Operating Systems – Monsoon 2022

Arani Bhattacharya, Sambuddho Chakravarty

November 7, 2022

Assignment 2 (Total points:150)

Due: Nov 22, 2022, Time:23:59 Hrs. (Hard deadline; delay penalties apply)

1 Linux `pthreads` and their scheduling (Total points: 80).

1.1 Thread scheduling

This exercise is to show you how to use Linux's scheduling policies for three threads. You need to launch three threads, each of which relies on three different functions, `countA()`, `countB()` and `countC()`. Each function does the same thing, *i.e.* counts from $1 - 2^{32}$. The following should be the detailed specification of each of the threads, to being with:

1. Thread 1 (call it **Thr-A()**): Uses SCHED_OTHER scheduling discipline with standard priority (nice:0).
2. Thread 2 (call it **Thr-B()**): Uses SCHED_RR scheduling discipline with default priority.
3. Thread 3 (call it **Thr-C()**): Uses SCHED_FIFO scheduling discipline with default priority.

Each of these threads must time the process of counting from $1 - 2^{32}$. You can use the `clock_gettime()` function for obtaining the actual time ticks that can be used to compute how long it took to complete a function.

We require you benchmark these three schedulers by changing the scheduling classes of the three threads (keeping the other scheduling priorities the same), against the counting program.

For these cases, you would require to rely on `pthread_schedsetparam()` and related functions for the same. You would require to generate histograms showing which scheduler completes the task when, depending upon the scheduling policy. You may choose different colors for the histogram bars, with one axis showing the time to complete, and the other showing the thread priorities. Of course, you cannot plot for all possible values of priorities; you would require to choose only some.

1.2 Process scheduling

This part of the exercise, involves creating three process, instead of the three threads. Each of these process should involve compiling a copy of the Linux kernel source (with the minimal config shared by the TAs earlier). The three processes should be created with `fork()` and thereafter the child processes should use `execl()` family system calls to run a different a different bash script, each of which should be having the commands to compile a copy of the kernel. To time the execution, the parent process could get the clock timestamp (using clock_gettime()) before the fork and after each process terminates (the event of which could be identified when the blocking system call `waitpid()`) returns.

What to submit/rubric.

1. Successful compilation of the program. **[Successfully compilation of all the three programs: 10 points. No points for programs that do not compile.]**
2. The program where three threads **Thr-A, Thr-B** and **Thr-C** are being created and they invoke their respective count functions that does the basic job of counting correctly. **[Programs run correctly as expected: 10 points. No points for programs do not do the tasks correctly.]**
3. Appropriately used system calls to create processes and threads, along with the system calls to set the scheduling discipline and their priorities.**[On use the appropriate system calls in the code: 20 points. Only 10 points to be awarded if the system calls to create process/thread, the calls to set the scheduling discipline and priorities is not used.]**
4. Program/thread runtimes and their variations due the different process/thread scheduling policies and priorities along with their histogram plot showing the same. **[On fully reproducible behaviour: 30 points. If behavior is not fully reproducible but the runtimes as controlled by the scheduler is predictable and exaplinable : 20 points. Otherwise no points.]**
5. Makefile to compile the above programs.**[Fully functioning Makefile: 5 points.]**
6. README/Write-up describing the program logic used for achieving the above (no more than one page)and an explanation of the outcomes of the tests/measurements.**[5 points]**

2 Kernel memory copy (`kernel_2d_memcpy()`) (Total points: 70).

This exercise aimed to test your understanding how system calls work. You need to write a system call, `kernel_2d_memcpy()`, which copies one 2-D floating point matrix to another. You would require using kernel functions like `__copy_from_user()` and `__copy_from_user()` to read data bytes from user space and write back to user space. In other words, this is a version of `memcpy()` that relies on the kernel to do the necessary copy operations, which are otherwise usually done directly in the user space (using the standard C library routines).

What to submit/rubric.

1. The `diff` between the stock kernel and the kernel with your appropriate system call. This `diff` could be patched into the stock kernel code so as to achieve the required functionality (*i.e.* the system call) and used eventually. **[Successfully compilation of the patched kernel: 50 points. No partial points.]**
2. A sample program to test the above system call. In the program you could hard-wire the source 2-D matrix (*i.e.* no need to take input from user at runtime or via a file). **[Program that correctly calls the system call with all the appropriate parameters and does the copy, provided (1) above works *i.e.*:10 points. Program compiles (as well as (1) above works), but copying is unsuccessful: 5 points.]**
3. README/Write-up describing the program logic used for achieving the above (no more than one page). **[5 points]**