Homework 3

W4118 Spring 2021

UPDATED: Thursday 2/18/2021 at 2:20pm EST
DUE: Monday 2/22/2021 at 11:59pm EST

All non-programming, written problems in this assignment are to be done by yourself. Group collaboration is permitted only on the kernel programming problems. All homework submissions (both individual and group) are to be made via **Git**.

You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this **file** in the top-level directory of your homework submission. Homeworks submitted without this file will not be graded. **Be aware that commits pushed after the deadline will not be considered.** Refer to the homework policy section on the **class web site** for further details.

Individual Written Problems:

The GitHub repository you will use can be initialized **here**. To clone this repository, use the following command: `git clone git@github.com:W4118/s21-hmwk3-written-UserName.git` (Replace UserName with your own GitHub username). This repository will be accessible only by you, and you will use the same SSH public/private key-pair used for homework 1.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points. Please include all your answers in a file named written.txt that is uploaded to your GitHub repository for this assignment.

1. Exercise 5.13
2. Exercise 5.14
3. Exercise 5.17
4. Exercise 5.21
5. Exercise 5.38

Group Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository for your group has been setup already on Github. You don't need the Github Classroom link for the group assignment and do not need to initialize the repository. It can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk3-teamN.git
```

(Replace `teamN` with the name of your team, e.g. `team0`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as **git-pull**, **git-merge**, and **git-fetch**.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux**

kernel coding style and check your commits with `checkpatch.pl`. Errors or warnings in your submission will cause a deduction of points.

As you have seen in homework assignment 2, the kernel maintains the state for each process and records that state in the `state` field of the `task_struct` of the process. The state indicates whether the process is runnable or running (`TASK_RUNNING`), sleeping (`TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`), stopped (`__TASK_STOPPED`), dead (`TASK_DEAD`), etc. When a process is dead, the `exit_state` field of the `task_struct` of the process indicats whether the process is zombied (`EXIT_ZOMBIE`) or really dead (`EXIT_DEAD`). For this homework, you will need to trace the state changes for processes and record them in a ring buffer. Then you will need to write a synchronization mechanism based on the ring buffer. Other than changes required in existing files in the kernel source code, your code should be implemented in a file `pstrace.c` in the kernel directory, i.e. `kernel/pstrace.c`, and a file `pstrace.h` in the kernel include directory, i.e. `include/linux/pstrace.h`.

1. **Trace the state change of processes in a ring buffer.** Write a system call that enables the tracing of a process and another system call that disables the tracing. The interfaces of these system calls are:

```
#define PSTRACE_BUF_SIZE 500    /* The maximum size of the ring buffer */

/*
 * Syscall No. 436
 * Enable the tracing for @pid. If -1 is given, trace all processes.
 */
long pstrace_enable(pid_t pid);

/*
 * Syscall No. 437
 * Disable the tracing for @pid. If -1 is given, stop tracing all processes.
*/
long pstrace_disable(pid_t pid);
```

   In addition to the global ring buffer, you should maintain a global data structure to track what processes are being traced; you can use the PSTRACE_BUF_SIZE as the maximum size required for the structure. You may not modify the `task_struct` for this assignment.

   You should implement a function that will record state changes for the respective processes and call this function from various places in the kernel to capture those state changes. The function should record the state change in a ring buffer. The interface of the function is:

```
/* The data structure used to save the traced process. */
struct pstrace {
        char comm[16];          /* The name of the process */
        pid_t pid;              /* The pid of the process */
        long state;             /* The state of the process */
}

/* Add a record of the state change into the ring buffer. */
void pstrace_add(struct task_struct *p);
```

   You should trace the following six states and record them in `pstrace.state`:

   - TASK_RUNNING
   - TASK_INTERRUPTIBLE
   - TASK_UNINTERRUPTIBLE

- __TASK_STOPPED
- EXIT_ZOMBIE
- EXIT_DEAD

For example, if a process's state changes from TASK_RUNNING to TASK_INTERRUPTIBLE, you should add a record with state TASK_INTERRUPTIBLE indicating that the state has changed to TASK_INTERRUPTIBLE. Note that this does not necessarily mean that you need to record every instance when the `state` field in the `task_struct` is modified. For example, if the Linux code changes `state` from TASK_RUNNING to TASK_INTERRUPTIBLE to TASK_RUNNING all without actually running another task, the process's state did not really change from TASK_RUNNING. A key part of this assignment is figuring out where a process's state actually changes and recording those events. You should carefully consider the discussion in class regarding the lifecycle of a process in Linux.

Since the ring buffer is shared by all CPUs, you should properly use locks to protect the ring buffer from race conditions. You should also maintain a buffer counter, which is a persistent count of the number of records that have been recorded to the ring buffer. You may find it helpful to define your own data structure for each record in the ring buffer that contains more information than the `pstrace` structure.

2. **Copy the tracing buffer into the user space.** You should write a system call that can copy the record in the ring buffer to user space. The interface of the system call is:

```
/*
 * Syscall No. 438
 *
 * Copy the pstrace ring buffer info @buf.
 * If @pid == -1, copy all records; otherwise, only copy records of @pid.
 * If @counter > 0, the caller process will wait until a full buffer can
 * be returned after record @counter (i.e. return record @counter + 1 to
 * @counter + PSTRACE_BUF_SIZE), otherwise, return immediately.
 *
 * Returns the number of records copied.
 */
long pstrace_get(pid_t pid, struct pstrace *buf, long *counter);
```

Note that if @counter is positive, your system call should sleep until a full buffer can be returned. A full buffer is when the buffer counter is equal to @counter plus `PSTRACE_BUF_SIZE`. Your system call should copy the records into @buf in chronological order such that the first entry is the entry corresponding to buffer counter @counter + 1 and the last entry is @counter + `PSTRACE_BUF_SIZE`, and should return in @counter the value of the buffer counter corresponding to the last record copied. For example, if `pstrace_get` is called with @counter=1000, it should not return until the ring buffer counter has reached 1500, and when it returns it should return the relevant buffer records from buffer counter 1001 to 1500 with @counter updated to 1500.

You should have a sychronization mechanism such that when the buffer is not full, `pstrace_get` should wait if the counter is positive; when the buffer is full for a waiting `pstrace_get`, the process calling this system call should be woken up. **You may NOT let the system call spin when the ring buffer is not full.**

You should also ensure that you account for the fact that there may be some time that elapses between when the process is woken up and when the system call gets to complete and return the records to the calling process.

You should have another system call that clears the ring buffer.

```
/*
 * Syscall No.439
 *
 * Clear the pstrace buffer. If @pid == -1, clear all records in the buffer,
 * otherwise, only clear records for the give pid.  Cleared records should
 * never be returned to pstrace_get.
 */
long pstrace_clear(pid_t pid);
```

The system call should also wake up all processes waiting on the `pstrace_get` or only those which are waiting for the pid, depending on whether `pid == -1`. The processes that are woken up should copy the relevant records in the buffer and return as opposed to waiting for their respective buffer full conditions to be met.

3. **Test your pstrace.** You should write a program that calls `pstrace` repeatedly to return the records in the buffer over time. Show how you can use the counter value so that successive calls to `pstrace` return a chronological ordering of all records, and explain any circumstances in which this may not be completely accurate. For testing purposes, you should also write another program that changes its states between running and sleeping for a certain amount of times and exits. Use the first program to trace the process of the second program. You should be able to observe how it turns from running to sleeping and finally, to zombie and exits. Your testing should generate at least one record for each of the six distinct process states we have asked you to record, and you should include the resulting output in your submission in a file `pstrace_output.txt`.