction of separate I/O requests
- More suitable for applications requiring high I/O request rates
- Less suited for applications demanding high data transfer rates
- Uses data striping with relatively large strips
- Parity information is calculated across corresponding strips on data disks
- Small write operations incur a performance penalty:
  - Array must update both user data and corresponding parity bits
  - Process requires reading old user strip and old parity strip
  - Then updating both with new data and newly calculated parity
  - Results in two reads and two writes per strip write
- Large write operations spanning all disks are more efficient:
  - Parity computed directly from new data bits
  - Parity disk updated in parallel with data disks
  - No extra reads or writes required
- Potential bottleneck:
  - All write operations must involve the parity disk
  - Parity disk can become a performance bottleneck

12.5.6 RAID Level 5



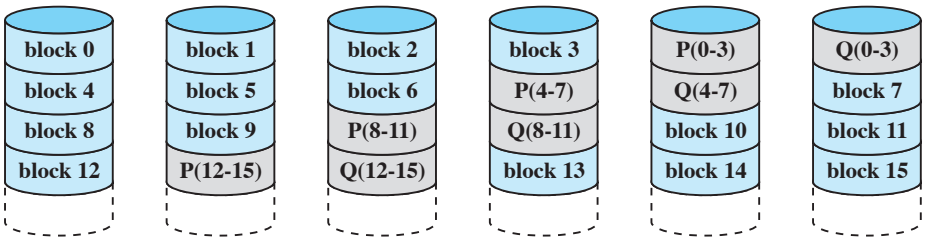
(f) RAID 5 (block-level distributed parity)

RAID 5 is similar to RAID 4 but with an important improvement in parity distribution:

- Unlike RAID 4, RAID 5 distributes parity strips across all disks in the array
- Typically uses a round-robin allocation scheme (as shown in Figure 11.8f)

- For an n-disk array, the parity strip rotates through all disks for every n stripes, then repeats
- This distribution eliminates the I/O bottleneck found on the single parity disk in RAID 4
- RAID 5 maintains data integrity even if any one disk fails

12.5.7 RAID Level 6



(g) RAID 6 (dual redundancy)

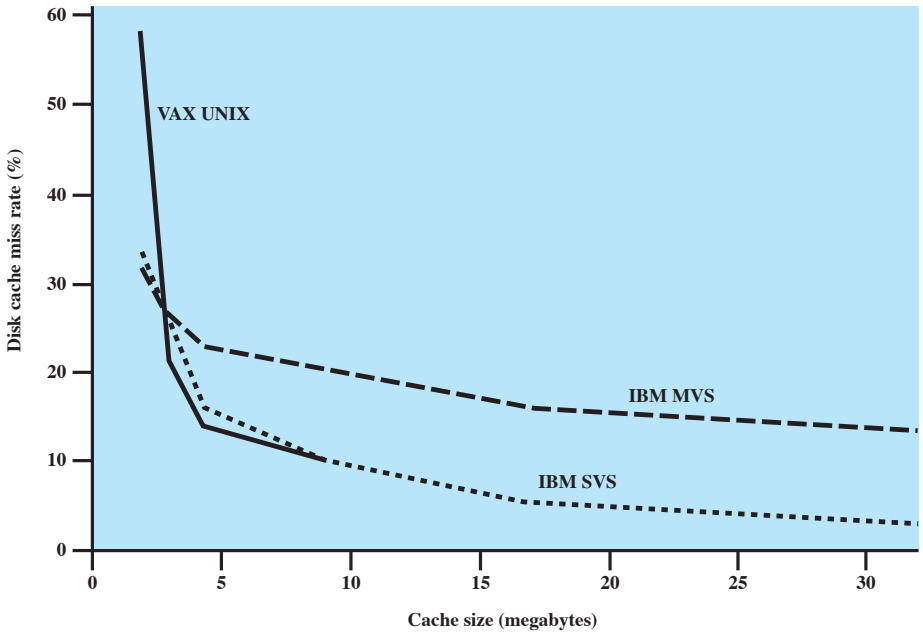
- RAID 6 uses two different parity calculations stored on separate disks
- A RAID 6 array requires $N + 2$ disks (where N is the number of disks needed for user data)
- The system employs two check algorithms:
  - P: The standard XOR calculation used in RAID 4/5
  - Q: An independent data check algorithm
- This design allows data recovery even if two data disks fail simultaneously
- **Primary advantage:** Extremely high data availability
  - Three disks would need to fail within the MTTR period to cause data loss
- **Primary disadvantage:** Substantial write performance penalty
  - Each write affects two parity blocks
  - Benchmarks show over 30% drop in write performance compared to RAID 5
  - Read performance is comparable to RAID 5

Among the seven described RAID levels, only four see widespread use: RAID 0, RAID 1, RAID 5, and RAID 6.

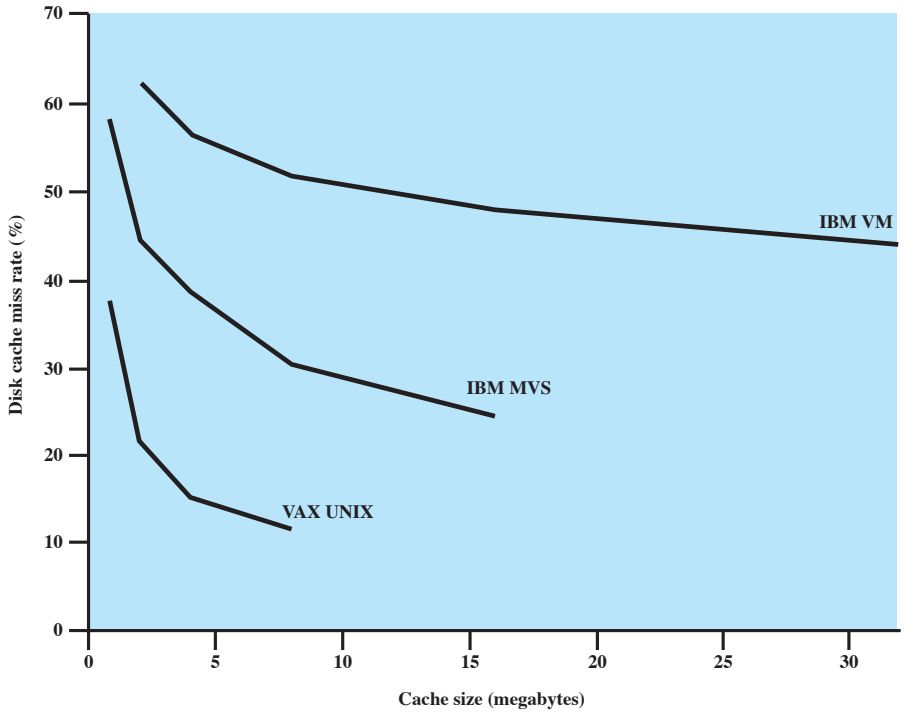
12.6 Disk Cache

- Cache memory typically refers to small, fast memory located between main memory and the processor
- A disk cache applies similar principles but functions as a buffer in main memory for disk sectors
- The disk cache contains copies of selected disk sectors
- When an I/O request occurs:
  - The system checks if the requested sector is in the disk cache
  - If present (cache hit), the request is satisfied through the cache
  - If absent (cache miss), the sector is read from disk into the cache
- Due to locality of reference, when data is fetched into the cache for one request, future references to the same data are likely

Several key design issues arise in disk cache implementations:



- **Data Delivery Methods:** When satisfying I/O requests from the disk cache, two approaches exist:
  - Transfer the data block from disk cache to user process memory
  - Use shared memory and pass a pointer to the cache slot (avoiding memory-to-memory transfers and enabling shared access via readers/writers model)
- **Replacement Strategies:** When bringing new sectors into a full cache, existing blocks must be replaced.
  - The most common algorithm is least recently used (LRU)
  - LRU replaces the block unused for the longest time
  - Conceptually, cache operates as a stack:
    * Referenced blocks move to the top
    * New blocks enter at the top
    * Replacements occur from the bottom
    * Implementation uses pointer stack rather than actually moving memory blocks



- **Least Frequently Used (LFU):** Replaces the block with the fewest references. Implementation:
  - Associate a counter with each block
  - New blocks receive count = 1
  - Increment count by 1 with each reference
  - Select block with smallest count for replacement
- **Intuitive advantage:** LFU appears better than LRU because it uses more relevant information in the selection process.
- **Key limitation:** LFU has a significant problem:
  - Some blocks experience short intervals of repeated references (locality)
  - This builds high reference counts despite overall infrequent use
  - After these intervals, the high count becomes misleading
  - The count no longer reflects probability of future reference
  - Thus, paradoxically, locality effects can cause LFU to make poor replacement choices

13 File Management

File systems are crucial components of operating systems from a user's perspective. They provide abstractions for secondary storage and allow users to create **files** with several important properties:

- **Persistence:** Files remain stored on secondary storage devices even after users log off
- **Sharing capabilities:** Files have names and access permissions that enable controlled sharing between processes
- **Internal structure:** Files can have formats suitable for specific applications
- **Organizational structure:** Files can be arranged in hierarchical or more complex structures to show relationships between them

File systems provide storage for organized data and functions for file manipulation. Common operations include:

- **Create:** Define a new file and position it within the file structure
- **Delete:** Remove and destroy a file from the structure
- **Open:** Declare a file as "opened" by a process, enabling operations on it
- **Close:** End a process's access to a file until reopened
- **Read:** Access all or part of a file's data
- **Write:** Update a file by adding new data or modifying existing content

File systems maintain attributes for each file, including owner, creation time, last modification time, and access privileges.

Four terms are commonly used when discussing files:

- **Field:** The basic element of data.
  - Basic element of data
  - Contains a single value
  - Fixed or variable length
- **Record:** A collection of related fields treated as a unit by an application.
  - Example: employee record contains fields like name, SSN, job classification
  - Can be fixed or variable length
- **File:** A collection of similar records.
  - Referenced by name
  - Can be created and deleted
  - Access control typically applied at file level
  - Some systems structure files as collections of fields without records
- **Database:** A collection of related data.
  - Relationships between data elements are explicit
  - Designed for multiple applications
  - May contain all information related to an organization or project
  - Consists of one or more types of files
  - Often managed by a separate database management system

File management systems are system software that provide file-related services to users and applications. These systems typically serve as the exclusive interface between users/applications and files. This system eliminates the need for custom file-handling code in each application. It provides consistent and well-defined control over system files.

Key objectives of file management systems include:

- Meeting user data management needs (storage and operations)
- Ensuring data validity

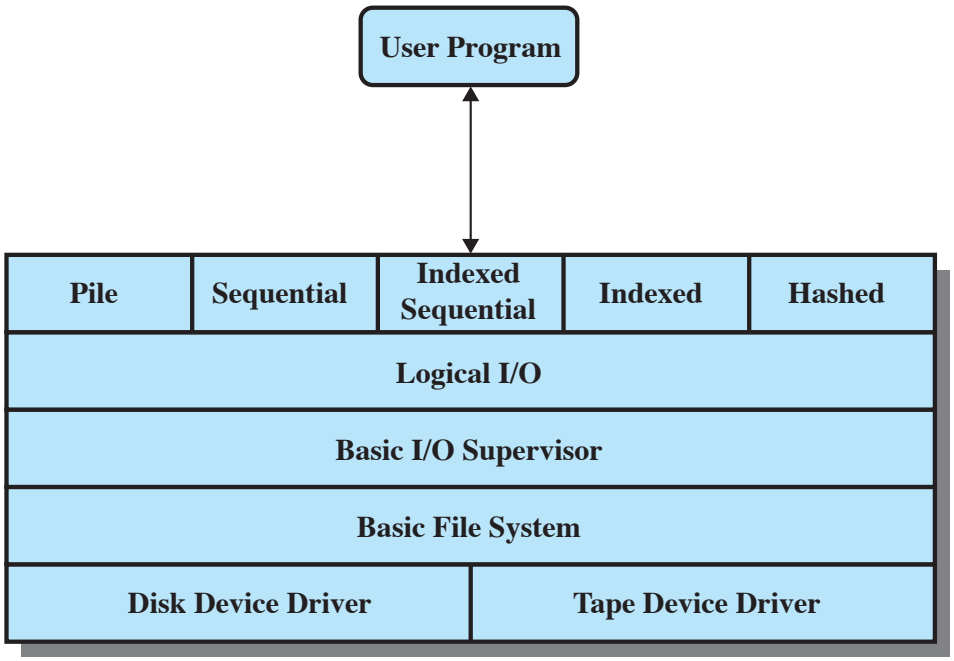
- Optimizing performance (system throughput and user response time)
- Supporting various storage device types
- Preventing data loss or corruption
- Providing standardized I/O interface routines
- Supporting multiple users in multi-user systems

Meeting user requirements is essential for computer systems, particularly for interactive, general-purpose systems. A minimal set of requirements includes:

- Each user should be able to create, delete, read, write, and modify files
- Each user may have controlled access to other users' files
- Each user may control what types of accesses are allowed to the user's files
- Each user should be able to move data between files
- Each user should be able to back up and recover files in case of damage
- Each user should be able to access files by name rather than by numeric identifier

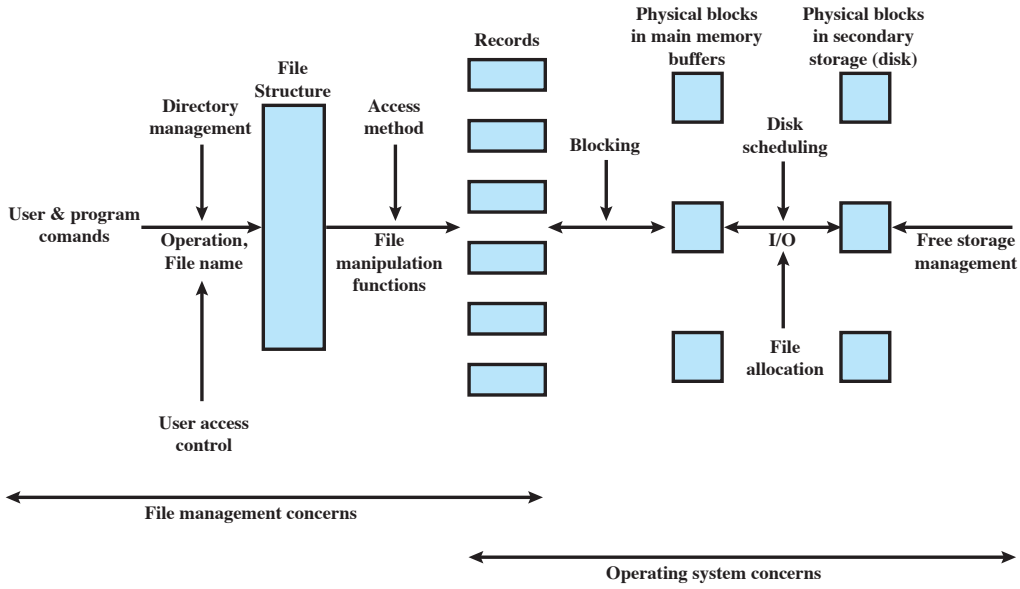
These objectives and requirements should guide the design of file management systems

Components of a basic file management system:



- **Device drivers** communicate directly with peripheral devices or their controllers or channels.
  - Initiating I/O operations on devices
  - Processing the completion of I/O requests
- **The basic file system**, also known as the physical I/O level, serves as the primary interface with the computer's external environment. Its key characteristics:
  - Manages data exchange with secondary storage devices (disk or tape systems)
  - Handles blocks of data without understanding their content or file structure
  - Responsible for block placement on storage devices
  - Manages buffering of blocks in main memory
- **The basic I/O supervisor** is the component responsible for managing all file input/output operations from initiation to termination. Its primary functions include:
  - Maintaining control structures for device I/O, scheduling, and file status monitoring

  - Selecting appropriate devices for file I/O based on the specific file requirements
  - Optimizing performance through strategic scheduling of disk and tape access
  - Assigning I/O buffers and allocating secondary memory resources
- **Logical I/O** provides an interface for users and applications to work with file records rather than data blocks. This module:
  - Handles record-level operations while the basic file system manages block-level operations
  - Delivers general-purpose record I/O capability
  - Maintains essential file metadata
- The file system level closest to the user is the **access method**. It provides a standard interface between applications and the underlying file systems and storage devices. Access methods represent different file structures and data access techniques.



The file system performs several key functions:

- **User Interface**: Processes commands for creating, deleting, and operating on files
- **File Management**:
  - Identifies and locates files using directories
  - Enforces user access control
  - Manages file structure (e.g., sequential organization)
- **Storage Management**:
  - Translates between record-level operations and block-level I/O
  - Allocates files to storage blocks
  - Manages free storage space
  - Schedules block I/O requests

The division between file management system utilities and operating system responsibilities is somewhat arbitrary, with record processing often serving as the interface between them.

13.1 FIle Organization and Access

File organization refers to the logical structuring of records based on access methods, distinct from physical organization which depends on blocking strategy and file allocation.

When selecting a file organization, several key criteria should be considered:

- Short access time
- Ease of update
- Economy of storage
- Simple maintenance
- Reliability

The priority of these criteria varies by application requirements. For batch processing where all records are accessed simultaneously, rapid retrieval of individual records becomes less important. Similarly, update capability is irrelevant for read-only media like CD-ROM.

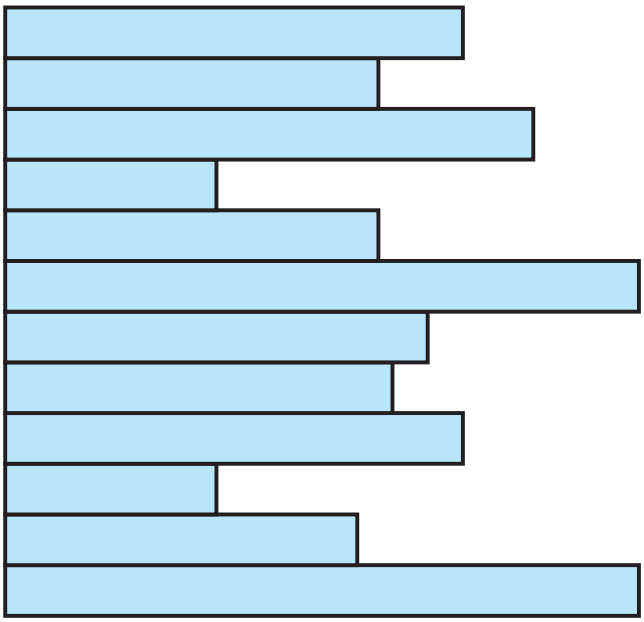
These criteria may conflict with each other. For instance, minimizing data redundancy improves storage economy, but redundancy (such as indexes) often enhances access speed.

File systems can be organized in multiple ways. Below are **five fundamental organizations that form the basis of most file systems**:

- **The pile:** A collection of records with no particular ordering.
- **The sequential file:** Records are stored in sequence based on a key field value.
- **The indexed sequential file:** Combines sequential access with indexes for faster direct access.
- **The indexed file:** Uses indexes to locate records without requiring sequential ordering.
- **The direct, or hashed, file:** Uses hash functions to determine record location from key values.

Most actual file systems either fall into one of these categories or implement combinations of these fundamental organizations.

13.1.1 The Pile



Variable-length records
Variable set of fields
Chronological order

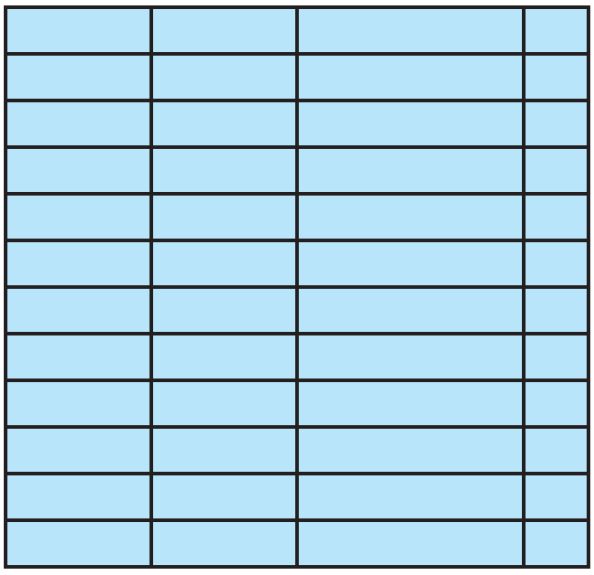
(a) Pile File

The pile represents the simplest form of file organization:

- Data are collected sequentially in arrival order

- Each record consists of one data burst
- Primary purpose is data accumulation and storage
- Records may have:
  - Different fields
  - Similar fields in different orders
- Each field should be self-describing, including field name and value
- Field length must be indicated by:
  - Delimiters
  - Explicit subfield inclusion
  - Known default for that field type
- Record access requires exhaustive search due to lack of structure
- Finding a specific record with a particular field value requires examining each record
- Finding all records with a particular field requires searching the entire file
- Advantages:
  - Efficient space utilization for variable-sized data
  - Adequate for exhaustive searches
  - Easy to update
- Limitations:
  - Unsuitable for most applications beyond simple storage

13.1.2 The Sequential File



Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

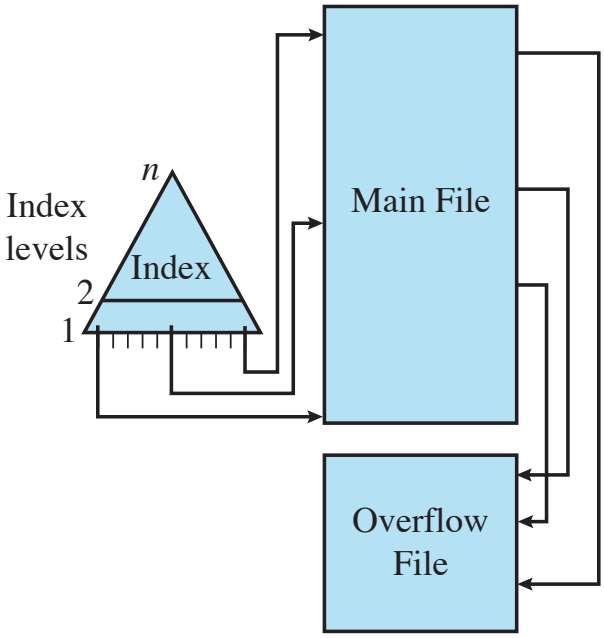
(b) Sequential File

Sequential files are the most common file structure type with the following characteristics:

- Records have fixed formats with uniform length

- Each record contains the same number of fixed-length fields in specific order
- Only field values need storage since field names and lengths are part of the file structure
- Records are identified by a key field
- Records are stored in key sequence (alphabetical for text, numerical for numbers)
- Optimal for batch applications processing all records (e.g., billing, payroll)
- Only file organization easily stored on both tape and disk
- Poor performance for interactive applications (queries/updates of individual records)
  - Access requires sequential search for key match
  - Large files cause significant processing delays for individual record access

13.1.3 Indexed Sequential File



(c) Indexed Sequential File

The indexed sequential file enhances sequential files by preserving their key ordering while adding:

- An index for efficient random access
- An overflow file for handling insertions

A simple indexed sequential structure uses a single-level index. Each index record contains:

- A key field (matching the main file)
- A pointer to the main file

To find a record, the system:

- Searches the index for the highest key value that is equal to or precedes the desired key value
- Follows the pointer to the main file
- Continues searching from that location

For a file with 1 million records:

- Sequential search: 500,000 accesses (average)
- With 1,000-entry index: 1,000 accesses (500 index + 500 main file)

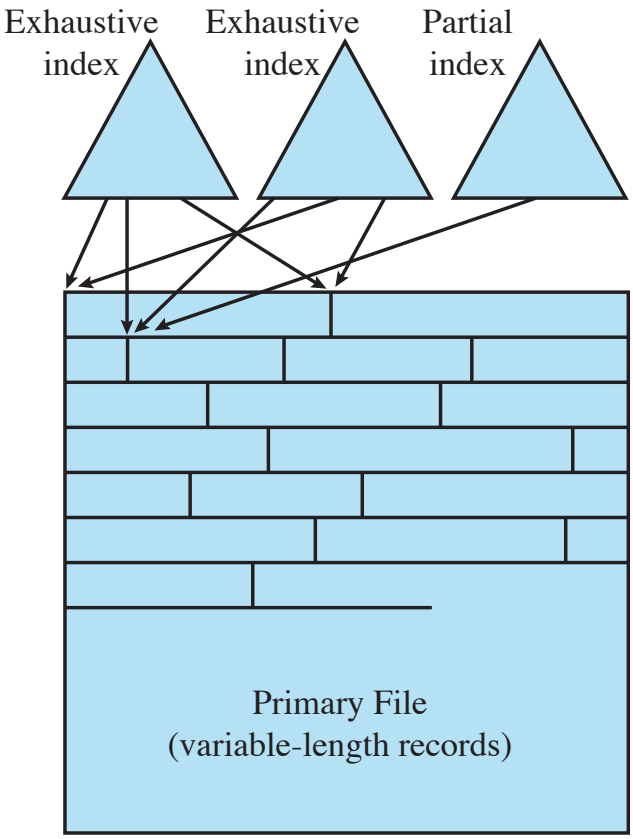
New records are managed through:

- Addition to the overflow file
- Updating pointers in the logically preceding record
- Periodic batch merging with the main file

For greater efficiency:

- Create multiple levels of indices
- Example (1 million record file):
  - Lower-level index: 10,000 entries
  - Higher-level index: 100 entries
  - Search path: higher index (50) → lower index (50) → main file (50)
  - Total accesses reduced to 150 (versus 500,000 originally)

13.1.4 The Indexed File



(d) Indexed File

- Indexed sequential files maintain a key limitation: efficient processing is restricted to a single field.
- When searching based on attributes other than the key field, sequential files are inadequate.
- Applications requiring flexible searches across various attributes need an alternative approach.
- To achieve flexibility, multiple indexes are needed (one per searchable field).
- This structure abandons the concepts of sequentiality and single keys.
- Records are accessed exclusively through indexes.
- Record placement has no restrictions if at least one index points to it.
- Variable-length records become feasible.
- **Exhaustive index:** Contains one entry for every record in the main file.

- The index itself is organized sequentially for efficient searching.
- **Partial index:** Contains entries only for records where the specific field exists.
- With variable-length records, some records may lack certain fields.
- Adding new records requires updating all index files.
- Indexed files are primarily used where information timeliness is critical.
- They serve applications where data are rarely processed exhaustively.
- Examples include airline reservation systems and inventory control systems.

13.1.5 An Example:Library Book Management System

- **File:** The entire database containing all information about books in the library.
- **Record:** Information about a single book, containing multiple attributes.
- **Fields:** Individual attributes within each record, such as:
  - ISBN (could be the primary key field)
  - Title
  - Author
  - Publication year
  - Genre
  - Current status (checked out, available, etc.)

Assume our library has 1,000 books (1,000 records in our file). Here's how different indexes would work:

Primary Index (by ISBN)

- This is an exhaustive index organized sequentially by ISBN numbers
- Every book has an entry in this index
- Example entry: ISBN 978-0134670949 → points to record #342 in the main file

Partial Index (by Genre)

- Only includes entries for records where the genre field exists
- Organized by genre categories
- Example entries:
  - Science Fiction → points to records #25, #103, #245, #678, ...
  - Mystery → points to records #12, #56, #198, #421, ...

Partial Index (by Author)

- Organized alphabetically by author names
- Example entries:
  - "King, Stephen" → points to records #78, #145, #356, ...
  - "Rowling, J.K." → points to records #45, #46, #47, #48, ...

1. If you want to find a book by its ISBN (978-0134670949):
   - Search the ISBN index (quickly, as it's sequential)
   - Follow the pointer to record #342
   - Retrieve the complete book information
2. If you want all books by "King, Stephen":
   - Search the Author index
   - Retrieve pointers to records #78, #145, #356, ...
   - Access each record to get complete book information
3. If you want all Mystery books:

- Search the Genre index under "Mystery"
- Retrieve pointers to records #12, #56, #198, #421, ...
- Access each record to get complete book information

13.1.6 Direct or Hashed File

Direct files, also known as hashed files, utilize the disk capability to access any block with a known address directly. Key characteristics include:

- Requires a key field in each record, similar to sequential and indexed sequential files
- No sequential ordering concept is maintained
- Employs hashing on the key value
- Typically organized with an overflow file structure

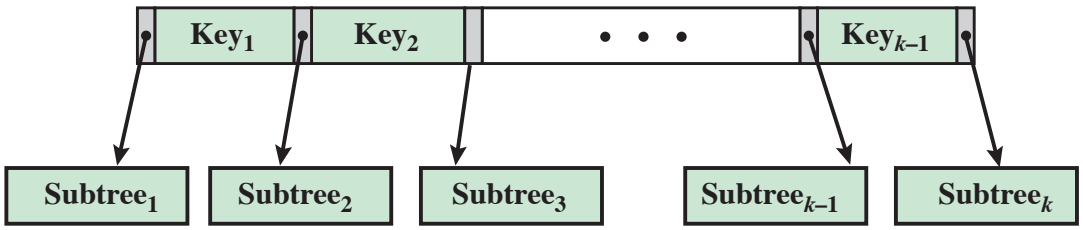
Direct files are particularly suitable for:

- Applications requiring very rapid access
- Systems using fixed length records
- Situations where records are accessed individually
- Common examples: directories, pricing tables, schedules, and name lists

13.2 B Trees

Index files provide access to individual records in databases or files. For large datasets, simple sequential index files become inefficient. More structured approaches include:

- **Two-level organization:** The original file is segmented, with an upper level containing sequential pointers to lower-level sections.
- **Multi-level extension:** The two-level approach extends to multiple levels, creating a tree structure.
- **Balanced trees:** Unstructured trees often develop uneven branches, causing inconsistent search times. Balanced trees, with branches of equal length, provide optimal average performance.
- **B-trees:** These have become the standard indexing method for databases and many file systems including Mac OS X, Windows, and several Linux implementations. B-trees efficiently support searching, insertion, and deletion operations.



A B-tree is a specialized tree structure with the following characteristics:

- The tree is composed of nodes and leaves.
- Each node contains:
  - At least one key (uniquely identifying a file record)
  - Multiple pointers to child nodes or leaves
  - The number of keys and pointers may vary within defined limits
- All nodes have the same maximum key capacity.
- Keys in a node are stored in nondecreasing order.
- Each key has an associated child that roots a subtree containing:

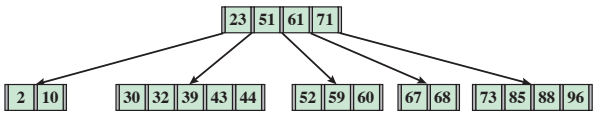
  - All nodes with keys less than or equal to the key
  - But greater than the preceding key
- Each node has one more pointer than keys (the rightmost pointer roots a subtree with all keys greater than any keys in the node).

FThe upper level contains $(k-1)$ keys and k pointers, satisfying:

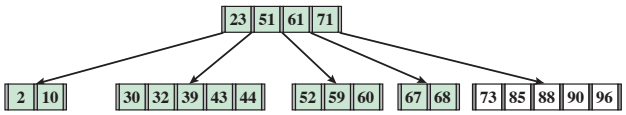
$$\text{Key}_1 < \text{Key}_2 < \ldots < \text{Key}_{k-1}$$

Each pointer references a node at the top level of a subtree. To search for a key, begin at the root node:

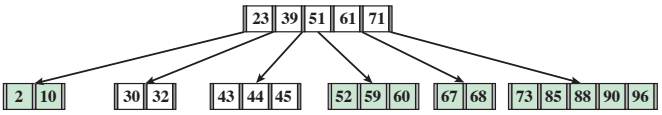
- If the key is found in the current node, the search concludes.
- Otherwise, traverse down one level based on these cases:
  - If the target key is less than the smallest key in the node, follow the leftmost pointer.
  - If the target key exceeds the largest key in the node, follow the rightmost pointer.
  - If the target key falls between two adjacent keys, follow the pointer between those keys.



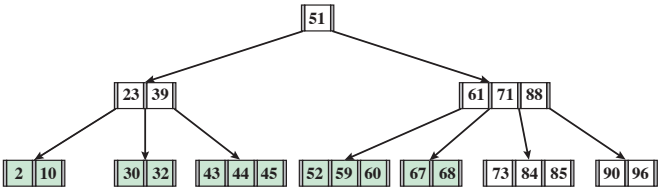
(a) B-tree of minimum degree d = 3.



(b) Key = 90 inserted. This is a simple insertion into a node.



(c) Key = 45 inserted. This requires splitting a node into two parts and promoting one key to the root node.



(d) Key = 84 inserted. This requires splitting a node into two parts and promoting one key to the root node
This then requires the root node to be split and a new root created.

A B-tree of minimum degree d satisfies:

- Each node contains at most $2d-1$ keys and $2d$ children/pointers.
- Each node except the root has at least $d-1$ keys and d pointers, making non-root internal nodes at least half full with at least d children.
- The root has at least 1 key and 2 children.
- All leaves appear at the same level and contain no information, serving as logical terminators (implementation may vary).
- A non-leaf node with k pointers contains $k-1$ keys.

B-trees typically have large branching factors, resulting in low-height trees.

13.3 File Directories

A file directory is an essential component of any file management system that contains metadata about files. The directory itself is a file managed by the operating system, with information typically accessed indirectly through system routines.

Each file entry in a directory typically includes:

- **File name:** Provides mapping between user-known names and actual files
- **Type information:** Identifies different file types and organizations
- **Storage information:** Includes file location and size
- **Access control:** Specifies ownership and access privileges for different users
- **Usage information:** Manages current file use and records usage history

In shared systems, access control is particularly important as it allows the file owner to grant specific privileges to other users.

Basic Information	
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory.
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
Address Information	
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
Access Control Information	
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, transmitting over a network
Usage Information	
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

When designing a file structure, we must consider the key operations performed on directories:

- **Search:** Locating entries corresponding to referenced files
- **Create file:** Adding new entries to the directory
- **Delete file:** Removing entries from the directory
- **List directory:** Displaying all or portions of directory contents with file attributes
- **Update directory:** Modifying entries when file attributes change

A simple sequential list structure proves inadequate for these operations. For individual users managing diverse file types (text documents, graphics, spreadsheets), this structure offers no organizational assistance and requires vigilance

to avoid name conflicts. In shared systems, these problems compound significantly—unique naming becomes critical, and concealing specific directory portions becomes challenging without inherent structural organization.

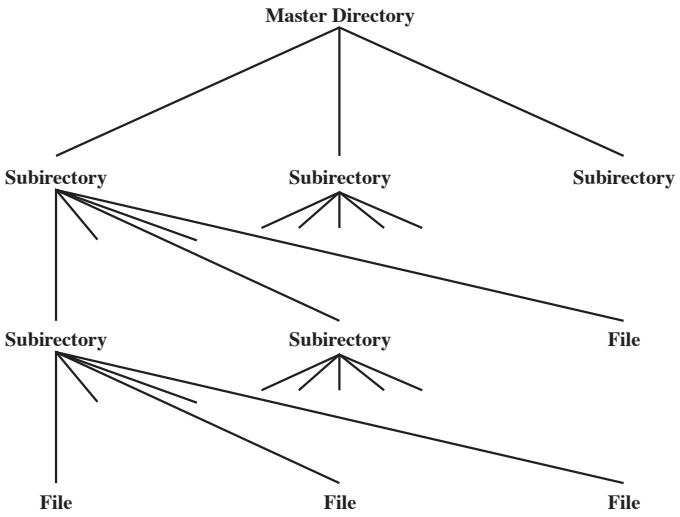
A simpler approach to file organization uses a **two-level hierarchy**:

- A master directory containing entries for each user directory
  - Stores address information for user directories
  - Provides access control mechanisms
- Individual user directories
  - Each contains a simple list of that user's files

This structure offers two advantages:

- File names need only be unique within a single user's collection
- The system can easily enforce directory access restrictions

However, this approach still fails to help users organize their files into meaningful structures.

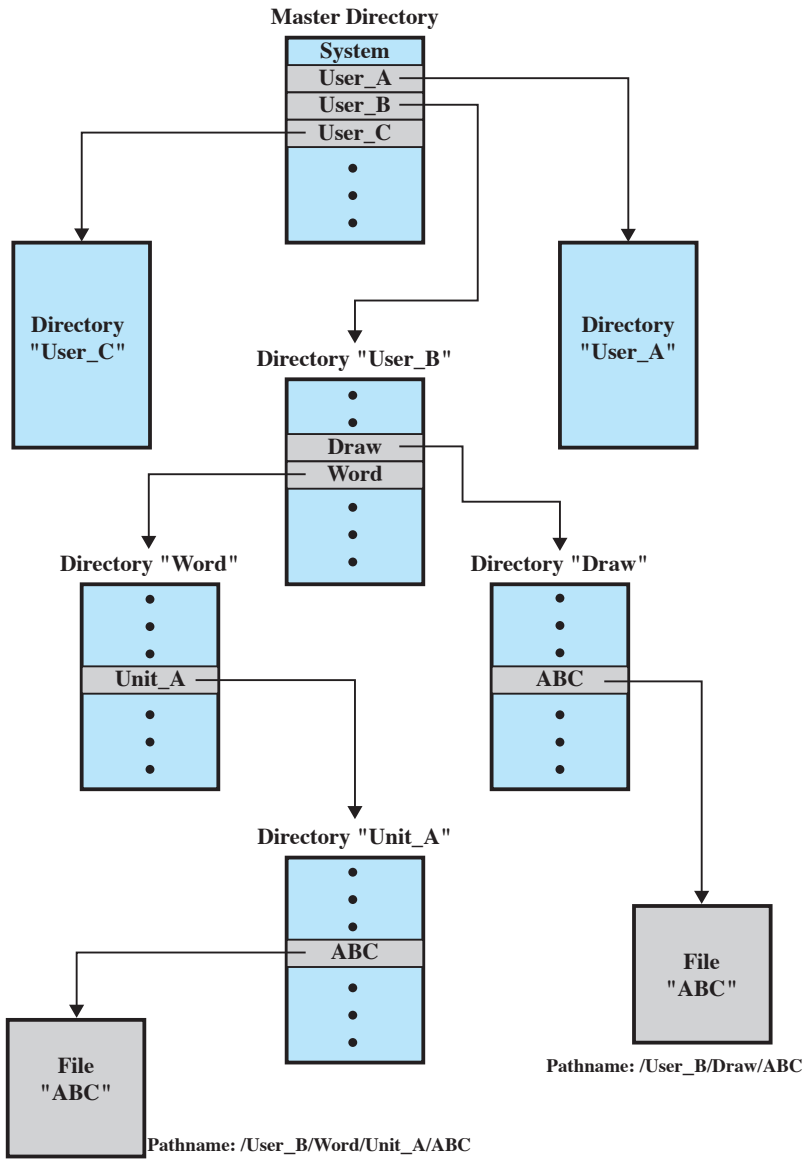


A hierarchical (tree-structure) approach offers a more powerful and flexible directory organization method that is almost universally adopted. The structure works as follows:

- A master directory exists at the top level
- Under the master directory are multiple user directories
- Each user directory may contain both subdirectories and files
- This pattern can repeat at any level: any directory may contain entries for both subdirectories and files

For the organization of each directory and subdirectory, two approaches exist:

- Simple approach: store each directory as a sequential file
- Preferred approach for large directories: use a hashed structure to avoid unnecessarily long search times



File systems require:

- Each file must have a unique identifier for unambiguous reference
- Users should not bear the burden of creating globally unique names

Tree-structured directories solve this problem by:

- Organizing files hierarchically from root directory through branches
- Creating unique pathnames as sequences: `/Directory/Subdirectory/Filename`
- Allowing identical filenames in different directories (e.g., `/User_B/Word/Unit_A/ABC` vs `/User_B/Draw/ABC`)

To avoid typing full pathnames:

- Users operate within a current "working directory"
- Files referenced relative to this position (e.g., `Unit_A/ABC` from `/User_B/Word/`)
- Default working directory at login is user's home directory
- Users can navigate up/down the directory tree to change working directory

13.4 File Sharing

File sharing in multiuser systems presents two main challenges:

- Access rights management

- Control of simultaneous access

An effective file system must provide flexible tools for file sharing among users, with various options to control access methods. Typically, specific access rights are granted to users or user groups. These rights form a hierarchical structure, where each level includes all previous rights.

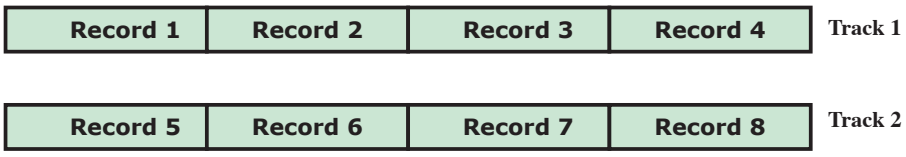
- **None:** User cannot discover the file's existence or access it
- **Knowledge:** User can verify file existence and ownership, allowing them to request additional access
- **Execution:** User can load and execute programs but cannot copy them (common for proprietary software)
- **Reading:** User can read, copy, and execute the file (some systems distinguish between viewing and copying)
- **Appending:** User can add data (often only at the end) without modifying existing content
- **Updating:** User can modify, delete, and add data to the file
- **Changing protection:** User can modify access rights granted to others (typically reserved for file owners)
- **Deletion:** User can remove the file from the system

The hierarchical nature means that granting a higher right (e.g., updating) automatically includes all lower rights (knowledge, execution, reading, and appending).

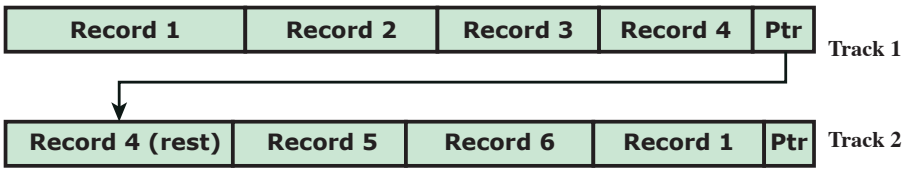
One user is designated as the owner of a file, typically the creator. The owner possesses all access rights and can grant permissions to others. Access can be provided to:

- **Specific user:** Individual users identified by user ID
- **User groups:** Collections of users managed by the system
- **All:** Every user with system access (public files)

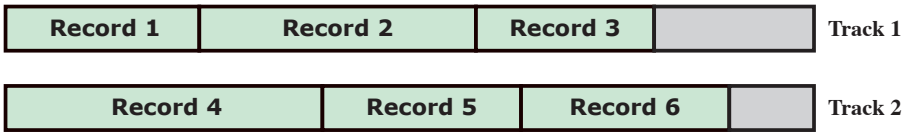
13.5 Record Blocking



(a) Fixed Blocking



(b) Variable Blocking: Spanned



(c) Variable Blocking: Unspanned

Records serve as the logical access unit of structured files, while blocks function as the I/O unit with secondary storage. For I/O operations, records must be organized into blocks. **A disk track is a circular path on the surface of a disk where data is stored.**

- **Length Type:** Fixed-length blocks are commonly used as they simplify I/O operations, buffer allocation in main memory, and organization on secondary storage.
- **Size Trade-off:** Larger blocks reduce I/O operations by transferring more records simultaneously, beneficial for sequential processing. However, larger blocks require larger I/O buffers, complicating buffer management.

- **Fixed blocking:** Uses fixed-length records with an integral number per block. May cause internal fragmentation (unused space at block end).
- **Variable-length spanned blocking:** Uses variable-length records packed without unused space. Records may span two blocks, requiring continuation pointers.
- **Variable-length unspanned blocking:** Uses variable-length records without spanning. Results in wasted space when remaining block space is insufficient for the next record.

13.6 Secondary Storage Management

File management on secondary storage involves two key challenges:

- Allocating disk blocks to files
- Managing available free space

These challenges are interconnected, as the file allocation method affects free space management and interacts with file structure.

13.6.1 File Allocation

When implementing file allocation on a single disk, several questions must be addressed:

- **Initial Allocation:** Should the maximum required space be allocated when a file is created?
- **Allocation Units:** Space is allocated as one or more contiguous units called portions. A portion contains contiguous allocated blocks, with size ranging from a single block to the entire file size. What portion size is optimal?
- **Tracking Structures:** What data structures should track allocated portions? Examples include file allocation tables (FAT) used in DOS and other systems.

A **preallocation policy** demands declaring a file's maximum size during creation. This works well when size can be reliably estimated, such as for:

- Program compilations
- Summary data files
- File transfers over networks

However, many applications cannot reliably predict maximum file sizes. In these cases, users typically overestimate to avoid space shortages, resulting in inefficient storage allocation. **Dynamic allocation** offers advantages by assigning space incrementally as needed.

The size of portions allocated to files represents a key design decision in storage systems. This decision involves a trade-off between individual file performance and overall system efficiency.

- **Performance benefit of contiguity:** Contiguous space allocation enhances performance, particularly for sequential operations and transaction-oriented systems.
- **Management overhead:** Small portions increase the size of allocation tables and associated management data structures.
- **Simplicity of reallocation:** Fixed-size portions (such as blocks) simplify the space reallocation process.
- **Storage efficiency:** Variable-size or small fixed-size portions minimize wasted storage space due to overallocation.

13.6.2 File Allocation Strategies

Two major alternatives exist for file allocation:

- **Variable, large contiguous portions:** Provides better performance with small allocation tables and reduced waste due to variable sizing. However, space reuse is challenging.
- **Blocks:** Small fixed portions offering greater flexibility. Requires larger tables or complex allocation structures. Contiguity is no longer a primary goal, as blocks are allocated as needed.

Both options work with either preallocation or dynamic allocation:

- With variable contiguous portions: Files receive one contiguous block group at preallocation, eliminating the need for allocation tables (only requiring a pointer to the first block and the block count).

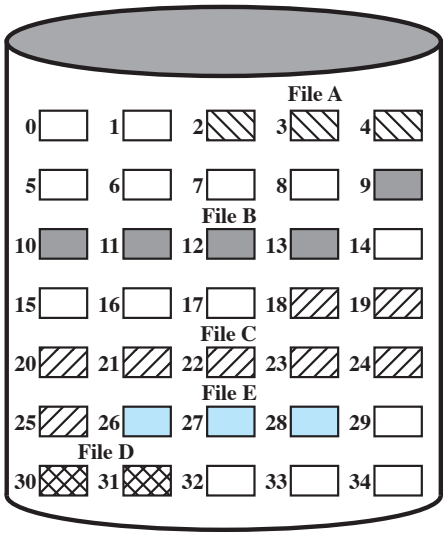
- With blocks: All required portions are allocated simultaneously, keeping the file allocation table size fixed as the block count remains constant.

When using variable-size portions, free space fragmentation becomes a concern. Possible strategies include:

- **First fit:** Select the first sufficiently sized contiguous block group from the free block list.
- **Best fit:** Select the smallest sufficiently sized unused group.
- **Nearest fit:** Select a sufficiently sized unused group closest to the previous allocation to enhance locality.

Determining the optimal strategy remains unclear due to numerous interacting factors: file types, access patterns, multiprogramming degree, system performance factors, disk caching, and disk scheduling.

`Contiguous File Allocation`



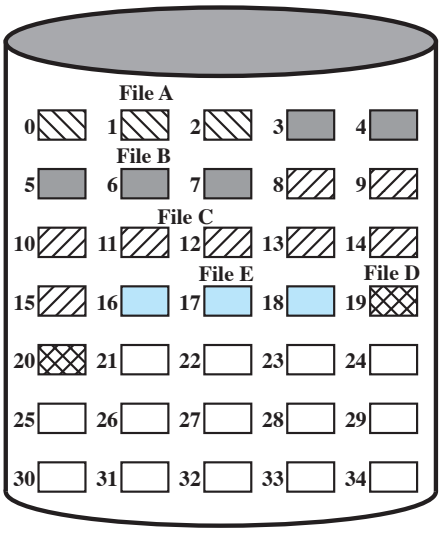
File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Contiguous allocation assigns a single contiguous set of blocks to a file at creation time. Key characteristics:

- Preallocation strategy using variable-size portions
- File allocation table requires only one entry per file (starting block and length)
- Optimal for sequential file access
- Allows efficient multiple block reads for improved I/O performance
- Simple block retrieval: location of ith block $= b + i - 1$ (where b is starting block)
- Main disadvantage: external fragmentation makes finding sufficient contiguous space difficult

`Contiguous File Allocation After Compaction`

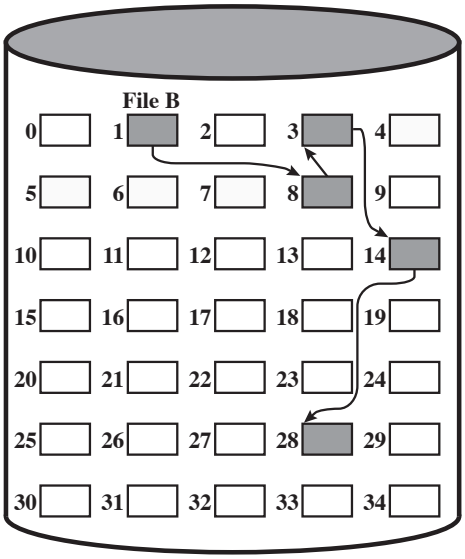


File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

From time to time, it will be necessary to perform a compaction algorithm to free up additional space on the disk. Also, with preallocation, it is necessary to declare the size of the file at the time of creation, with the problems mentioned earlier.

Chained Allocation



File Allocation Table

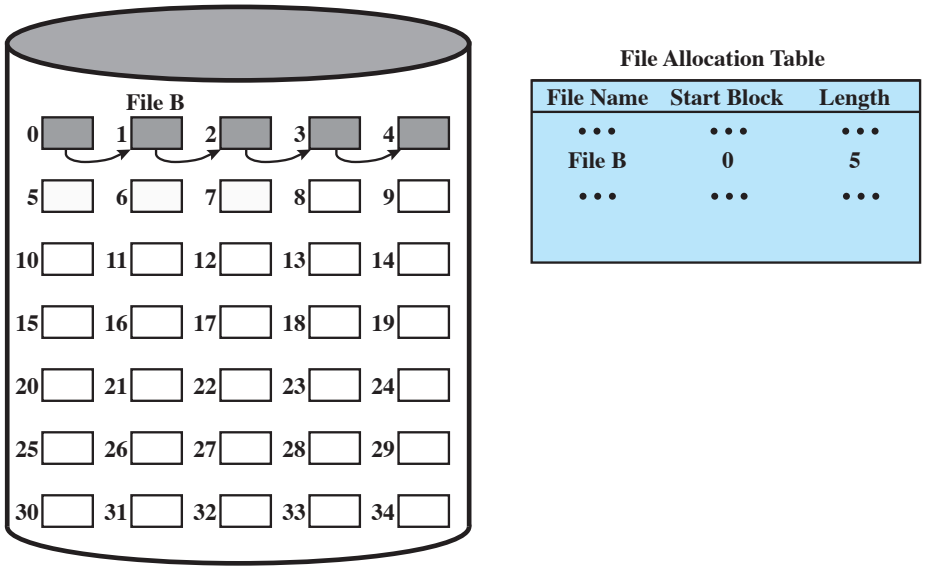
File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Chained allocation represents the opposite extreme from contiguous allocation:

- Allocation occurs on an individual block basis
- Each block contains a pointer to the next block in the chain
- File allocation table requires only a single entry per file, storing:
  - Starting block
  - File length
- Block selection is straightforward as any free block can be added to a chain
- No external fragmentation concerns since only one block is needed at a time
- Best suited for sequential files processed sequentially
- Accessing an individual block requires tracing through the chain to reach the desired block

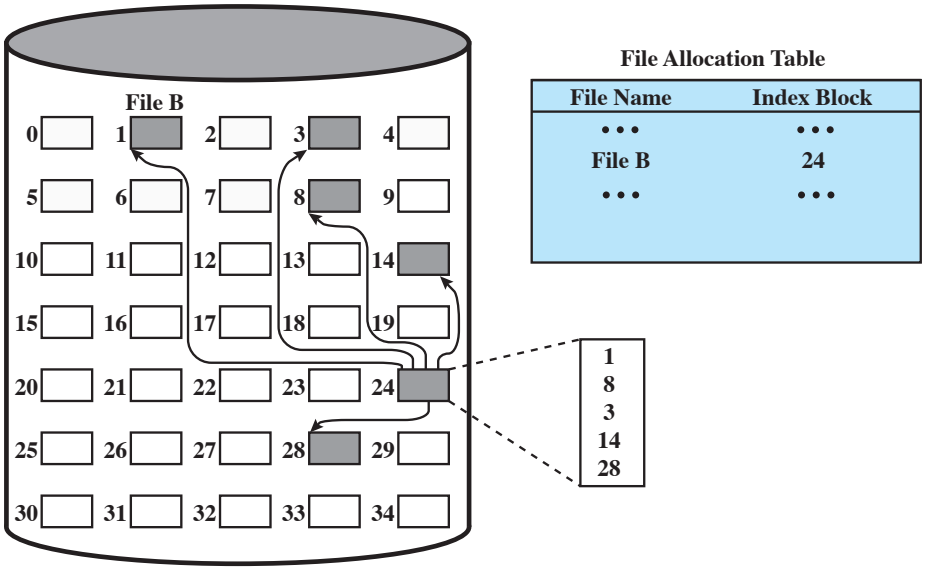
While preallocation is possible, blocks are typically allocated as needed.

Chained Allocation Afdter Consolidation



Chaining, as previously described, fails to accommodate the principle of locality. When sequential processing requires multiple file blocks, the system must access different disk areas separately. This inefficiency impacts both single-user and shared systems. To address this issue, some systems implement periodic file consolidation.

Indexed Allocation with Block Portions



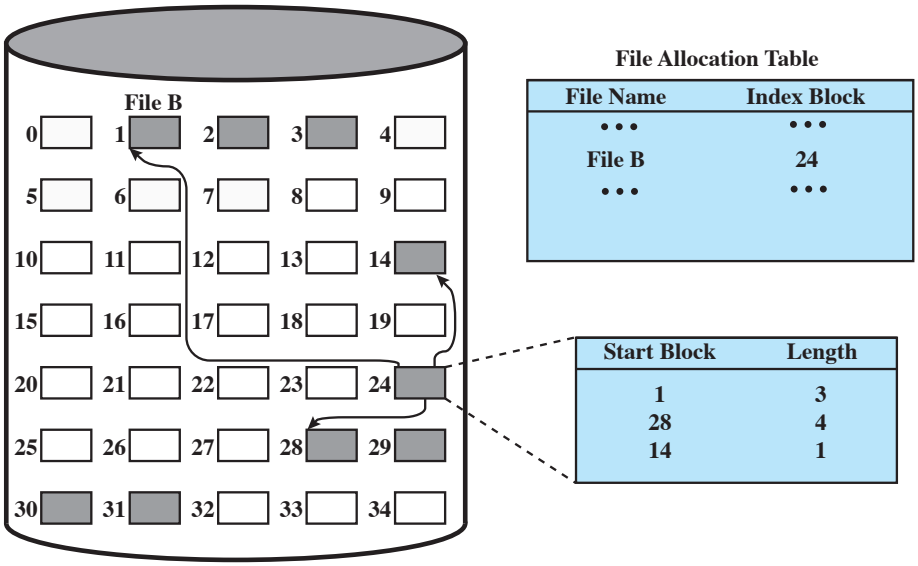
Indexed allocation resolves many issues present in contiguous and chained allocation methods. The file allocation approach uses a separate one-level index for each file, with each index entry corresponding to an allocated portion of the file. These indexes are typically stored in separate blocks rather than within the file allocation table itself, with table entries simply pointing to these index blocks.

Two allocation approaches exist:

- Fixed-size blocks (eliminates external fragmentation)
- Variable-size portions (improves locality)

File consolidation may be performed periodically, which reduces index size in variable-size allocation but not in block allocation. Indexed allocation's key advantage is supporting both sequential and direct file access, making it the most widely used file allocation method.

Indexed Allocation with Variable-Length Portions



THe portions can have. for examples, 3 blocks, 1 block, 4 blocks.

13.6.3 Free Space Management

Just as allocated file space requires management, unallocated disk space must also be tracked efficiently. Any file allocation technique requires knowledge of available disk blocks. A disk allocation table is needed alongside the file allocation table. Several implementation techniques exist:

- **Bit Vector:** Uses a bit map where each block is represented by a single bit (1=allocated, 0=free)
- **Linked List:** Maintains a linked list of free disk blocks
- **Grouping:** Stores addresses of multiple free blocks in the first free block
- **Counting:** Records address of first free block and count of contiguous free blocks

Each approach offers different trade-offs between storage efficiency and allocation performance.

`Bit Table`

The bit vector method uses a vector with one bit per disk block:

- 0 = free block
- 1 = block in use

For example, a 35-block disk might have this bit vector:

`00111000011111000011111111111011000`

Advantages:

- Easy to find free blocks or contiguous free block groups
- Compatible with all file allocation methods
- Minimal size for block allocation tracking

The memory required for a bit vector is:

$$\text{Size (bytes)} = \frac{\text{disk size in bytes}}{8 \times \text{file system block size}} \tag{25}$$

For a 16 GB disk with 512-byte blocks, the bit vector requires approximately 4 MB of memory.

Implementation Challenges:

- Memory usage: 4 MB is significant memory for a single function
- Disk storage alternative: An 8,000-block on-disk bit table would be too slow to search
- Performance impact: Exhaustive search becomes inefficient as the disk fills

Most implementations maintain auxiliary data structures:

- Divide the bit table into logical subranges
- Create a summary table tracking for each subrange:
  - Number of free blocks
  - Maximum size of contiguous free blocks
- File system first scans the summary table, then searches only appropriate subranges

Chained Free Portions

Free portions can be chained together using a pointer and length value in each free portion.

- This method has minimal space overhead, requiring only:
  - A pointer to the chain's beginning
  - The length of the first portion
- This approach is compatible with all file allocation methods:
  - For block-by-block allocation: select the free block at the chain's head and adjust the first pointer/length
  - For variable-length allocation: use first-fit algorithm to find suitable free portions, then adjust pointers/lengths

Disadvantages:

- Disk fragmentation increases with use, resulting in many single-block portions
- Performance issues:
  - Each block allocation requires reading the block first to retrieve the pointer to the next free block
  - Allocating many blocks at once significantly slows file creation
  - Deleting highly fragmented files is time-consuming

Indexing

The indexing approach manages free disk space by treating it as a file and utilizing an index table (similar to file allocation methods). For optimal efficiency:

- The index tracks variable-size portions rather than fixed blocks
- One table entry exists for each free disk portion
- This approach efficiently supports all file allocation methods

Free Block List

This method maintains a list of free block numbers in a reserved disk area. Each block is assigned a sequential number, requiring either 24 or 32 bits per block number.

- The space overhead is minimal, less than 1% of total disk space (4 bytes per 512-byte block when using 32-bit block numbers)
- While the complete free block list is too large for main memory, two efficient partial-storage techniques exist:
  - **Stack approach**: Store the first few thousand elements in main memory
    - Allocation: pop from top of stack (in memory)
    - Deallocation: push onto stack (in memory)
    - Disk transfers only needed when in-memory portion becomes full or empty
    - Provides near-zero access time in most cases
  - **FIFO queue approach**: Store a few thousand entries from both head and tail in memory
    - Allocation: take first entry from head
    - Deallocation: add to end of tail
    - Disk transfers only needed when in-memory head portion empties or tail portion fills
- For either approach, a background thread can sort in-memory lists to facilitate contiguous allocation

13.7 Volume

A volume is essentially a logical disk, though the term varies somewhat across operating systems and file management systems. A volume is:

> A collection of addressable sectors in secondary memory that an OS or application can use for data storage. The sectors in a volume need not be consecutive on a physical storage device; instead, they need only appear that way to the OS or application. A volume may be the result of assembling and merging smaller volumes.

Volumes can exist in several configurations:

- A single disk functioning as one volume
- A disk divided into partitions, with each partition serving as a separate volume
- Multiple disks treated as a single volume
- Partitions across multiple disks combined into a single volume