CS 446/646 – Principles of Operating Systems

Homework 3

On Time Due date: Sunday, 3/24/2024, 11:59 pm
Late Due date: Sunday 3/31/2024, 11:59 pm

Objectives: You will test the effect of different system parameters on the Linux Scheduler's behavior, while running a Threaded program (which will be developed with the **`pthread`** library). You will be able to describe how Process Scheduling takes place, and how it can be controlled by various settings and options.

General Instructions & Hints:

- All work should be your own.
- The code for this assignment must be in C. Global variables are not allowed and you must use function declarations (prototypes). Name files exactly as described in the documentation below. All functions should match the assignment descriptions. If instructions about parameters or return values are not given, you can use whatever parameters or return value you would like.
- All work must compile on **Ubuntu 22.04**. Use a **`Makefile`** to control the compilation of your code. The **`Makefile`** should have at least a default target that builds your application.
- To turn in, create a folder named **`PA3_Lastname_Firstname`** and store your **`Makefile`**, your **`.c`** source code, and any text document files in it. Do not include any executables. Then compress the folder into a **`.zip`** or **`.tar.gz`** file with the same filename as the folder, and submit that on WebCampus.
- To write your **`sched.c`** code, you may ***only*** use the stdio, stdlib, string, pthread, sys/wait.h, sys/types.h, unistd.h, fcntl.h, errno.h, and sys/time.h libraries, and not all of them may be necessary. **If you choose to copy/paste the code in the print_progress.c file provided, you may include any additional libraries from that file.** You may **not** use the functions in those libraries within the code you've been asked to write in **`sched.c`**

Background:

Linux Scheduler Policy

In (very) short, there are 2 classes of Schedulable Entity Policies that are considered:

- Normal Task Policies
- Real-Time Task Policies

The default Normal Task Policy is **`SCHED_OTHER`**, which corresponds to a Round-Robin time-sharing policy with Priority adjustment by a Completely Fair Scheduling (CFS) scheme, which additionally takes into account a potential user-defined **`nice`**-ness value for each Process. The **`nice`** command allows a Process' niceness to be incremented / decremented. Additionally, there is a **`nice`** C-library function. Incrementing/decrementing a process' niceness effectively decreases/increases its Priority Weighting by the CFS. Note that there is no direct control over static priority queues, you can only indicate to the CFS that a Process should be favored more/less compared to others that have been awarded similar virtual execution times. Other Normal Task Policies (**`SCHED_BATCH`**, **`SCHED_IDLE`**, **`SCHED_DEADLINE`**) act as indications to the CFS on how to perform Priority Weighting as well. At the end of the day, it is a single CFS that is managing all Scheduling of Normal-class Tasks, and there are is **no notion of static Priority-Levels** that can affect its behavior.

Regarding Real-Time Task Policies, **SCHED_FIFO** and **SCHED_RR** implement **fixed Priority-Levels** Policies, and they are always able to Preempt any Normal-class Tasks managed by the CFS. I.e. Real-Time-class Tasks are always prioritized over Normal-class ones. Additionally, Real-Time Tasks have different **static Priority-Level Queues**, such that there exists a separate Priority_1 Queue, a separate Priority_2 Queue, …, Priority_99 Queue. Whenever a Task at Priority_n Queue gets execution time and then gets Preempted, it is returned back to the end of that same Priority_n Queue. Whenever a Task at Higher-Priority than the one currently executing arrives at Priority_m Queue (m>n), it immediately Preempts the Lower-Priority Task. If a Real-Time-class Task is Priority 99 on a single CPU machine, it will lock up (hang) the entire system. The difference between **SCHED_FIFO** and **SCHED_RR** applies therefore only to Real-Time-class Tasks at the same Priority-Level Queue. The main difference is that a **SCHED_RR** Task will operate on a Time-Slice policy that allows preemption when a quantum is complete, while a **SCHED_FIFO** Task is non-Preemptable and context switching with **SCHED_FIFO** is only done when it voluntarily yields.

The **chrt** command can be used to manipulate the Scheduling Policy (and potentially Priority) of a Task. To see the available Policies and their Min/Max priorities, open a terminal and input the following (NOTE: # is used by linux/bash to indicate a comment; you should not enter what follows a # into the terminal):

```
chrt -m #displays min/max valid priorities for each schedule policy

sudo chrt -p -r <prio> <pid> #switch task pid to Real-Time RR policy
     #with priority level prio

sudo chrt -p -f <prio> <pid> #switch task pid to Real-Time FIFO
     #with priority level prio

sudo chrt -p -o 0 <pid> #switch task pid to Normal OTHER Policy
```

Remember, you must replace anything in angled brackets <> with the requested information. For example, **<pid>** indicates replace the entire thing with the pid of an existing process, such as 12345. PID values for existing processes can be found using the **ps** command and various flags shown in class.

Linux Multi-Processor Control

The Scheduler is responsible for the allocation of Tasks between different CPUs of a Multi-Processor system. However, the CPUs that are available to it for manipulation, as well as the Affinity of Tasks for specific CPUs can be configured one of two existing sets of tools:

a) **isolcpus** & **taskset**
   This method allows CPUs to be reserved at kernel boot time such that the Scheduler has no control/awareness of them. You can use **isolcpus** as follows:
   1. On your Linux machine, edit (with **sudo**) the file **/etc/default/grub**. This is the file that controls the kernel selection entries you see listed when you boot up your system using grub (e.g. if you have a dual OS, Windows/Linux side-by-side setup). This file is not the kernel selection entries themselves, but it is used to generate them. Simply editing this file will not affect anything without completing step 3.
   2. Find the following line (may not match, look for **GRUB_CMDLINE_LINUX_DEFAULT)**:

      ```
      GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
      ```

Modify the line by appending the **isolcpus=0** (or **isolcpus=0-2** etc.) setting, e.g.

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=0"
```

Note: **=0** means isolate CPU0. **=0-2** means isolate CPUs 0,1,2.

3. Save the file, open a terminal, and run:

```
sudo update-grub
```

This will actually update the kernel boot entries you modified in the grub file in step 1, and you will see changes *after* rebooting. Changes made to the kernel are persistent, so if you want to revert them to allow all CPUs to be used, simply remove the **isolcpus=**... option that was appended in step 2.

4. After rebooting, you can explicitly control reservation and allocation of specific CPUs to certain tasks. To do so, you can open a terminal and use **taskset** to set the CPU Affinity of a specific Task, like so:

```
sudo taskset -p 0x00000001 <pid> #put task pid o CPU 0
```

You will not be required to use method a but it is encouraged to experiment with its workings and check out its side-effects.

b) **cpuset**

This method relies on the **cpuset** tool which provides more flexibility and does not have to be configured at boot time. It has certain limitations, such as the fact that for system stability certain (limited) kernel tasks have to remain active across all CPUs.

1. Install **cpuset** on your Linux system:

```
sudo apt-get install cpuset
```

2. **cpuset** use is covered here. Assuming you have a system with 8 CPUs (4 Physical Cores w/ HyperThreading), i.e. CPU 0 – CPU 7, you can do the following from any terminal with sudo permissions:

```
sudo cset set -l #show list of all active cpusets; assuming no
   #customizations, there will be one cpuset named root
   #containing all CPUs (0-7) and has all active tasks
   #associated with the set

sudo cset set -c 0-6 system #create cpuset named system
   #containing CPU 0-6, but not 7


sudo cset set -c 7 dedicated #create another cpuset named
   #dedicated that only has CPU 7

sudo cset set -l #same as first command, use to verify created
   #cpusets; only root cpuset should have tasks associated
```

```
sudo cset proc -m -f root -t system #move user-level tasks
   #from root to system cpu set created in second command

sudo cset proc -k -f root -t system #move all kernel-level
      #tasks from root cpuset to system cpuset created in
      #second cmd. A notification may warn you that certain
      #Kernel-Level Tasks shouldn't be moved from root. You can
      #force it, but try to remember that you proportionally
      #want fewer tasks scheduled to CPU 7 as compared to 0-6
```

3. We now have almost all tasks allocated to cpuset **`system`** (CPUs 0-6), and an (almost) free cpuset **`dedicated`** that is associated with CPU 7. We can now run a Processs, find its PID, and move it to the **`dedicated`** cpuset so that the process basically has exclusive access to CPU 7, like so:

   ```
   sudo cset proc -m -p <pid> -t dedicated
   ```

4. cpusets can be deleted to reclaim their tasks and assign the tasks back to the root cpuset. Clean up the cpusets like so:

   ```
   sudo cset set -d dedicated
   sudo cset set -d system
   ```

In this assignment, you must use Method b), which is much more flexible, in order to report your observations when changing such Scheduling-critical parameters for running Tasks. You will develop a Program that creates a user-controlled number of Threads that will each be doing busy-work, and you will examine the comparative behavior of manipulating just one of the Threads with respect to its Scheduling properties and Affinity.

You can implement further fine-grained control of the Scheduler using various **`settings`**. You are not required to tweak any of the settings mentioned in the linked documentation, but it is noteworthy how certain principles seem to violate the Policies discussed earlier (e.g. **`sched_rt_runtime_us`**).

General Directions:
Name your program **`sched.c`** . You will turn in C code for this. This program is *very similar* to what you wrote for assignment 2, so if you were confused then, please go to office hours and ask questions! **Locking is required and should not be indicated by the commandline arguments, unlike in assignment 2.** To build a program with **`pthread`** support, you need to provide the following flags to gcc: **`-pthread`**

You will need the following struct declaration in your code (it can be declared in **`sched.c`**):

```
typedef struct _thread_data_t {
    int localTid;
    const int *data;
    int numVals;
```

```
    pthread_mutex_t *lock;
    long long int *totalSum;
} thread_data_t;
```

You will need to write a minimum of the following functions (you may implement additional functions as you see fit):

➢ **main**

Input Params: **int argc, char* argv[]**
Output: **int**
Functionality: It will parse your command line arguments (**./sched 4**), and check that 2 arguments have been provided (executable, number of Threads to use). If 2 arguments aren't provided, **main** should tell the user that there aren't enough parameters, and then return -1. Otherwise, **main** should first dynamically allocate a fixed-size array of **2,000,000 int**s. You do not need to initialize the array with any values; the array is there to allow us to read the junk values in the array and calculate latency, which should not affect program behavior since we don't care about the sum of the array.

Create a **long long int** variable which will hold the total array sum, and initialize it to 0. Create and initialize a **pthread_mutex_t** variable that will be used by any created Threads to implement required locking, using pthread_mutex_init. Next, construct an array of **thread_data_t** objects large enough to hold the number of threads requested by the user.

Loop through the array of **thread_data_t**, and set:

a) the **localTid** field with a value that allows to zero-index the created Threads (i.e. if using 8 Threads, you should have 8 **thread_data_t** objects each with a **localTid** field of 0-7)
b) the **data** field pointer to the previously allocated array,
c) the **numVals** field which will correspond to the array size (2,000,000),
d) the **lock** field pointer to the previously created **pthread_mutex_t**, and
e) the **totalSum** field pointer to point to the previously created **totalSum** variable.

Create an array of **pthread_t** objects large enough to hold the number of requested threads, and run a loop of as many iterations. Within the loop, call **pthread_create** while passing it the corresponding **pthread_t** object in the array, the routine to invoke (**arraysum**), and the corresponding **thread_data_t** object in the array created and initialized in the previous step.

Subsequently, the program should perform **pthread_join** on all the **pthread_t** objects in order to ensure that the *main Thread* waits they are all finished with their work before proceeding to the next step.

➢ **arraySum**

Input Params: **void***
Output: **void***
Functionality: This function is executed by each **thread_data_t*** object in the array, which is what is "passed" as a parameter. Since the input type is **void*** to adhere by the pthread API, you have to typecast the input pointer into the appropriate pointer type to reinterpret the data.

In this assignment, **arraySum** will be doing constant *Busy-Work*. Just like in assignment 2, the program will still be reading values from the **thread_data_t->data** array, calculating a locally defined **long long int** threadSum with the array sum, and then updating the **thread_data_t->totalSum** (remember to Lock and Unlock the **thread_data_t->lock** mutex while modifying shared data), but now the function:

a) Repeatedly calculates the sum of the entire array (**thread_data_t->numVals**) without terminating ; i.e. the **for** loop doing the **threadSum** calculation across the values array and the update of the **thread_data_t->totalSum** should be inside a **while(1)** loop.
c) Calculates timing the latency for each iteration of the sum **for** loop, and extracts the maximum latency that took place for each execution of the **while** loop, i.e.:

i) within the inner **for** loop, create a **struct timespec** object and use **clock_gettime** to store its value. At the end of the for loop create another **struct timespec** object and use **clock_gettime** to update it. Use these **timespec** objects to calculate a **long int** variable which will be the time difference between them (in **nanoseconds**).
ii) At the beginning of the outer **while** loop, create a **double** variable that represents your maximum observed latency. At each iteration of the inner **for** loop, update the double variable to reflect the maximum observed latency. In other words, if the most recent iteration took longer than your maximum observed latency, the maximum observed latency should be updated with the new larger latency from that iteration.
iii) at the end of each iteration of the outer **while** loop, report the maximum observed latency. You are provided with a sample function **print_progress(pid_t local_tid, size_t value)** that is capable of doing that nicely on your terminal. You just have to provide it with the **thread_data_t->localTid** and the maximum latency you just calculated. It will print out the Process PID of the Thread that called it, as well as progress-bar visualization of the maximum latency that was observed over the last run of the outer while loop.

As mentioned, this program will have a number of user-defined Threads just doing redundant calculations infinitely (Busy-Work). The purpose of this assignment is to observe the maximum latency and its variation *indirectly*, through a timing that happens within the fast inner **for** loop and its maximum value over each **while** loop iteration.

Finally answer the following questions in a pdf document:

Note: For these Questions the system's behavior will not be the same when running under a Virtual Machine environment vs running on the actual Operating System. Therefore, you may develop the **sched.c** application and test that your commands sequences work in a VM or in WSL, but in order to report actual observations of the effect of these commands you should execute them at least once on machine with a real Linux OS (ECC/WPEB labs/dual booted laptop/ friend's machine).

1) Run the program with the same number of threads as there are CPUs on the machine. if you don't know how many CPUs the machine has, you can use the **nproc --all** command in your terminal, or the **sysconf(_SC_NPROCESSORS_ONLN);** syscall in C. What do you observe?

2) Run the program again with the same number of Threads. Open a different terminal and issue the command:

```
watch -n .5 grep ctxt /proc/<pid>/status
```

This will display the number of Context Switches (voluntary and involuntary) that a Task with PID **<pid>** has undergone so far, and get an update of it every 0.5 seconds. You can find the PID of a specific task through **pstree -p**, or more easily by the running **sched.c** which should now be printing out the PID of each thread alongside each progress bar. Now (as you are observing the Context Switches of a specific Thread), switch the process' Scheduling Policy to a Real-Time policy. Try both Policies, and 1-2 different Priority Levels. What sequence of commands did you use to answer this question? What did you observe?

3) Run the program again with the same number of Threads. Create cpuset named **system** with all CPUs except 1. This is described in the background section containing the *Linux Multi-Processor Control* subsection that contains Method b. Move all Tasks (all User-Level and Kernel-Level ones that are possible to move) into that set. Create a cpuset named **dedicated** with only the CPU that is excluded from the **system** cpuset. Move one of the Threads of **sched.c** to the **dedicated** cpuset. What sequence of commands did you use to answer this question? What did you observe?

4) While 3) is still executing with one Thread on the **dedicated** cpuset, observe the Context Switches of that Thread with:

```
watch -n.5 grep ctxt /proc/<pid>/status
```

and then change its Scheduling Policy to a Real-Time one at 1-2 Priority Levels.
What sequence of commands did you use to answer this question? What did you observe?

Submission Directions:
When you are done, create a directory named **PA3_Lastname_Firstname**, place your **Makefile** (which has to have at least one default target that builds your **sched** application executable), your **sched.c** source code file (you can embed the provided **print_progress()** function and the struct within it), and the text document providing your answers to the above questions into it, and then compress it into a **.zip** or **.tar.gz** with the same filename as the folder.
Upload the archive (compressed) file on Webcampus.

Early/Late Submission: You can submit as many times as you would like between now and the due date. A project submission is "late" if any of the submitted files are time-stamped after the on time due date and time. There is no penalty for late submission; however, a 12.5 point bonus will be added to students who complete all 5 assignments by the on time due date.

Verify your Work:
You will be using **gcc** to compile your code. Use the **-Wall** switch to ensure that all warnings are displayed. Strive to minimize / completely remove any compiler warnings; even if you don't warnings will be a valuable indicator to start looking for sources of observed (or hidden/possible) runtime errors, which might also occur during grading.

After you upload your archive file, re-download it from WebCampus. Extract it, build it (e.g. run **make**) and verify that it compiles and runs on the ECC / WPEB systems.

➢ Code that does not compile and run will receive an automatic 0.