Homework 3

W4118 Fall 2023

UPDATED: Tuesday 10/10/2023 at 9:22pm EST
DUE: Wednesday 10/18/2023 at 11:59pm EST

All homework submissions are to be made via **Git**. You must submit a detailed list of references as part of 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 in the `main` branch of your team repo. **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.

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. It can be cloned using:

```
git clone git@github.com:W4118/f23-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**, **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 the checkpatch.pl (or provided run_checkpatch.sh) script on the default path in the provided VM. Errors from the script in your submission will cause a deduction of points.

For students on Arm Mac computers (e.g. with M1 or M2 CPU): if you want your submission to be built/tested for Arm, you must create and submit a file called `.armpls` in the top-level directory of your repo; feel free to use the following one-liner:

```
cd "$(git rev-parse --show-toplevel)" && touch .armpls && git add .armpls
&& git commit -m "Arm pls"
```

You should do this first so that this file is present in any code you submit for grading.

For all programming problems you should submit your source code as well as a single README file documenting your files and code for each part. Please do NOT submit kernel images. The README should explain any way in which your solution differs from what was assigned, and any assumptions you made. You are welcome to include a test run in your README showing how your system call works.**It should also state explicitly how each group member contributed to the submission and how many hours each member spent on**

the homework. The README should be placed in the top level directory of the `main` branch of your team repo (on the same level as the linux/ and user/ directories).

Assignment Overview:

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 indicates whether the process is zombied (`EXIT_ZOMBIE`) or really dead (`EXIT_DEAD`).

In this homework, you will be **tracing the state changes of processes**, and recording them in a ring buffer. A ring buffer is a **circular buffer** that has no real end. It can be represented using a fixed size array that keeps wrapping around. After writing to the last array location, new entries will start writing at the beginning of the array again, overwriting any records that were previously there. You will write a system call to copy the contents of the ring buffer to userspace so that a user program can dump the contents of the buffer. In addition, you will need to write a synchronization mechanism to copy the contents of the ring buffer to userspace at the occurrence of some future event.

Your code should be implemented in:

- A file `pstrace.c` in the kernel directory, i.e. `kernel/pstrace.c`.
- A file `pstrace.h` in the kernel uapi include directory, i.e.`include/uapi/linux/pstrace.h`.
- A file `pstrace.h` in the kernel include directory, i.e.`include/linux/pstrace.h`.
- Additional changes in existing files in the kernel source code.

You should not change any existing kernel data structures. Almost all your kernel changes should be in `pstrace.c`. `pstrace.h` should only contain the necessary data structure and macro definitions. The only changes to existing files in the kernel source code should be those required for the system call table modifications and calls to record process state changes in various kernel code paths.

PART 1: Tracing a process state change in the ring buffer

1. **System calls for enabling and disabling tracing**

   In this part, we will set up the relevant system calls to enable/disable the tracing of a process's state change. We only want to record the state changes of processes that have had tracing enabled by these system calls. The interface for these functions are:

```
/*
 * Syscall No. 451
```

```
 * Enable the tracing for @pid. If -1 is given, trace all proces
 */
long pstrace_enable(pid_t pid);

/*
 * Syscall No. 452
 * Disable tracing.
 */
long pstrace_disable();
```

In other words, your tracing system will either trace a **single** process or trace **all** processes. You should use the global ring buffer data structure provided below to keep track of what processes are being traced. Note that this system call simply *enables/disables* the tracing - the actual writing to the ring buffer will happen in a subsequent function.

A couple of requirements to keep in mind while implementing these system calls:

- Tracing a process includes tracing all threads in the same thread group.
- If the pid does not exist, an appropriate error should be returned.
- Do not modify the `task_struct` (for the entire assignment).
- If tracing is already enabled, `pstrace_enable` will replace the set of processes being traced with what is specified in the newer `pstrace_enable` call.

Tasks

1. Implement `pstrace_enable()` and `pstrace_disable()`.
2. Modify any relevant files in the kernel to reflect the newly added system calls.

2. **The global ring buffer**

Once you can successfully indicate which processes should be traced, you should set up the structure of the ring buffer. Here are some data structures that are relevant for this part of the assignment:

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

/* The data structure of the global ring buffer */
struct pstrace_kernel {
    struct pstrace entry[PSTRACE_BUF_SIZE];     /* records of st
    atomic_t counter;                           /* count number

    /* include other fields that you may need */
};

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

```
        pid_t tid;              /* tid of the thread; ie returne
};
```

You are free to modify the `pstrace_kernel` structure and you will need to add additional fields to the structure. Note that in addition to the array for the ring buffer itself, `pstrace_kernel` has a counter, which should be initialized to zero and be incremented for each record that is added to the ring buffer. It is a count of how many trace records have been ever added to the ring buffer and should never be reset.

Tasks

1. Initialize the provided data structures and any other necessary information to be used in `pstrace.h`.

3. **Recording one state change**

For now, we want to focus on tracing only one state change: when a process that is running becomes blocked but interruptible. This will result in the process state changing from `TASK_RUNNING` to `TASK_INTERRUPTIBLE`. When a process blocks in this manner, you should call `pstrace_add()` to create a new entry that records its updated state `TASK_INTERRUPTIBLE` in `pstrace.state`, and add this new entry to the global ring buffer. The interface of the function is:

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

`pstrace_add()` should only record the state change if the given process has tracing enabled. The `stateoption` field is an additional argument passed to `pstrace_add()` that you can use as you see fit. For example, you could use the argument to pass in the state that you want recorded or any other information that would be helpful, but you do not have to use the field for this part of the assignment.

Since the ring buffer is shared by all CPUs, multiple state changes may occur simultaneously. You will need to define a lock to protect the ring buffer from race conditions.

Your job is to record actual changes in process state, not necessarily whenever the `__state` field is changed. For example, if the Linux code changes `__state` from `TASK_RUNNING` to `TASK_INTERRUPTIBLE` to `TASK_RUNNING` all without actually running another process, 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.

Tasks

1. Implement `pstrace_add()` to track process state changes from `TASK_RUNNING` to `TASK_INTERRUPTIBLE`.

2. Modify any relevant files in the kernel to record this process state change.

4. **Return a copy of ring buffer to userspace**

   At this point, we've allowed users to enable/disable the tracing of processes, and have written the state change from `TASK_RUNNING` to `TASK_INTERRUPTIBLE` into a buffer located in kernel space. Now, we want to be able to access these records in userspace. Write a system call that can copy the records in the ring buffer to userspace. The interface of the system call is:

```
/*
 * Syscall No. 453
 *
 * Copy the pstrace ring buffer into @buf.
 * If @counter == 0, copy the pstrace ring buffer.
 * If @counter < 0, return an error.
 *
 * Returns the number of records copied.
 */
long pstrace_get(struct pstrace *buf, long *counter);
```

   If @counter == 0, copy the current state of the pstrace global ring buffer into @buf and set @counter to the value of the buffer counter corresponding to the last record copied. If @counter < 0, it is invalid and so return an error. Later in the assignment, you will make use of the counter argument more extensively, but for now, just have your system call handle these two cases.

   In addition, you should have another system call that clears the ring buffer.

```
/*
 * Syscall No.454
 *
 * Clear the pstrace buffer. Cleared records should
 * never be returned to pstrace_get. Clear does not
 * reset the value of the buffer counter.
 */
long pstrace_clear();
```

   Cleared records should no longer be able to be returned to future `pstrace_get` calls. Do not reset the ring buffer counter value.

   Tasks

   1. Implement `pstrace_get()` and `pstrace_clear()`.

   You will be required to write a user test in part 4, but it may be a good idea to test your current basic pstrace implementation. Make sure that you can record this one state change and return the immediate buffer results to userspace before moving on to the next sections.

PART 2: Record additional state changes

Extend the usage of `pstrace_add()` to now record changes in processes' state throughout the kernel. Specifically, the following seven states should be traced and recorded in `pstrace.state`:

1. `TASK_RUNNING`
2. `TASK_RUNNABLE`
3. `TASK_INTERRUPTIBLE`
4. `TASK_UNINTERRUPTIBLE`
5. `_TASK_STOPPED`
6. `EXIT_ZOMBIE`
7. `EXIT_DEAD`

Note that you should represent a task that is on the run queue as `TASK_RUNNABLE` while a task that is actually running on the CPU should be `TASK_RUNNING`. However, Linux does not have an actual state `TASK_RUNNABLE`. It denotes both the state of being on the run queue and actually running as `TASK_RUNNING`. To do this, you should introduce a `TASK_RUNNABLE` state for tracing purposes only (i.e. `TASK_RUNNABLE` is not stored in the actual __state field of the task_struct). **You should define** `TASK_RUNNABLE` **to have a value of 3.** Note that since `TASK_RUNNABLE` is not an actual value stored in the `__state` field of the `task_struct`, you will find it useful to use the `stateoption` argument of `pstrace_add()` to record that state change.

Again, your job is to record actual changes in process state, not necessarily whenever the `__state` field is changed. You should identify where a process's state actually changes and record those events as opposed to trying to insert `pstrace_add()` calls whenever you find the `__state` field is being modified. You should minimize the number of `pstrace_add()` calls required to trace the seven states.

You will also need to carefully consider the code paths in which you insert `pstrace_add()` when doing locking, or you may deadlock your system. In particular, you should ask yourself whether the code path in which you insert `pstrace_add()` could be executed as a result of an interrupt and what implications that may have on the specific locking primitives you use.

Tasks

1. Update `pstrace_add()` to now track all specified process state changes.
2. Modify any relevant files in the kernel where these changes occur.

PART 3: Waiting to copy the tracing buffer into userspace

You will now update the implementation of your `pstrace_get` system call to support waiting for the ring buffer counter to reach some value before copying the contents of the ring buffer to userspace. The interface to `pstrace_get` remains

the same, but the system call should now provide new functionality for the case when the counter argument is positive:

```
/*
 * Syscall No. 453
 *
 * Copy the pstrace ring buffer into @buf.
 * If @counter == 0, copy the immediate pstrace ring buffer.
 * If @counter < 0, return an error.
 * If @counter > 0, the caller process will wait until a full buff
 * be returned after record @counter (i.e. copy records @counter +
 * @counter + PSTRACE_BUF_SIZE from pstrace ring buffer to @buf).
 *
 * Returns the number of records copied.
 */
long pstrace_get(struct pstrace *buf, long *counter);
```

Specifically, a positive @counter value indicates a request for a full buffer starting at ring buffer counter @counter + 1. That is, your system call should copy records into @buf in chronological order such that the first record is the record corresponding to ring buffer counter @counter + 1 and the last record as @counter + `PSTRACE_BUF_SIZE`. If the ring buffer counter has not reached the last record, your system call should sleep until the full buffer can be returned. Update @counter to the value of the ring buffer counter corresponding to the last record that was requested.

- For example, if `pstrace_get` is called with @counter = 1000, it should not return until the ring buffer counter has reached 1500. When it returns, it should return the relevant corresponding buffer records from buffer counter 1001 to 1500 with @counter updated to 1500.

If the ring buffer counter has already exceeded the counter for the last record that is requested, less than a full buffer may be copied.

- For example, if `pstrace_get` is called with @counter = 800, then the last record that is requested should be the record corresponding to the counter 1300. However, if the ring buffer counter is already at 1400, then it means that it only contains the previous 500 records (corresponding to counters 901 to 1400). Therefore, it will only return 400 buffer records (corresponding to counters 901 to 1300) instead of a full buffer since those are the only relevant records within the requested range that are available.

If there are no records that can be copied, that is if @counter + `PSTRACE_BUF_SIZE` is less than the current ring buffer counter - `PSTRACE_BUF_SIZE`, set @counter to be the current value of the ring buffer counter and return immediately. Do not copy any records from the ring buffer in this case.

- For example, if `pstrace_get` is called with @counter = 100 when the ring buffer counter is already at 1400, no records can be copied and @counter is updated to 1400.

Hints

1. You will need to implement an appropriate synchronization mechanism to wake up waiting processes that called `pstrace_get` when their desired counter condition has been met. Waiting processes should be blocked with state `TASK_INTERRUPTIBLE`. **You may NOT let the system call spin when the counter condition has not been met.**
2. You should plan to use **Linux wait queues**. A wait queue is a list of processes waiting for some event to occur. The functions that use wait queues are designed to either wake up a single process or all processes on a wait queue. If the desired functionality is for processes to wait for different events to occur, the typical approach is to use a separate wait queue for each event. The functions that use wait queues are not designed to search a wait queue to identify processes waiting for different events to occur.
3. You should think carefully about how and when you will copy the contents of the ring buffer in response to a `pstrace_get` to provide the expected functionality. In particular, there may be an arbitrary amount of time that elapses between when a process is woken up and when the process is actually run to complete the system call and return the records to the calling process. Be sure to return the correct records requested when possible.
4. You should also think carefully about what happens if you are tracing a process that is waiting on a `pstrace_get` call to ensure that you do not deadlock your system. How does that process wake up? What trace records are generated as a result of waking up that process? What locks are held when you are recording trace records?
5. Also make sure that you do not deadlock your system when handling wait queues and interrupts.

If the buffer provided for copying into is not large enough to hold the records requested, you should return an error.

A process that is sent a signal while it is waiting on `pstrace_get` should be woken up and return an appropriate error.

`pstrace_clear()` should also wake up all processes waiting on the `pstrace_get`. The processes that are woken up should copy any relevant records currently in the buffer and return as opposed to waiting for their respective buffer full conditions to be met. @counter should still remain as the value of the last record that was originally requested.

Tasks

1. Update `pstrace_get()` to support positive values of @counter.
2. Update `pstrace_clear()` to wake up any waiting processes.

PART 4: 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. The program

should be in the `user` directory of your team repo, and your Makefile should generate an executable named **test**.

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. This should be done by modifying **test** so that it forks a child process that execs the second program. In other words, we should be able to see the results of your second program by simply doing: `make ./test`.

You should be able to observe how the second program turns from running to sleeping and finally, to zombie and exits. Your testing should generate at least one record for each of the seven 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`.

PART 5: Process Lifecycle

Write answers to the following questions in the `user/part5.txt` text file, following the provided template **exactly**. Make sure to include any references you use in your `references.txt` file, and that you are referencing the correct kernel version in bootlin (**v6.1.11**). For reference, the URLs you answer with should be in the following format:
https://elixir.bootlin.com/linux/v6.1.11/source/kernel/sched/core.c#L6432

1. Give each bootlin URL that shows the file and line number in which you inserted in a `pstrace_add` call to trace the state change to:
   a. `TASK_RUNNING`
   b. `TASK_RUNNABLE`
   c. `TASK_INTERRUPTIBLE`
   d. `TASK_UNINTERRUPTIBLE`
   e. `_TASK_STOPPED`
   f. `EXIT_ZOMBIE`
   g. `EXIT_DEAD`

   For the following questions, replace STATE with the relevant state. For example:

   ```
   [TASK_RUNNING] -> [TASK_INTERRUPTIBLE]
   ```

   Do NOT remove the brackets around the state names.

2. Which state change(s) can be directly caused by interrupts (i.e. an interrupt handler will change the `__state` field)?

3. Which state change(s) can only be caused by the running process itself (i.e. current will change its own `__state` field)?
4. Which state change(s) always involves or results in the current running process calling schedule?
5. There are 3 tasks with respective pid values 1150, 1152, and 1154 and they run in the following order on a CPU: pid 1150, 1152, 1154, 1152, 1150, 1154. In `__schedule`, there are two variables, `prev` and `next`.
   a. When each task starts running, what will be the respective PIDs for prev and next? For any prev or next that cannot be known from the information provided, indicate unknown.
   b. When each task stops running, what will be the respective PIDs for prev and next? For any prev or next that cannot be known from the information provided, indicate unknown.

Submission Checklist

Include the following in your main branch. Only include source code (ie *.c,*.h) and text files, do **not** include compiled objects.

- `.armpls` file for teams with M1/M2 CPUs
- `README` file
- `references.txt` file
- Implementation of system calls and additional functions added to `linux/kernel/pstrace.c`
- Implementation of `test.c` in `user/part4`
- Output of `test.c` in `user/part4/pstrace_output.txt`
- Answers to written questions in `user/part5.txt`