CSC362 Operating System

Project 3 – Implementation and Application of Semaphores

Dr. Qixin Deng

Wednesday 9th April, 2025

Implementation and Application of Semaphores

Practicing the use and implementation principles of semaphores, and applying them to solve the Producer-Consumer problem.

1. Experiment Objectives

- Implement semaphore in Linux 0.11, including semaphore creation, destruction, and operation.
- Resolve the producer-consumer problem using the implemented semaphore.

2. Experiment Tasks

In Linux 0.11, semaphore functionality has not been implemented yet. Achieving a fully POSIX-compliant semaphore system would undoubtedly be a significant accomplishment. However, time currently does not allow us to do so. Therefore, let's first create a simplified version of POSIX-like semaphores. Its function prototypes and standards are not entirely the same as POSIX, and it only includes the following system calls:

```
1 sem_t *sem_open(const char *name, unsigned int value);
2 int sem_wait(sem_t *sem);
3 int sem_post(sem_t *sem);
4 int sem_unlink(const char *name);
```

sem_t is a semaphore type, customized as needed by the implementation.
sem_open() is used to create a semaphore or to open an existing one. Name is the name of the semaphore. Different processes can share the same semaphore by providing the same name. If the semaphore does not exist, a new semaphore name is created; if it exists, the existing semaphore name is opened. value is the initial value of the semaphore, which is only relevant when creating a new semaphore; in other cases, it is ignored. On success, the return value is a unique identifier for the semaphore (such as an address in the kernel, ID, etc.), used by two other system calls. If it fails, the return value is NULL.
sem_wait() performs the semaphore's P atomic operation (The letter "P" originates from the Dutch word "proberen," which means "to test" or "to try."). If the conditions for continuing

execution are not met, it makes the calling process wait on the semaphore. A return of 0 indicates success, and -1 indicates failure.
sem_post() performs the semaphore's V atomic operation (The letter "V" comes from the Dutch word "verhogen," meaning "to increase" or "to increment."). If there are processes waiting on semaphore, it will wake one of them. A return of 0 indicates success, and -1 indicates failure.
sem_unlink() is used to delete the semaphore. A return of 0 indicates success, and -1 indicates failure.
Create a new file named **"sem.c"** in the kernel directory to implement the above functionalities. Then, write a short program **"pc.c"** to test the semaphore implementation. The whole process is very similar to assignment 1 – System Calls.
Write a short "pc.c" to solve the classical producer-consumer problem, achieving the following functionalities:

1. Establish one producer process and N consumer processes (N > 1).
2. Use a file to establish a shared buffer.
3. The producer process sequentially writes integers 0, 1, 2, ..., M into the buffer, where M ¿= 500.
4. Each consumer process reads one number from the buffer, removes the read number from the buffer, and then outputs its process ID and the number read to the standard output.
5. The buffer can only hold a maximum of 10 numbers at a time.

A possible output could be:

```
10: 0
10: 1
10: 2
10: 3
10: 4
11: 5
11: 6
12: 7
10: 8
12: 9
12: 10
12: 11
12: 12
......
11: 498
11: 499
```

The order of IDs may vary significantly, but the number following the colon will always increment by one starting from 0. The output format is not limited to this. This is the simplest output format. You can output richer information, including the IDs of producers and consumers, etc.

1. **Producer-Consumer Problem** The solution to the producer-consumer problem is almost universally found in all operating systems textbooks. Its basic structure is as follows:

```
Producer()
{
    Produce an item;
    P(Empty);  // Wait for available buffer resource
    P(Mutex);  // Ensure mutual exclusion
    Place the item into the empty buffer;
    V(Mutex);
    V(Full);   // Signal that product is available
}

Consumer()
{
    P(Full);
    P(Mutex);
    Retrieve an item from the buffer;
    V(Mutex);
    V(Empty);  // Signal that buffer space is available
    Consume the item;
}

```

   Clearly, demonstrating this process requires the creation of two types of processes: one type executes the Producer() function, and the other type executes the Consumer() function.

2. **Sharing Files Among Multiple Processes**

   In C language on Linux, file reading and writing can be accomplished through three methods:

   - Using standard C functions like fopen(), fread(), fwrite(), fseek(), and fclose();
   - Using system calls such as open(), read(), write(), lseek(), and close();
   - Using memory-mapped files with the mmap() system call. However, in Linux 0.11, only the first two methods are available.

   After a successful fork() call, the child process inherits most of the resources owned by the parent process, including any files opened by the parent process. Therefore, the child process can directly use the file pointers/descriptors/handles created by the parent process, accessing the same file as the parent process.

   When using standard C file functions, it's important to note that they utilize file buffers within the process space. These buffers are not shared between the parent and child processes. Hence, after any process completes a write operation, it's necessary to call fflush() to force the data to be updated on the disk, allowing other processes to read the required data.

   It's recommended to directly use system calls for file operations.

3. **The Terminal is Also a Critical Resource**

   It's natural to use printf() to output information to the terminal. However, when multiple processes are outputting simultaneously, the terminal becomes a critical resource that requires proper mutual exclusion protection. Otherwise, the output information may become garbled.

   Additionally, after printf(), the information is only stored in the output buffer and hasn't been sent to the terminal yet, which can also result in inconsistent output timing. Using fflush(stdout) ensures that the data is sent to the terminal.

4. **Atomic Operations, Sleep, and Wakeup**

   Linux 0.11 is a modern operating system that supports concurrency. Although it hasn't implemented any locks or semaphores for application-level usage, it must utilize lock mechanisms internally when multiple processes access shared kernel data to achieve mutual exclusion and synchronization. Locks are inherently atomic operations. By mimicking the locks in 0.11, it's possible to implement semaphores.

   Concurrent access to disks by multiple processes requires locks. The basic approach in Linux 0.11 for disk access is to allocate a portion of disk cache in memory to speed up disk access. When a process requests disk access, it first checks the disk cache in memory. If found, the process directly returns the data; otherwise, it allocates a section of free disk cache and issues a disk read/write request with this cache as a parameter. After issuing the request, the process sleeps (as disk I/O is slow and it should yield the CPU to other processes). This method is commonly used in many operating systems, including modern Linux and UNIX systems. Since multiple processes are involved in operating the disk cache and processes may be scheduled out while operating, mutual exclusion is necessary when operating the disk cache. Thus, locks are used, as well as sleep and wakeup operations.

   Below are two functions extracted from the file kernel/blk_drv/ll_rw_blk.c:

```
1  static inline void lock_buffer(struct buffer_head * bh)
2  {
3      cli();          // Disable interrupts
4      while (bh->b_lock)
5          sleep_on(&bh->b_wait);     // Put the current process to sleep on bh->b_wait
6      bh->b_lock = 1;
7      sti();          // Enable interrupts
8  }
9
10 static inline void unlock_buffer(struct buffer_head * bh)
11 {
12     if (!bh->b_lock)
13         printk("ll_rw_block.c: buffer not locked\n\r");
14     bh->b_lock = 0;
15     wake_up(&bh->b_wait);     // Wake up processes sleeping on bh->b_wait
16 }
```

   Analyzing lock_buffer(), it's apparent that atomic operations are achieved by enabling and

disabling interrupts when accessing the lock variable to prevent process switches. Of course, this method has drawbacks and is not suitable for multi-processor environments, but for Linux 0.11, it's a simple, direct, and effective mechanism.

Additionally, the above functions indicate that Linux 0.11 provides interfaces for process sleep with sleep_on() and wakeup with wake_up(). Their parameters are pointers to a structure - struct task_struct *, which represents the process PCB structure chain where processes sleep or are awakened.

Therefore, we can implement atomic operations using interrupt enable and disable operations and use sleep_on() and wake_up() for process sleep and wake up.

The functionality of sleep_on() is to put the current process to sleep on the specified list (note that this list is an implicit list, see the A Heavily Commented Linux Kernel Source Code for details). The functionality of wake_up() is to wake up all sleeping processes on the list. These processes will be scheduled for execution, so after being awakened, they need to reevaluate if they can continue running. Refer to the while loop in lock_buffer() for an example.

5. **Some Other Hints**

- You need to implement the functions sem_open(), sem_wait(), sem_post(), and sem_unlink() as system calls. To learn how to add system calls, please refer to Assignment 1. All of these 4 functions will be in file sem.c. Use the functions and mechanisms I mentioned in Instruction 4 to implement sem_wait() and sem_post().
- All producers and consumers need to be implemented using processes (fork()), rather than using thread libraries as we did in class. You need to use three semaphores, one of them is a binary semaphore, which is used to ensure mutual exclusion access to the buffer. Check the code we wrote during the lecture.
- You need to write the sem.h file, which should be placed in the oslab/linux-0.11/include/linux directory. Here, I directly give you the code:

```
#ifndef _SEM_H
#define _SEM_H
#include <linux/sched.h>
#define SEMTABLE_LEN    20
#define SEM_NAME_LEN    20
typedef struct semaphore{
    char name[SEM_NAME_LEN];
    int value;
    struct task_struct *queue;
} sem_t;
extern sem_t semtable[SEMTABLE_LEN];
#endif
```

To interpret the file:

(a) #ifndef _SEM_H and #define _SEM_H: These are preprocessor directives used for header file guards. They prevent the contents of the header file from being included

more than once in the same translation unit. If _SEM_H is not defined, the contents of the header file are included, and _SEM_H is defined. Subsequent inclusions of the header file will not include its contents again because _SEM_H is already defined.

(b) #include <linux/sched.h>: This line includes the header file sched.h from the Linux kernel source tree. It provides access to definitions related to task scheduling.

(c) #define SEMTABLE_LEN 20 and #define SEM_NAME_LEN 20: These are preprocessor macros defining the lengths of two arrays used later in the code. SEMTABLE_LEN specifies the length of an array (semtable) used to store semaphore objects, while SEM_NAME_LEN specifies the maximum length of a semaphore name.

(d) typedef struct semaphore ... sem_t;: This declares a structure named semaphore with three members: name, value, and queue. This structure represents a semaphore, where name is the name of the semaphore, value is its current value, and queue points to a task queue associated with the semaphore. Additionally, a typedef is used to define sem_t as an alias for struct semaphore.

(e) extern sem_t semtable[SEMTABLE_LEN];: This line declares an array named semtable of type sem_t with a length specified by SEMTABLE_LEN. The extern keyword indicates that the array is declared here but is defined elsewhere. This allows other source files to access the semtable array.

(f) #endif: This marks the end of the header file guard.

- open function "oflag" parameter and its meaning:

  Mode flags

  O_RDONLY - Open for reading only

  O_WRONLY - Open for writing only

  O_RDWR - Open for reading and writing

  Behavior flags

  O_CREAT: When this flag is specified, the file will be created if it doesn't already exist. If the file does exist, this flag has no effect unless used with O_EXCL, which then can be used for creating a new file under the condition that it does not exist.

  O_TRUNC: If the file already exists, it is truncated to zero length. This effectively deletes all the data in the file, starting you off with an empty file. This is used when the intent is to overwrite a file from scratch.

Submission

1. The Linux 0.11 kernel you modified.(30%)
2. A detailed report. (70%) Inside the report, you need to finish three parts:
   (a) "Goal". Explain the tasks and the expectations. (10%)
   (b) "Implementation Details". You need to show every place that is added, deleted, or modified by you (C code). You need to give all of the intermediate results and final results and discuss them. Also, you need to show you fully understand the "Instructions" part. (40%)
   (c) "Challenges". Put everything you think is challenging but not mentioned in the assignment document. Explain why it is challenging and how to overcome it.(20%)

One group needs to submit at least one copy to Canvas. **I do not accept late submissions.** Please put all group members' names in the report.

Academic Integrity

1. Collaborate honestly with your group members, ensuring that everyone contributes fairly to the project. Encourage and ensure equal participation from all group members, avoiding situations where one or two members do all the work.
2. Ensure that all sources used in the assignment are properly cited. This includes text, images, and ideas that are not your own.
3. Work should be original and not reused from past assignments, either your own or someone else's.
4. Use technology ethically, avoiding the temptation to use tools or resources in a manner not permitted by the professor, such as letting AI write code for the assignment.
5. Be aware of the consequences of violating academic integrity policies, which can range from grade penalties to more severe academic sanctions.