Homework 2

W4118 Fall 2023

UPDATED: Wednesday 9/27/2023 at 10:02pm EST
DUE: Wednesday 10/4/2023 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-hmwk2-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.

The kernel programming for this assignment will be run using your Linux VM. As a part of this assignment, you will be experimenting with Linux platforms and gaining familiarity with the development environment. Linux platforms can run on many different architectures, but the specific platforms we will be targeting are the X86_64 or Arm64 CPU families. All of your kernel builds will be done in the same Linux VM from homework 1. You will be developing with the Linux 6.1.11 kernel.

For this assignment, you will write a system call to dump the process tree and a user space program to use the system call.

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 much time 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).

---

1. **Build your own Linux 6.1.11 kernel and install and run it in your Linux VM.**

   Kernel building instructions for VM

   Build and run a custom kernel for your VM. The source code for the kernel you will use is located in the `linux/` folder in the `main` branch of your team repo. You can checkout to that branch. Clone the `main` branch in a separate directory–the `linux/` folder within this directory will be the root of your kernel tree.

   Ensure you have the following dependencies before you begin:

   ```
   build-essential bc python3 bison flex rsync
   libelf-dev libssl-dev libncurses-dev dwarves
   ```

   1. In your repo, run `git checkout main` to switch to the "main" branch.
   2. The first thing you need to do is make the config file for the VM kernel, which means that you will create a `.config` file in the root of your kernel tree that has the appropriate configuration options set for the kernel you are going to build. **This guide** provides detailed instructions for you to follow to create the `.config`, though you should replace wherever it refers to the kernel version as 5.10.58 with 6.1.11. As described in the guide, the two steps of the process will be executing the following commands in the root directory of your kernel tree:

      ```
      make olddefconfig
      make menuconfig
      ```

   3. Run `make -j$(nproc)`(or `make -jN`, where N is the number of parallel compilation jobs to run, which should correspond to the number of cores in your VM). Wait for the kernel to compile.
   4. Install the new kernel by running:

```
sudo make modules_install && sudo make install
```

5. Reboot your VM.
6. In the boot selection screen (called `grub`), which shows up immediately after the VMWare logo screen, select "Advanced options for Ubuntu GNU/Linux" and choose the kernel identified by "cs4118" (the `LOCALVERSION` identifier you set in `.config`).
7. You are now running your custom kernel! You can use `uname -a` to check the version of the kernel. You now have an unmodified kernel from the 6.1.11 Linux source provided in the `main` repo. You should name it 6.1.11-cs4118, and keep it around as your fallback kernel for all future assignments (including this one), in case you run into any trouble booting into the kernel you're working on. Additionally, make sure that the `CONFIG_BLK_DEV_LOOP` option is set to `y` in your `.config` file before you build and install your unmodified kernel. This will come in handy in later assignments.
8. Optimize your kernel compile time. A large amount of time is spent compiling and installing kernel modules you never use. To reduce your kernel compilation time, you can optionally regenerate a `.config` so that it only contains modules you are using by following these instructions:
   - Backup your `.config` to something like `.config.[UNI]-from-lts`. Make sure to keep your local version the same; that is, your kernel should still be named 6.1.11-cs4118.
   - Run `make localmodconfig` in your Linux kernel source tree. This will take your current `.config` and turn off all modules that you are not using. It will ask you a few questions. You can hit ENTER to accept the defaults, or just have yes do so for you:

     ```
     $ yes '' | make localmodconfig
     ```

     Make sure that `CONFIG_BLK_DEV_LOOP` is still set to `y` before building and installing this kernel. Now you have a much smaller `.config`. You can follow the rest of the steps starting from make (step 3). Note that you only need to do `make localmodconfig` once, not each time you build the kernel.
   - When you are hacking kernel code, you'll often make simple changes to only a handful of .c files. If you didn't touch any header files, the modules will not be rebuilt when you run make; thus there is no reason to reinstall all modules every time you rebuild your kernel. In other words, in lieu of step 4, you can just do:

     ```
     sudo make install
     ```

     assuming you have already done step 4 with the kernel configuration you are using.

Just to reemphasize the earlier point, when you are hacking kernel code, the standard workflow will be to modify kernel code, then build the kernel and install the updated kernel using the following two steps:

```
make -j$(nproc)
sudo make install
```

Then reboot your VM and select your kernel in `grub`. In other words, no need to do step 2 or step 8 each time you build your kernel; those steps only need to be done once.

2. **Write a new system call in Linux**

   General Description

   The system call you write will retrieve information from each thread associated with each process in some subset of the process tree. That thread information will be stored in a buffer and copied to userspace.

   Within the buffer, thread information should be sorted first in breadth-first-search (BFS) order of the process (main threads) within the process tree, and then between the (non-main) threads associated with each process, sorted in ascending order of pid. You may ignore PID rollover for the purposes of ordering threads. See 'additional requirements' for an example ordering.

   The prototype for your system call will be:

```
int ptree(struct tskinfo *buf, int *nr, int root_pid);
```

   You should define `struct tskinfo` as:

```
struct tskinfo {
    pid_t pid;              /* process id */
    pid_t tgid;             /* thread group id */
    pid_t parent_pid;       /* process id of parent */
    int level;              /* level of this process in the sub
    char comm[16];          /* name of program executed */
    unsigned long userpc;   /* pc/ip when task returns to user
    unsigned long kernelpc; /* pc/ip when task is run by schedu
};
```

   in `include/uapi/linux/tskinfo.h` as part of your solution. The uapi directory contains the user space API of the kernel, which should include structures used as arguments for system calls. As an example, you may find it helpful to look at how `struct tms` is defined and used in the kernel, which is another structure that is part of the user space API and is used for getting process times.

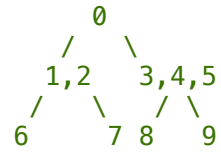
   Parameters description

   - **buf** should point to a buffer to store the thread data from the process tree. The thread data stored inside the buffer should be sorted first in BFS order: processes at a higher level (level 0 is considered to be higher than level 10) should appear before processes at a lower level. Then, when listing threads within a process, the threads should be stored in ascending order of pid.
   - **nr** points to an integer that represents the size of this buffer (number of entries). The system call copies at most *nr entries of thread data to the buffer, and stores the number of entries actually copied in `nr`.

- **root_pid** represents the pid of the root of the subtree you are required to traverse. Note that information of nodes outside of this subtree shouldn't be put into buf. If the root pid contains multiple threads, they all should be included in `buf` (unless the maximum number of entries is reached).
- **Return value:** The function defining your system call should return 0 on success, and return an appropriate error on failure such that `errno` contains the correct error code.

Additional Requirements

- You should fill the buffer when traversing the tree primarily in **BFS order** of containing process, then for each thread within a process in **ascending order of pid** until either every thread is stored, or the number of threads reaches `nr`. E.g. if we have a tree as below:



and you are given a buffer of size `10 * sizeof(struct tskinfo)`, the buffer should be filled as follows (excluding program counters):

```
[
tskinfo {pid=0, tgid=0, ppid=0, level=0, … },
tskinfo {pid=1, tgid=1, ppid=0, level=1, … },
tskinfo {pid=2, tgid=1, ppid=0, level=1, … },
tskinfo {pid=3, tgid=3, ppid=0, level=1, … },
tskinfo {pid=4, tgid=3, ppid=0, level=1, … },
tskinfo {pid=5, tgid=3, ppid=0, level=1, … },
tskinfo {pid=6, tgid=6, ppid=1, level=2, … },
tskinfo {pid=7, tgid=7, ppid=2, level=2, … };
tskinfo {pid=8, tgid=8, ppid=4, level=2, … };
tskinfo {pid=9, tgid=9, ppid=4, level=2, … };
]
```

Note that in this example, there are 7 processes (0, 1, 3, 6, 7, 8, and 9) and 3 threads (2, 4, and 5). Also, 7 is a child of 2 not 1, and 8 and 9 are children of 4. In this scenario, commands like `ps` might differ from ours in output (see the hint below for an explanation).

- There should be no duplicate information of processes inside the buffer.
- If a value to be set in `tskinfo` is accessible through a pointer which is null, set the value in `tskinfo` to 0.
- Your system call should be assigned the number **451** and be implemented in a file `ptree.c` in the kernel directory, i.e. `linux/kernel/ptree.c` in your repository. Note that you will need to modify the appropriate kernel Makefile so that it is aware of your new source code file or it will not know to compile your source code file with the rest of the kernel code.

- Retrieving some of the information for your `struct tskinfo` will be architecture-specific (i.e. it will be implemented differently depending on whether your platform is x86-64 or ARM64). You should call generic functions from your main `linux/kernel/ptree.c` file for retrieving these values, but implement them in `linux/arch/[x86 or arm64]/kernel/ptree.c` (and make sure you modify the Makefile in the same folder appropriately). This ensures that only the correct retrieval function is included when your kernel is compiled. Although you are only required to complete solutions for one architecture, defining these functions in both `arch/x86` and `arch/arm64` will allow you to develop alongside students with a different platform.
- Your algorithm shouldn't use recursion since the size of the function stack in the kernel is quite small, typically only 8KB.
- Your code should handle errors that could occur. For example, some error numbers your system call should detect and return include:
  - `-EINVAL`: if `buf` or `nr` are null, or if the number of entries is less than 1
  - `-EFAULT`: if `buf` or `nr` are outside the accessible address space.

Hints

- This syscall implementation has a few different components. We recommend that you approach the problem in steps, doing incremental development and test to check the correctness of the functionality along the way:
  1. To understand how to fill a `struct tskinfo` with the relevant information, you should first write a simple version of the system call which returns the information for one process only, given its pid.
  2. Now expand part 1 to allow listing all/any process in the process tree (not including threads). This will help you understand how to implement BFS over the process tree without worrying about the nuances of listing threads versus processes.
  3. Finally, write the system call that works for both processes and threads. Think carefully about how threads intersect with the process tree, and how you must modify your BFS algorithm to include them. You may find **this blog post** helpful for relating processes and threads.
- Linux maintains a list of all processes in a doubly linked list. Each entry in this list is a `task_struct` defined in **include/linux/sched.h**. You may find these three functions/macros (as well as others you run across) in the Linux kernel useful for translating between pids and task structs, and walking the list of processes to find your root `task_struct`:
  - `for_each_process` to walk through the list of processes (and `for_each_thread` to walk through the list of threads)
  - `task_pid_nr` to translate from a `task_struct` to its pid
  - `task_tgid_nr` to translate from a `task_struct` to its tgid
- When traversing the process tree data structures, it is necessary to prevent the data structures from changing to ensure consistency. For this purpose

the kernel relies on RCU, which involves calling the `rcu_read_lock()` and `rcu_read_unlock()` primitives before you begin traversal and only released after the traversal is completed. You do not need to worry about the details of how RCU works for now, but between the calls to `rcu_read_lock()` and `rcu_read_unlock()`, your code must not perform any operations that may block the executing process, such as memory allocation, copying of data into and out from the kernel etc. Your code should look something like:

```
rcu_read_lock();
do_some_work();
rcu_read_unlock();
```

If your code needs to do memory allocation or copying of data into and out from the kernel, such code should be before `rcu_read_lock()` or after `rcu_read_unlock()`.

- You can use the `ps -eLf` command to display a list of current threads. However, when using this and other user commands that reference threads, you might notice that they list all the threads within a process as having the same "PID", and differentiate those threads with some other field (maybe called TID or LWP). This can be confusing, because this is **not** how the kernel represents threads internally. In the kernel, threads have unique pids, and use a thread group id (TGID) to identify which process the thread is associated with. So why do some user programs identify threads differently? It has to do with historical differences between how the POSIX standard defines threads/processes, and how the Linux kernel implements them. See **this stackoverflow answer** for more information on how these two representations differ. In your program, you should return the **Linux kernel representation** of pid, so each thread `tskinfo` struct you return should have a unique pid.
  - If you're interested, **this (optional) paper from 2002** provides some historical context for the theoretical differences between POSIX and Linux when it comes to threads/processes. Note that it is quite old, so use it as historical context and **not** a reference for programming.
- To learn about system calls, you may find it helpful to search the Linux kernel for other system calls and see how they are defined. Take a look at at **include/linux/syscalls.h** and **kernel/sys.c**.
- Your system call should not blindly trust the arguments that are passed in, especially pointers. You should make use of functions such as `copy_from_user`, `copy_to_user`, `get_user`, and `put_user`.
- The kernel provides various data structures as discussed in your reading, especially Ch 6 of the Linux Kernel Development book. In addition to the kernel implementation of linked lists, you may also find the kernel implementation of queues helpful.
- Remember that the standard C library is not available in the kernel, but the kernel often provides similar functionality. For memory allocation, instead of `malloc` and `free`, you should use the **kernel memory allocation** functions, `kmalloc` and `kfree`; the former requires a GFP flag, which should be `GFP_KERNEL` for normal memory allocation. For debugging purposes, instead of `printf`, you will find it helpful to use **printk**, a robust mechanism

to print information that is designed to be callable from almost any C code in the kernel. However, do not leave extraneous debugging messages enabled in your code when submitting your homework.

3. **Test your new system call**

   General Description

   Write a simple C program which calls `ptree` with the root_pid as an argument. If no argument is provided, your program should return the entire process tree. The program should be in the `user/part3` folder of your team repo, and your makefile should generate an executable named **test**.

   Since you do not know the tree size in advance, you should start with some reasonable buffer size for calling `ptree`, then if the buffer size is not sufficient for storing the tree, repeatedly double the buffer size and call `ptree` until you have captured the full process tree requested. Print the contents of the buffer from index 0 to the end. For each process, you must use the following format for program output:

```
printf("%s,%d,%d,%d,%p,%p,%d\n", buf[i].comm, buf[i].pid, buf[i
    buf[i].parent_pid, (void *)buf[i].userpc, (void *)buf[i].ke
```

   Example program output (yours will likely be different depending on the processes running in your VM):

```
$ ./test
swapper/0,0,0,0,(nil),0xffffffff8e7a4281,0
systemd,1,1,0,0x7f8321f27c46,0xffffffff8d7f8b52,1
kthreadd,2,2,0,(nil),0xffffffff8d7f8b52,1
systemd-journal,299,299,1,0x7f6de3927c46,0xffffffff8d7f8b52,2
…
vmware-vmblock-,322,322,1,0x7f30f6c1d4a6,0xffffffff8d7f8b52,2
vmware-vmblock-,324,322,1,0x7f30f6c801bd,0xffffffff8d7f8b52,2
vmware-vmblock-,325,322,1,0x7f30f6c801bd,0xffffffff8d7f8b52,2
…
test,1558,1558,950,0x7f90f7be8539,0xcc0,3
```

   Hints

   - The `ps` command in the VM will help in verifying the accuracy of information printed by your program. In particular, you can use `ps -eLf` (however, see the note from part 2 about why this returns a "PID" for threads that is different from the kernel's idea of a pid).
   - Although system calls are generally accessed through a library (libc), your test program should access your system call directly. This is accomplished by utilizing the generall purpose `syscall(2)` system call. You can consult its man page for more details: `man 2 syscall`.

4. **Investigate Kernel Source Code**

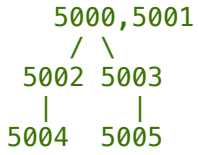
Write answers to the following questions in the `user/part4.txt` text file, following the provided template **exactly**. Make sure to include any references you use in your `references.txt` file.

1. There are a few PIDs that are reserved for system processes and kernel threads. These include PIDs 0, 1, and 2. What is the name associated with each of these three PIDs?
2. Give the exact URL on **elixir.bootlin.com** pointing to the file and line number of code at which the data structure describing the process with PID 0 is defined. Note: make sure you use v6.1.11.
3. Give the exact URL on **elixir.bootlin.com** pointing to the file and line number of code at which the function that executes instructions to context switch from one task to another is defined. The function you identify, which may be in assembly code, should be the one that contains the actual instruction that switches the CPU's program counter register to the task so it can run. Please provide an answer for both arm64 and x86-64. Note: use v6.1.11.
4. Give the exact URL on **elixir.bootlin.com** pointing to the file and line number of code at which the process with PID 1 starts running as the currently running process. Please provide an answer for both arm64 and x86-64. Note: use 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**

5. **Create your own process tree**

Using the program you developed in part 3, write another program such that you can use the program from part 3 to output the following process tree:



With `foo` running in another shell, the console output for running `./test 5000` should be:

```
$./test 5000
foo,5000,5000,1,x,y,0
foo,5001,5000,1,x,y,0
foo,5002,5002,5000,x,y,1
foo,5003,5003,5000,x,y,1
foo,5004,5004,5002,x,y,2
foo,5005,5005,5003,x,y,2
```

where `x` represents the `userpc`, and `x` represents the `kernelpc`. Any valid values of `x` and `y` are okay. Otherwise, all of the other fields shown in the output above should **exactly** match the strings and integers shown.

This program should be in the `user/part5` directory of your team repo, and your `Makefile` should generate the `foo` executable. While we will be testing your code on a freshly booted system, you may find it helpful to **change the maximum possible pid value** to make it easier to test your program (this allows pid values to rollover more quickly). For example, using the following commands may be helpful: `echo 10000 | sudo tee /proc/sys/kernel/pid_max`

Hints

- To create threads, you should use the pthreads **pthread_create()** call.
- As with part 2, it may be helpful to break the problem into pieces:

1. First, write a program that does not involve creating threads to better understand when/where to fork. That program should produce the following process tree:

```
  5000
  /  \
5001 5002
 |    |
5003 5004
```

2. Once you have this intermediate program, you should be able to make the target process tree easily by calling pthread_create() from a certain process.

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 ptree system call added to `linux/kernel/`
- Implementation of `test.c` in `user/part3`
- Answers to written questions in `user/part4.txt`
- Implementation of `foo.c` in `user/part5`