CSE 422S – Spring 2021 Exam 1

Released 4pm Thursday, February 18th, 2021

Name (please print):

Tara Renduchintala

This exam focuses on concepts and details we have covered in the assigned readings, lectures, and studios. Please answer each question as completely and correctly as possible. Partial credit may be given for incorrect answers that show understanding of the material.

During the exam you may use your notes, text books, and on-line sources of information, but you may not post content from this exam anywhere at any time. Communicating or sharing materials with other people regarding the exam is not permitted.

Please sign or print your name below to indicate your understanding of, and agreement to abide by, the exam conditions described above:

Tara Renduchintala

Exam scoring (to be completed by the grader)

Question	Possible	Score
1	8	
2	12	
3	12	
4	12	
5	8	
6	12	
7	12+1	
8	12	
9	12	
Total	100+1	

1. (8 points) Briefly, how does **`strace`** allow a user space process (the tracer) to inspect system calls made by another user space process (the tracee)?

strace outputs a line for each interaction with the kernel that the application had. Each line outputted corresponds to a system call that is made by the program. It prints the system call name, the parameters and the value. Thus, the tracer can examine all the system calls made by the process and understand what was passed and what was returned. One also has the ability to see which user started and owns the process.

Briefly, why does using **`strace`** to allow system calls to be inspected in that way distort those programs' timing?

The timing of the programs gets distorted by *strace* because *strace* adds a lot of overhead due to the fact that it has a few system calls itself for each system call that is traced. Therefore, the time that is reported by *strace* will result in a much larger time than the process alone would report.

Briefly, what is meant by a "trace point" in **`ftrace`**?

A "tracepoint" is the static target of *ftrace* that are considered events within the kernel. Tracepoints can be things such as interrupts, power state transition, scheduling events, etc. A tracepoint allows the user to trace internal operation of the kernel.

What bad thing can happen if the **`kthread`** associated with a CPU's ring buffer of **`ftrace`** events can't empty it as fast as events are being recorded into it?

If the ring-buffer isn't emptying fast enough and it gets nearly full, then there is a potential for the tracing information that is being written could be overwriting the data in the head page which could thereby corrupt the data that is being read. This results in data being "dropped".

2. (12 points) Briefly, how does the CLOCK_REALTIME clock differ from the CLOCK_MONOTONIC clock?

The CLOCK_REALTIME clock is a system wide clock that measures and keeps track of the wall-clock time. The CLOCK_REALTIME clock can be changed manually. The CLOCK_MONOTONIC clock is based upon an arbitrary and unspecified point in the past that doesn't change once the system starts up. The CLOCK_MONOTONIC is used for applications that are reliant on continuous time keeping (i.e., processes that are sensitive to discontinuous changes to the system clock).

Briefly, how does the CLOCK_MONOTONIC_COARSE clock differ from the CLOCK_MONOTONIC clock?

The CLOCK_MONOTONIC_COARSE takes the time less frequently, therefore it has less precision. The COARSE clock is more efficient however to gain that efficiency, the program is sacrificing precision. The CLOCK_MONOTONIC takes time more frequently, thus producing a more precise output.

Briefly, how does the CLOCK_MONOTONIC_RAW clock differ from the CLOCK_MONOTONIC clock?

The CLOCK_MONOTONIC_RAW has access to a raw hardware-based time that is not subject to network time protocol (NTP) adjustments. It does not change in frequency as the NTP understands the error between the local oscillator and online servers. The CLOCK_MONOTONIC changes in frequency in accordance to the time adjustments of the NTP. It corrects the imprecision in the hardware clock rate.

Briefly, how does the CLOCK_PROCESS_CPUTIME_ID clock differ from the CLOCK_MONOTONIC clock?

While the CLOCK_MONOTONIC measures relative real time, the CLOCK_PROCESS_CPUTIME_ID measures the amount of CPU time a process consumes. The CLOCK_PROCESS_CPUTIME_ID is specifically for measuring CPU time whereas CLOCK_MONOTONIC is a general clock used for tracking time.

Briefly, what is meant by relative time and how does it differ from absolute time?

Relative time is the measure of time from one point to another. For example, “5 seconds from now” is relative because the actual wall-clock time is irrelevant. The system only needs to keep track of now and count 5 seconds, it does not need the concept of absolute time. However, absolute time needs to manage the current time of day as well as maintain a measurement of it.

Briefly, how could a periodic timer expiring more frequently decrease the amount of a work a process can accomplish in a given time?

A periodic timer expiring more frequently could decrease the amount of work a process could accomplish because when the timer expires, the timer is then destroyed and the process that is tied to the timer may be paused and have to be rescheduled to a different timer once it is created. Therefore, when a periodic timer expires more frequently, the interrupt stops the current process more frequently, therefore, doing less work.

3) (12 points) Often, when we talk about “compiling” a program, we use both a compiler and a linker. Briefly, describe the differences between the two.

A compiler takes the source code and creates an object code file. The linker then takes those generated object code files and combines them into an executable file.

Briefly, describe one reason it might be beneficial to perform *static* linking and one reason it might be beneficial to perform *dynamic* linking.

It may be beneficial to preform static linking when there is a reasonable suspicion that the application or program would be at risk if a library upgrade introduces a bug to the application, thereby breaking the application. By statically linking the libraries, the code that the application requires to run correctly would be unaffected by changes in any of the shared libraries.

It might be beneficial to use dynamic linking to save memory space as only the element that is actually needed gets mapped in. It is also more efficient as multiple programs can reference the same library, which also results in memory preservation (when statically linked, each program would have a copy of the whole library which takes up a lot of memory space.

For a process to create a new process that executes a different executable, it must make two system calls. The first is fork(); what is the second?

The second call exec() [or execve()], and it reads the program into memory and begins executing.

After a process calls fork(), how can the program distinguish the parent from the child process?

The program can distinguish the parent from the child process by the value that is returned from fork(). The parent returns the PID of the child process, while the child returns 0 on success. If the child obtains its own PID, the two processes can be distinguished via their PIDs. The child’s PPID (parent PID) will be that of the parent process that it was derived from.

After a process calls fork(), explain one benefit of duplicating file descriptors from the parent to the child process?

One benefit of duplicating and sharing file descriptors is that if both the parent and child are writing to a file, the sharing allows the two processes to share the file offset. This way, the two processes don't overwrite the output of each other.

After a process calls fork(), the kernel optimizes the copying of memory from the parent to the child process. Name this optimization, and describe briefly what it is.

The optimization is called "copy-on-write". When fork() is first called, the kernel sets the child and parent up such that both of the page table entries point to the same physical memory pages. However, the pages are marked read-only. When a parent or child try to modify one of these read-only pages, the kernel catches the modification and makes a duplicate copy of the page that is about to be modified. From here, the parent and the child will have their own private copies of the page which they will modify. The parent and child do not have visibility into each other's new private page.

4) (12 points) If a kernel module's `init()` function initializes and starts a timer, what must the module's `exit()` function then also be sure to do?

Since exit functions are responsible for undoing what the init function did, it must stop the timer and free the resources used (de-initialize the timer).

Briefly, why must a kernel module that takes input parameters be able to use default values for them?

A kernel module must be able to use default values because it needs to be programmed defensively. If no values are passed in, then the default parameters can be used and are always instantiated. Therefore, if a parameter that is passed in is not of the right type or the number of arguments doesn't match, the module can still work using the defaults.

Briefly, why could `sudo rmmod` fail on your Pi?

sudo rrmod could fail because the module isn't loaded. It cannot unload a module that is not already loaded.

Briefly, why could `sudo insmod` fail on your Pi?

sudo insmod could fail because it is already loaded. It cannot load a module that is already loaded.

Briefly, what does `sudo dmesg --clear` do?

sudo dmesg –clear clears the contents of the system log.

Briefly, what kind of information is indicated by the first part of a call to **`printk`**, e.g., **`KERN_WARNING`** or **`KERN_ALERT`**?

The first part of a call to printk(…) is known as the log level. The log level specifics the importance of a message. That way, depending on the importance, the kernel can decide whether to show the message immediately or not (depending on the priority of the message). Log levels have priority levels associated to them.

5) (8 points) Please give the name of one of the kernel threads that the init process creates, and describe briefly what that thread does.

One of the kernel threads created by the init process is the *migrate* thread. It distributes the threads to other cores so that one core is not overloaded. It essentially rebalances the work among the different cores.

Briefly, why are any floating point errors or segmentation violations that are caused by a user space thread handled synchronously with respect to that thread's flow of execution?

When a process violates the rules that are in place (i.e., memory protection rules) the CPU suspends the process and generates an interrupt that is sent to the kernel. These interrupts are received by the kernel and lead to a signal being sent by the kernel to a process such that the signal handler terminates the process. Thus, it is important that the threads flow of execution is synchronous such that the process gets terminated in a timely manner so that more errors aren't generated. Furthermore, it is important so that no child processes are because once the parent processes terminates, those children will zombies.

Briefly, what does a kernel thread always lack, which an otherwise comparable user space thread always has?

The kernel thread does not have any process memory space while the user space has an address space for the process that's running. Therefore, anything that kernel threads access has to be put in a location that they are given access to, or the data might be part of the data structures that the kernel already maintains.

Briefly, what does issuing the command **`ps aux`** in a Linux terminal window tell you about that system?

ps aux lists all the processes and for all the users while displaying the owner/user for each process. It also shows processes that are not attached to a terminal.

6) (12 points) Next to each of the statements below, please write T next to it if it is true or write F next to it if it is false.

T ---- Different definition macros must be used for syscalls that take different numbers of arguments.

T ---- Process identifiers 0 and 1 are for idle and init.

F ---- The Linux kernel runs as a process.

T ---- TSC and CCNT are architecture-specific features.

F ---- One-shot timers can be used to implement regular periodic wake-ups of a thread.

F ---- A svc instruction traps to the kernel on x86 platforms.

T ---- A **`timespec`** can be obtained from a **`ktime_t`** value.

T ---- An oops in the idle task causes a kernel panic.

T ---- More lines of C code are for device drivers than for anything else in the Linux source overall.

F ---- A syscall always spawns a new thread to do its work.

F ---- A Linux **`rbtree`** provides logarithmic-time search.

F ---- A Linux linked list provides constant-time search.

7) (12 points) A user space program may invoke a syscall directly using the syscall interface. Please explain briefly what kind of information is provided to the kernel via the **`__NR_setuid`** argument in the following line of code.

```
int return_val = syscall(__NR_setuid, 0);
```

The __NR_setuid parameter is passing an integer constant that is the number associated to the system call from the system call table. The kernel uses that to determine which system call to execute. The 0 that is passed in is the parameter that is to be passed to the system call that corresponds to __NR_setuid that takes in one parameter. This sets the uid to 0. The system call number is passed into register %eax, while the parameter to pass to that system call is stored in %ebx, which will be read as the first parameter for the system call. (All of the registers noted are relative to x86)

Please explain briefly what the value that is found in the **`return_val`** variable indicates, immediately after that line of code has finished execution.

The *return_val* variable indicates whether or not the system call was a success or an error. If the *return_val* variable is a 0 (usually) , then it was a success. If it is a non-zero number, then it means there was an error.

What is the name of another variable that may contain additional related information at that same time, if the **`return_val`** variable contains a non-zero value?

Another variable that may contain additional information in the case of a non-zero value (error) is the *errno* variable. When the system call returns an error, a special error code gets written into the *errno* variable that can be translated into human-readable errors via functions that exist in the library.

Please explain briefly how a syscall may be invoked even if the program never uses the syscall interface directly.

In order for user space applications to invoke a system call, they need to signal the kernel using a software interrupt in order to have the system switch into kernel mode. There, the system call can be executed in kernel space on behalf of the application.

The software interrupt will incur an exception (syscall trap) and then the system will switch into the kernel mode. In this case, the exception handler becomes the system call handler. Using the exception (or trap) the user space is then able to enter the kernel. The number corresponding to the system call that is desired is placed into the %eax register by the user space. The system call handler then reads that value from the %ebx (based on x86) register and places the argument onto the kernel stack.

The kernel then takes the syscall number and jumps to the routine specified by that number in the interrupt table and executes the syscall.

Extra credit (1 point) Briefly, before we rebuilt our kernel after adding new syscalls, why was it important to issue the command `make clean` before recompiling the kernel?

Make only builds things that have been modified. Therefore, make clean needs to remove artifacts from the previous build so that the new syscalls can be registered and implemented and the old artifacts (that are seemingly unchanged) are updated.

8) (12 points) Briefly, what does `#pragma once` do?

#pragma ensures that the source file is only to be included once in a compilation. In other words it avoids multiple inclusions. It reduces compilation time for projects that have header files that are frequently used with #include.

Which top-level directory of the Linux kernel source code provides architecture independent .c source files?

The “kernel” directory holds the .c files (../linux_source/linux/kernel)

Which top-level directory of the Linux kernel source code provides architecture independent .h header files?

The “include” directory holds the .h files. (../linux_source/linux/include)

Which top-level directory of the Linux kernel source code provides architecture-dependent code in its sub-directories?

The “arch” directory holds the architecture dependent code. (../linux_source/linux/arch)

What processor architecture does the Raspberry Pi have?

The Raspberry Pi has a quad-core Arm Cortex-A53 CPU.

What is cross-compilation, and how have we used it?

Cross compilation is when code is compiled for the target computer system on a different system. We have used it when we compile code on the school lab computers that we then sftp over to our Raspberry Pi (the target). This is because the Raspberry Pi is too small to host the compiler and all the necessary files for it, whereas the Linux Lab computers at school are bigger.

9) In the blank next to each phrase on the left, please write the letter for the description that best matches it and that it best matches. (12 points)

D - procedure linkage table	~~A. pause execution at an instruction~~
C - global offset table	~~B. pause execution if memory changes~~
F - ld	~~C. is used to resolve library data~~
E - gcc	~~D. is used to resolve library functions~~
B - watchpoints	~~E. generates binary object files~~
A - breakpoints	~~F. performs dynamic linking~~
I - jiffies/HZ	~~G. file descriptor 1~~
L - jiffies	~~H. file descriptor 0~~
K - HZ	~~I. system uptime in seconds~~
H - stdin	~~J. data are kept in memory until flushed~~
G - stdout	~~K. tick rate~~
J - buffering	~~L. system uptime in ticks~~