CSC362 Operating System

Project 2 – Tracking Process Execution Trajectories
Dr. Qixin Deng

Tuesday 18th March, 2025

Tracking Process Execution Trajectories

Tracking and statistically analyzing all states experienced by multiple processes from creation to termination.

1. Experiment Objectives

- A deeper understanding of the process from creation to termination.
- A more comprehensive comprehension of process state transitions.
- Understanding how to create and maintain files in the kernel.

2. Experiment Tasks

The entire process from creation (invoking fork() under Linux) to termination constitutes the lifespan of a process. The trajectory of a process during its lifespan manifests as multiple state transitions. For instance, after creation, a process enters the ready state; upon being scheduled, it switches to the running state. During execution, if a file read/write operation is initiated, the operating system transitions the process to a blocked (waiting) state to yield the CPU. Once the file operation completes, the operating system switches it back to the ready state, awaiting the process scheduling algorithm to schedule its execution.

1. Write a sample program based on the template "process.c" to achieve the following functionalities:
   (a) All child processes (more than 10 processes) run concurrently, with each child process typically running for no more than 30 seconds.
   (b) The parent process prints the IDs of all child processes to standard output and exits only after all child processes have exited.
2. Implement process execution tracking in Linux 0.11. The basic task is to maintain a log file /var/process.log in the kernel, recording the execution trajectories of all processes from operating system startup to shutdown.

3. Modify the time slice for process scheduling in 0.11, then rerun the same sample program and collect the same time data. Compare the results with the original settings to understand the differences brought about by different time slices.

4. The format of the /var/process.log file must be as follows:

   pid X time

   Where:

   (a) pid is the process ID.

   (b) X can be any one of N, J, R, W, and E, representing the process being newly created (N), entering the ready state (J), entering the running state (R), entering the blocked state (W), or exiting (E).

   (c) time indicates the time at which X occurs. This time is not physical time but rather the system's tick time.

   (d) fields are separated by tabs. For example:

```
12    N    1056
12    J    1057
4     W    1057
12    R    1057
13    N    1058
13    J    1059
14    N    1059
14    J    1060
15    N    1060
15    J    1061
12    W    1061
15    R    1061
```

3. Instructions

The writing of process.c involves the fork() and wait() system calls. Modifying the 0.11 kernel involves init/main.c, kernel/fork.c, and kernel/sched.c. Before starting the experiment, it would be beneficial to thoroughly read relevant sections of the A Heavily Commented Linux Kernel Source Code.

1. **Write process.c**

   It refers to a program that generates various processes. Our modifications to 0.11 involve recording the system's scheduling of these processes to a log file. This program can run on any Unix/Linux system, so it's recommended to debug it on Ubuntu before copying it to run on 0.11.

   I offer you a template of porcess.c file. It mainly implements a function:

```
/*
 * This function consumes CPU and I/O time according to the parameters.

 * last: Total time the function actually occupies CPU and I/O,
 * excluding time spent in the ready queue, must be >= 0.

 * cpu_time: Time spent continuously occupying CPU in one go, must be >= 0.

 * io_time: Time spent for one I/O operation, must be >= 0.

 * If last > cpu_time + io_time, then repeat the cycle of occupying CPU
 * and I/O until the total runtime exceeds last.

 * All times are in seconds.
 */
 cpuio_bound(int last, int cpu_time, int io_time);

//several examples to show its usage:
//If a process needs to occupy the CPU for 10 seconds, it can call:
 cpuio_bound(10, 1, 0);

//To prioritize I/O tasks:
cpuio_bound(10, 0, 1);

//Alternating between CPU and I/O every 1 second:
cpuio_bound(10, 1, 1);

//More I/O, less CPU:
cpuio_bound(10, 1, 9);  // I/O time is 9 times CPU time
```

   Modify this template to use fork() to create several child processes that run simultaneously. The parent process waits for all child processes to exit before exiting itself. Each child

process executes cpuio_bound() according to your preference, either with different or identical parameters, thereby completing a personalized sample program. This can be used to verify the correctness of modifications to the log file. The wait() system call allows the parent process to wait for the exit of child processes.

Something might be helpful:

(a) In Ubuntu, the top command can monitor real-time process status. In top, pressing 'u' and then entering your username can limit the display to only the processes running under your identity, making it easier to observe. Pressing 'h' provides help.

(b) In Ubuntu, the ps command can display the status of various processes at the time. "ps aux" will display all processes; "ps aux — grep xxxx" will only display processes named xxxx.

(c) In Linux 0.11, pressing F1 can display the real-time status of all current processes.

2. **Log File**

After the operating system boots up, it first opens /var/process.log and then writes a record to the log file for each process state transition, much like a user-space application. However, the presence of kernel state makes many details of the process entirely different.

(a) **Open log file.**

To start recording as early as possible, the log file should be opened when the kernel starts up. The entry point of the kernel is the main() function in init/main.c, where a segment of code is as follows:

```
1  ......
2  move_to_user_mode();
3  if (!fork()) {        /* we count on this going ok */
4      init();
5  }
6  ......
```

This piece of code runs in process 0. It switches to user mode first and then, for the first time across the system, calls fork() to create process 1. Process 1 then calls init(). In init():

```
1  ......
2  setup((void *) &drive_info);        // Load the file system
3
4  // Open /dev/tty0, associating file descriptor 0 with /dev/tty0
5  (void) open("/dev/tty0", O_RDWR, 0);
6
7  (void) dup(0);                 // Associate file descriptor 1 with /dev/tty0
8  (void) dup(0);                 // Associate file descriptor 2 with /dev/tty0
9
10 ......
```

This piece of code establishes file descriptors 0, 1, and 2, which correspond to stdin, stdout, and stderr, respectively. Their values are system standard and cannot be changed. Descriptor 3 can be associated with the log file. After initializing the file system and associating descriptors 0, 1, and 2, the log file can be opened to start recording process execution trajectories. To access the log file as early as possible, we need to ensure that these tasks are completed in process 0. Therefore, this code segment should be moved from init() to main() and placed after move_to_user_mode() (but cannot be placed any earlier), while also adding the code to open the log file. The modified main() function is as follows:

```
......
move_to_user_mode();

/*************** Addition Starts ***************/
setup((void *) &drive_info);
// Associate file descriptor 0 with /dev/tty0
(void) open("/dev/tty0", O_RDWR, 0);
(void) dup(0);        // Associate file descriptor 1 with /dev/tty0
(void) dup(0);        // Associate file descriptor 2 with /dev/tty0
(void) open("/var/process.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);
/*************** Addition Ends ***************/

if (!fork()) {        /* we count on this going ok */
    init();
}

......
```

The parameters used to open the log file signify creating a write-only file, and if the file already exists, its contents are cleared. The file permissions allow read and write access for all users.

As a result, file descriptors 0, 1, 2, and 3 are established in process 0. According to the principles of fork(), process 1 inherits these file descriptors, so there's no need to open them again in init(). Subsequently, all newly created processes are descendants of process 1 and will inherit these descriptors.

(b) **Write log file**

The log file will be used to record the trajectories of process state transitions. All state transitions occur within the kernel. In kernel mode, the write() function is disabled; only printk() can be called. Writing a write() function that can be called in kernel mode is challenging, so here is the source code directly. It is mainly written based on printk() and sys_write():

```
#include <linux/sched.h>
#include <sys/stat.h>

static char logbuf[1024];
```

```
int fprintk(int fd, const char *fmt, ...)
{
    va_list args;
    int count;
    struct file * file;
    struct m_inode * inode;

    va_start(args, fmt);
    count = vsprintf(logbuf, fmt, args);
    va_end(args);

    /* If outputting to stdout or stderr, directly call sys_write */
    if (fd < 3) {
        __asm__("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
            "call sys_write\n\t"
            "addl $8, %%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            ::"r" (count), "r" (fd): "ax", "cx", "dx");
    }
    /* Assume descriptors >= 3 are associated with files*/
    else {
        /* Get file handle from process 0's file descriptor table */
        if (!(file = task[0]->filp[fd]))
            return 0;
        inode = file->f_inode;

        __asm__("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
            "pushl %2\n\t"
            "call file_write\n\t"
            "addl $12, %%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            ::"r" (count), "r" (file), "r" (inode): "ax", "cx", "dx");
    }
    return count;
```

```
}
```

Since its functionality is similar to printk, it is recommended to place this function in kernel/printk.c. The usage of fprintk() is analogous to the C standard library function fprintf(), with the only difference being that the first parameter is a file descriptor instead of a file pointer. For example:

```
// Print the ID of the running process to stdout
fprintk(1, "The ID of the running process is %ld", current->pid);
// Output to the log file
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'R', jiffies);
```

3. Tracking process execution trajectories

(a) jiffies tik tok

jiffies is defined as a global variable in the kernel/sched.c file:

```
    long volatile jiffies=0;
```

It records the number of clock interrupts that have occurred from boot time to the current time. In the sched_init() function in the kernel/sched.c file, the clock interrupt handler is set to:

```
    set_intr_gate(0x20,&timer_interrupt);
```

While in the kernel/system_call.s file, timer_interrupt is defined as:

```
    timer_interrupt:
    ......
    incl jiffies     # add jiffies
    ......
```

This indicates that "jiffies" represents the number of clock interrupts that have occurred since boot time until now, which is also referred to as the "tick count." In addition, in the sched_init() function in kernel/sched.c, there is the following code:

```
    outb_p(0x36, 0x43); // Set 8253 mode
    outb_p(LATCH & 0xff, 0x40);
    outb_p(LATCH >> 8, 0x40);
```

These three statements are used to set the interval for each clock interrupt, referred to as LATCH, which is defined as a macro in the file kernel/sched.c:

```
#define LATCH  (1193180/HZ)
#define HZ 100  //defined in include/linux/sched.h
```

Adding to that, the input clock frequency of the PC's 8253 timer chip is 1.193180 MHz, which is equivalent to 1193180 ticks per second. Therefore, LATCH = 1193180/100, meaning a clock interrupt occurs every 11931.8 ticks, or every 1/100 second (10 ms). Consequently, jiffies effectively records how many 10 ms intervals have passed since the system boot.

(b) **Find state switching points**

We need to identify all the code points where process state transitions occur and add appropriate code at these points to output the process state changes to a log file. This task involves an understanding of fork.c, sched.c within the kernel, and exit.c might also be involved. We provide two examples to describe how this task should be approached, and experimenters can follow suit for other scenarios.

The first example is to see how to record the beginning of the life cycle of a process. Of course, this event is the process creation function fork(). The fork() function is implemented as sysT_fork() in the kernel. The "function" is implemented in the file kernel/system_call.s as:

```
sys_fork:
    call find_empty_process
    ......
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call copy_process  //call copy_process to create a new process
    addl $20,%esp
```

So the function that actually implements process creation is copy_process(), which is defined in kernel/fork.c as:

```
int copy_process(int nr,...)
{
    struct task_struct *p;
    ......
    p = (struct task_struct *) get_free_page();  //get a task_struct space
    ......
    p->pid = last_pid;
    ......
    p->start_time = jiffies;    //set start_time to jiffies
    ......
    //Set the process state to ready. The state of all ready processes is
    //TASK_RUNNING(0). The global variable current points to the running process.
```

```
        p->state = TASK_RUNNING;
        return last_pid;
    }
```

Therefore, to complete the recording of the process running trajectory, it is necessary to add an output statement in copy_process(). There are two states to be output here, namely "N (New)" and "J (Ready)".

The second example is to record the time of entering a sleep state. sleep_on() and interruptible_sleep_on() put the current process into a sleep state. These two functions are defined in the kernel/sched.c file as follows:

```
    void sleep_on(struct task_struct **p)
    {
        struct task_struct *tmp;
        ......
        tmp = *p;
        \\Read carefully, you actually insert current into the
        \\head of the "ready queue", and tmp is the original head.
        *p = current;
        current->state = TASK_UNINTERRUPTIBLE; //switch to sleep state
        schedule();  //give up CPU

        /*Wake up the previous (tmp) sleeping process in the queue.
        0 is better replaced by TASK_RUNNING
        This situation must be taken into consideration
        when the recording process is awakened,
        and you must pay attention!*/

        if (tmp)
            tmp->state=0;
    }
    /* The difference between TASK_UNINTERRUPTIBLE and TASK_INTERRUPTIBLE
     *  is that uninterruptible sleep can only be explicitly awakened by wake_up(),
     *  and then Wake up by
     *
     *   if (tmp) tmp->state=0;
     *
     *  in sequence, so the uninterruptible sleeping process must be woken up
     *  strictly from the head of the "queue" (a queue maintained by the
     *  pointer variable tmp placed in the process kernel stack). For
     *  interruptible processes, in addition to waking up with wake_up,
     *  you can also use signals (sending a signal to the process is actually
     *  to set a certain position of a vector maintained in the process PCB.
     *  The process needs to process this at the appropriate time.
     *  Interested experimenters can read about the code) to wake up,
     *  as in schedule():
```

```
 *
 *  for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
 *      if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
 *          (*p)->state==TASK_INTERRUPTIBLE)
 *          (*p)->state=TASK_RUNNING;//wake up
 *
 * That is, when the process is in interruptible sleep, it will wake up
 * if it encounters some signals. A problem with such a wake-up is that
 * it may wake up a process in the middle of the waiting queue. At this
 * time, the chain needs to be adjusted appropriately. The main difference
 * between the interruptible_sleep_on and sleep_on functions is here.
 */
void interruptible_sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;
    ......
    tmp=*p;
    *p=current;
repeat:    current->state = TASK_INTERRUPTIBLE;
    schedule();
    /* If the process at the head of the queue and the current process that
    * was just woken up are not the same, it means that a process was woken
    * up from the middle of the queue and needs to be processed.
    */
    if (*p && *p != current) {
    //Wake up the queue head and put yourself to sleep again through goto repeat
        (**p).state=0;
        goto repeat;
    }
    *p=NULL;
    if (tmp)
        tmp->state=0;
}
```

I believe you have found appropriate places to insert statements to record the transitions of processes from running to sleeping.

In general, Linux 0.11 supports four types of process state transitions: ready to run, running to ready, running to sleep, and sleep to ready, in addition to two cases of creation and exit. The transition between ready and running states is completed through schedule() (where the scheduling algorithm is located); running to sleep relies on sleep_on() and interruptible_sleep_on(), as well as the system calls sys_pause() and sys_waitpid() for processes to voluntarily sleep; and the transition from sleep to ready relies on wake_up(). Therefore, by inserting appropriate processing statements at suitable positions in these functions, comprehensive tracking of the process execution trajectory can be achieved.

To make the generated log file more accurate, please note the following:

i. The last step of process exit is to notify the parent process of its own exit, with the

purpose of waking up the parent process waiting for this event. In terms of timing, the child process should exit first before the parent process wakes up.

ii. When schedule() finds the next process, it is the process that is going to run next (note: it is essential to analyze clearly what "next" refers to). If the next process happens to be the one currently in the running state, switch_to(next) will also be called. In this case, it's as if the current process's state hasn't changed.

iii. When the system has nothing to do, process 0 will continuously call sys_pause() to activate the scheduling algorithm. At this time, its state can be waiting, waiting for other runnable processes, or it can be considered running because it is the only process running on the CPU, even though its execution results in waiting.

(c) **Managing log files** Managing log files is unrelated to code writing, but there are several key points to note:

i. Execute the "sync" command before closing Bochs each time. It flushes the cache to ensure that files are indeed written to the disk.

ii. In 0.11, you can use "ls -l /var" or "ll /var" to check if process.log is created, and its attributes and length.

iii. In 0.11, you can use "vi /var/process.log" or "more /var/process.log" to view the entire log file. However, it's more comfortable to copy it to Ubuntu for viewing.

iv. In 0.11, you can use "tail -n NUM /var/process.log" to view the last NUM lines of the log file.

In a possible scenario, the first few lines of the process.log file may be as follows:

```
1    1    N    48    // Process 1 created (init()). Previously, process 0 was
2                    // created and run, but why doesn't it appear in the log file?
3    1    J    49    // After creation, enter the ready queue
4    0    J    49    // Process 0 transitioned from running to ready, yielding the CPU
5    1    R    49    // Process 1 is running
6    2    N    49    // Process 1 created process 2.
7                    // 2 will run the /etc/rc script and then exit
8    2    J    49
9    1    W    49    // Process 1 starts waiting (waiting for process 2 to exit)
10   2    R    49    // Process 2 is running
11   3    N    64    // Process 2 created process 3.
12                   // 3 is a subprocess of the /bin/sh script
13   3    J    64
14   2    E    68    // Process 2 moves on without waiting for process 3 to exit
15   1    J    68    // Process 1 was previously blocked waiting for process 2 to exit.
16                   // After process 2 exits, it re-enters the ready queue
17   1    R    68
18   4    N    69    // Process 1 created process 4, i.e., the shell
19   4    J    69
20   1    W    69    // Process 1 waits for the shell to exit
21                   // (unless the exit command is executed, the shell will not exit)
22   3    R    69    // Process 3 starts running
23   3    W    75
```

```
    4    R    75
    5    N    107    // Process 5 is a process created by the shell for an unknown purpose
    5    J    108
    4    W    108
    5    R    108
    4    J    110
    5    E    111    // Process 5 exits quickly
    4    R    111
    4    W    116    // Shell waits for user input commands.
    0    R    116    // Process 0 re-emerges because there's nothing to do
    4    J    239    // User inputs a command, waking up the shell
    4    R    239
    4    W    240
    0    R    240
    ......
```

4. **Modify time slice** Below is the scheduling function schedule in Linux 0.11, defined in the file kernel/sched.c:

```
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];

        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }

        if (c)
            break;

        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
    switch_to(next);
```

Analyzing the code, we understand that the scheduling algorithm of Linux 0.11 selects the ready process with the highest counter value for scheduling. Here, the counter value of the running process (i.e., current) decreases by 1 with each clock interrupt (occurring every 10ms), making it a typical round-robin scheduling algorithm based on time slices.

Additionally, from the above program, when there are no ready processes with counter values greater than 0, all processes, including those in a blocked state, undergo counter decay with the formula

```
(*p)->counter = ((*p)->counter >> 1) + (*p)->priority.
```

This results in all processes' counters being halved and then added to their priority values. Consequently, for processes that are actively blocked, the longer they stay in the blocked queue, the higher their priority and, therefore, the larger their allocated time slice.

Overall, Linux 0.11's process scheduling is a combination of considering process priorities and dynamically adjusting time slices.

To modify the time slice size for the existing scheduling algorithm, we need to know two things:(1)How is the process counter initialized? (2)What value is assigned when a process's time slice is exhausted?

Firstly, regarding the initialization of the process counter, it's evident that this value is set in the fork(). Linux 0.11's fork() invokes copy_process() to copy information from the parent process (hence the name "fork"). Looking at the implementation of copy_process() (also in the kernel/fork.c file), we find the following two statements:

```
1    *p = *current;
2    p->counter = p->priority;
```

Because the parent process's counter value has changed while the priority hasn't, the second line of code sets p-¿counter to p-¿priority. Each process's priority is inherited from the parent process unless it changes its priority itself. Searching through all the code, there is only one place where priority is modified, which is the nice system call:

```
1    int sys_nice(long increment)
2    {
3        if (current->priority-increment>0)
4            current->priority -= increment;
5        return 0;
6    }
```

This experiment assumes that no one has called the nice system call, so the initial value of the time slice is the priority of process 0, as defined in the INIT_TASK macro:

```
1    #define INIT_TASK \
2        { 0,15,15, //state; counter; and priority;
```

Now, addressing the second question, when the counter of a ready process reaches 0, it won't be scheduled (as schedule selects the process with the highest counter, greater than 0). When all ready processes have counters set to 0, the following statement is executed:

```
1     (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
```

Clearly, the newly calculated counter value is equal to the priority, which is the initial size of the time slice.

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.