Project 2: Process Management

CSE 330: Operating Systems - Fall 2022

Due by **25th October 2022 11:59 PM**

Summary

We will start serious kernel development from the second project. In this project, we will implement a kernel module to solve the classic synchronization problem, the producer-consumer problem, using the Linux VM you set up in Project 1. We will use kernel threads to implement producers and consumers and use kernel-space semaphores to implement synchronization. This project will provide us with interesting kernel development experience and help us understand how to solve synchronization problems in real-world operating systems.

Description

In this project, you will implement a new kernel module to calculate the total elapsed time of all the processes that belong to a given user in the system. The producer and the consumer are kernel threads. The producer searches the system for all the processes that belong to a given user and adds the processes information to the shared buffer. The consumer removes the processes from the buffer, collects the elapsed time of these processes, and outputs the total elapsed time in the kernel log. There is only one producer, but there can be multiple consumers. The producer and the consumers share a fixed-size buffer that contains process descriptors (task_struct).

Specifically, you can follow the steps below to develop this project. Note that, unlike Project 1, there are no step-by-step instructions in this project. You will need to use your understanding of OS process management and the Linux functions/macros/data structures explained below to complete these steps.

1. **Module interface and module_init**

   - You **MUST** name your module "producer_consumer".
   - It takes four arguments & you **MUST** name your input parameters for the kernel module as follows & in the same order such that the test script can load the kernel module successfully. Failure to do so, your kernel module will not be loaded by the automated test script.
     - **buffSize**: The buffer size
     - **prod:** number of producers (0 or 1), and
     - **cons:** number of consumers (a non-negative number).
     - **uuid**: The UID of the user
   - To get the UID. You can directly use the "id -u <username>" command.
   - In your module, you will use the **module_param()** macro to pass input arguments to your kernel module. Call this macro at the beginning of your module code.

```
/*macro for module command line parameters.
name is the name of the parameter,
type is the type of the parameter, and
perm sets the visibility in sysfs.
For example, module_param(buff_size, int, 0) defines an input
argument named buffer_size, type is int, and the default value is
0.*/
module_param(name, type, perm);
```

Reference on passing command line arguments to a kernel module.

- [Passing data to a kernel module – module_param]

This project requires two kernel threads: producer thread and consumer thread. To create and start the kernel threads, you can use the `kthread_run()` function.


```
/* threadfn is the function to run in the thread; data is the data
pointer for threadfn; namefmt is the name for the thread. It returns
a pointer to the thread's task_struct if the thread creation is
successful or ERR_PTR(-ENOMEM) if it fails.*/
struct task_struct *kthread_run(int (*threadfn)(void *data), void
*data, *const char *namefmt, ...)
```


Example code to create a kernel thread:

```
#include <linux/kthread.h>
// the function to run in the thead
static int kthread_func(void *arg) {
    ...
}
// Create and run "thread-1"
ts1 = kthread_run(kthread_func, NULL, "thread-1");
```

2. Producer Thread

The producer thread searches the system for all the processes that belong to a given user and adds their task_struct to the shared buffer. There is only one producer in the system, and it exits after it has iterated through the entire task list.

The kernel stores all the processes in a circular doubly linked list called task list. Each element in the task list is a process descriptor of the type struct task_struct which contains all the information about a process. The below figure shows the process descriptors and task list.

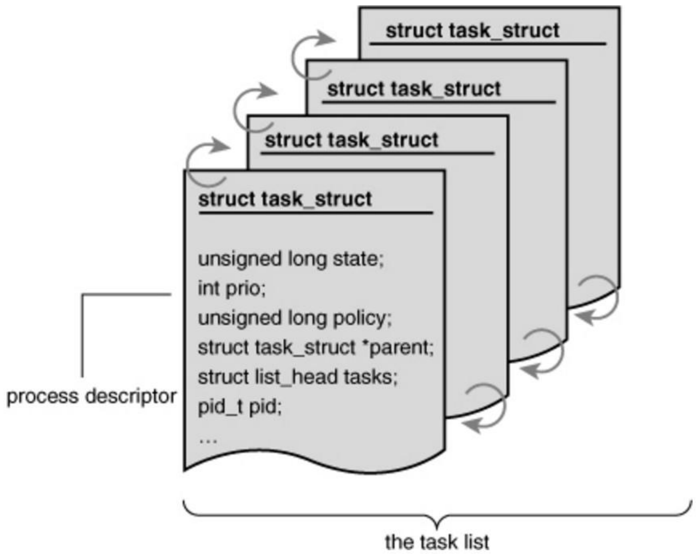

Fig. The process descriptor and task list.

You can use the `for_each_process` macro to iterate through the task list to access each task_struct. The macro goes over all the task_struct in the task list one by one. The task_struct contains the process' PID in pid and the process' user's UID in `cred->uid.val`


```
for_each_process(struct task_struct *p) // On each iteration, p points to the next task in the list.
task_struct *task;
task->pid // PID of the process
task->cred->uid.val // UID of the user of the process
```

The following code example calculates the number of processes in the task list.

```
#include<linux/sched.h>
#include <linux/sched/signal.h>

struct task_struct* p;
size_t process_counter = 0;
for_each_process(p) {
   ++process_counter;
}
```

- To synchronize producers and consumers, you will need **three** semaphores: mutex, full, and empty.
- In order to use a kernel semaphore, first, you need to define a semaphore. Then, you will use `sema_init()` to initialize the semaphore.
- To signal a semaphore, you will use `down_interruptible()`. To wait on a semaphore, you will use `up()`.

```
struct semaphore name; // Defines a semaphore with a given name
static inline void sema_init(struct semaphore *sem, int val) // a function to initialize a semaphore structure
void down_interruptible(struct semaphore *sem) // acquire a lock
void up(struct semaphore *sem) // release a lock
```

The following code snippet explains how to use a kernel semaphore.

```
#include <linux/semaphore.h>

// Semaphore Definition
struct semaphore empty; // define a semaphore named 'empty'
sema_init(&empty, 5);   // init the semaphore as 5

// if the thread works in an infinite loop, this is how it knows when to stop. Check (4) module_exit for more information.
while (!kthread_should_stop())
{
    if (down_interruptible(&empty))
    /* down_interruptible allows the calling thread to be
       interrupted even if it has not acquired the semaphore that
       it has been waiting for */
         break; // exit

     // Critical section

      up(empty)
      // signal the semaphore
}
```

Note: Your module cannot exit, if there is a thread still running or waiting on a semaphore. To avoid this, you must signal all the semaphores and stop all the threads in `module_exit`. See (4) below for more details. Ensure you implement module_exit correctly, or your kernel will hang or crash.

For each item your kernel module produces, print the following information in the mentioned format in the kernel log:

```
[<Producer-thread-name>] Produced Item#-<Item-Num> at buffer index:
<buffer-index> for PID:<PID of the process>

Example:
[Producer-1] Produced Item#-12 at buffer index:1 for PID:136042
```

3. Consumer Thread

The consumer thread reads out the task_struct of the processes from the buffer and calculates the elapsed time for each process and the total elapsed time. There can be multiple consumers in the system, and they work in an infinite loop, remember to check kthread_should_stop() so they know when to stop.

To calculate the total elapsed time of each process, you will retrieve the start time of the process (in nanoseconds) from start_time in its task_struct. Then you will use ktime_get_ns() to find out the current time. The difference between these two values is the elapsed time for this process.


```
task_struct *task;
task->start_time // start time of the process (in nanoseconds)
ktime_get_ns()   // current time in nanoseconds.
//Defined in include/linux/timekeeping.h
```

For each item your kernel module consumes, print the following information in the mentioned format in the kernel log:

```
[<Consumer-thread-name>] Consumed Item#-<Item-Num> on buffer index:
<buffer-index> PID:<PID consumed>  Elapsed Time- <Elapsed time of the
consumed PID in HH:MM:SS>

Example:
[Consumer-1] Consumed Item#-12 on buffer index:1 PID:136042 Elapsed
Time-0:0:8
```

The consumer threads also accumulate the total elapsed time of all the processes consumed in a shared variable.

4. module_exit

You need to make sure that no threads are waiting for semaphores. To do so, signal all the semaphores; if you expect multiple threads waiting on a semaphore; signal it multiple times.

To stop a kernel thread, you need `kthread_stop(struct task_struct *k)`. It sets kthread_should_stop for thread *k* to return true so the thread can check it (using kthread_should_stop()) and determine if it should stop.

```
// stop the kernel thread pointed by task_struct k
kthread_stop(struct task_struct *k)

// when kthread_stop() is called, this function will return true
kthread_should_stop(void)
```

Before the module exits, it should print the total elapsed time of all processes belonged to the given user in the mentioned format in the kernel log:


```
The total elapsed time of all processes for UID <UID of the user> is
<HH:MM:SS>

Example:
The total elapsed time of all processes for UID 1005 is    0:1:36
```


5. Testing

Follow the instructions here and make use of the provided scripts to test your code.

Your kernel module should meet the following three criteria.

- Module should be loaded and unloaded successfully.
- All the processes for the given user should be found by the producer and processed by the consumers, and no process should be produced or consumed more than once.
- The total elapsed time of all the processes should match the output of ps. Note that they will not exactly match due to the fact that they are not run at exactly the same time; a small difference is allowed.

Submission Requirements & Guidelines

Project 2 is **due** in four weeks by **25th October 2022 11:59 PM**. Submit the project work following the below guidelines

1. Only one submission is required per group
2. Submit the following (Maintain your code on the provided private GitHub repository in the CSE330 Operating Systems - Fall 2022 Organization)
   2.1. Source code (Only the Kernel Module. No need of submitting the Makefile, if you are using the one provided in the testing module.)
   2.2. README file, listing the following:
      a. Full names of your group members
      b. Anything that you would like the TAs to consider for grading your submission
3. **Do not** submit any other source code
4. **Do not** submit any binary
5. Create a .zip file with all your submission files. Name the zip file following the below-naming convention. **"Project-2-Group-<GroupNo>.zip"**
   Example: If you belong to Group-1, the name of your zip file should be **Project-2-Group-1.zip**
6. Individual members of the group **MUST** make reasonable commits on GitHub to make sure their individual contributions to the project are marked. **Failure to do so & you will be awarded 0-grade points for the Project.**

Bonus !!

There is an opportunity for you to earn a total of "**15**" bonus grade points in this project.

1. "**5**" points:

   a. If you submit the completed project **a week before the deadline**.
   b. If you make any submission after **"18th October 2022 at 11:59 PM"**; you are **not** eligible for these bonus points.
1. "**10**" points:
   a. If your code also passes the Bonus Test Case. (**Test Case#- 6**)
   b. Details of this test case are on the assignment rubrics on Canvas.

Policies

1. Late submissions will *absolutely not* be graded (unless you have verifiable proof of emergency). It is much better to submit partial work on time and get partial credit for your work than to submit late for no credit.
2. Every group needs to *work independently* on this exercise. We encourage high-level discussions among students to help each other understand the concepts and principles. However, a code-level discussion is prohibited and plagiarism will directly lead to failure of this course. We will use anti-plagiarism tools to detect violations of this policy.