CS 446/646 – Principles of Operating Systems

Homework 3

Due date: Tuesday, 10/17/2023, 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 18.04**. 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.

Background:

Linux Scheduler Policy
The Linux Scheduler Policy is described in https://linux.die.net/man/2/sched_setscheduler.
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`**](https://linux.die.net/man/1/nice) command allows to increment / decrement a Process' niceness, effectively decreasing/increasing its Priority Weighting by the CFS (there is no direct control over static priority queues, you can only indicate to the CFS that a Process should be favored more/less when compared to others that have been awarded similar execution times).
Note: There also exists a nice [**`nice`**](https://linux.die.net/man/3/nice) C-library function.

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.

Also, 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.

Note: The existence of a never-ending Real-Time-class Task at Priority 99 on a single-CPU machine 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, and it's the fact a **SCHED_RR** Task will operates on a Time-Slice policy (a Priority_n Queue Task will get Preempted when it consumes its Quantum such that another Priority_n Queue Task can receive execution time, etc), while a **SCHED_FIFO** Task is non-Preemptable (by Tasks of the same Priority-Level Queue) and switching away from is only done when it voluntarily yields.

Note: Or of course when a Higer-Priority Task arrives at a Higher Priority-Level Queue.

Usage:
The [**chrt**](https://linux.die.net/man/1/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 do:
chrt -m
To have a Task with PID **<pid>** switch to a Real-Time RR Policy of Priority Level **<pri>**:
sudo chrt -p -r <prio> <pid>
To have a Task with PID **<pid>** switch to a Real-Time FIFO Policy of Priority Level **<pri>**:
sudo chrt -p -f <prio> <pid>
To have a Task with PID **<pid>** switch to the Normal OTHER (default) Policy:
sudo chrt -p -o 0 <pid>

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 through a set of tools:

a) **isolcpus** & **taskset**
This method allows to reserve at kernel boot time certain CPUs ("isolate" them) 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 (if e.g. you have a dual OS, Windows/Linux side-by-side setup)
Note: This file is not the entries themselves, but it is used to generate the above entries (i.e. just editing will not affect anything, see step 3. on how to do that).
2. There should be a line
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
or similar, modify it 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 and run
sudo update-grub
This will actually update the kernel boot entries, and will be used *the next time you reboot.*
Note: This is now a *persistent* change. If you wish to revert back to having all the system CPUs available for the Scheduler to use, you have to apply steps 1-3 and remove the **isolcpus=**... option that was appended in step 2.

4. After rebooting, you can use [**taskset**](https://linux.die.net/man/1/taskset) to set the CPU Affinity of a specific Task, i.e.:
   sudo taskset -p 0x00000001 <pid>
   will have the Task with PID **<pid>** run on CPU 0.

This way you can achieve explicit control over the reservation and allocation of specific CPUs to certain tasks, which for instance depending on your application can be considered critical to exhibit minimal latency.

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. Its use is covered here: https://documentation.suse.com/sle-rt/15-SP4/html/SLE-RT-all/cha-shielding-cpuset.html. For a minimal example you can do the following:
   Assuming you have a system with 8 CPUs (4 Physical Cores w/ HyperThreading), i.e. CPU 0 – CPU 7:
   Show a list of all active cpusets:
   sudo cset set -l
   (You will see that at this point you only have one cpuset (named **root**) that contains all your system CPUs (0-7) and has all active Tasks associated with it.)
   Now, create a new cpuset named **system** that includes CPUs 0-6 (**all except CPU 7**)
   sudo cset set -c 0-6 system
   Create also another cpuset named **dedicated** that include only CPU 7
   sudo cset set -c 7 dedicated
   (Verify you can see all the cpusets and the CPUs associated by running **cset set -l** again. You will notice your **system** and **dedicated** cpusets have no Tasks associated with them, only the **root** one has Tasks.)
   Now, move all User-Level Tasks from the **root** set to the **system** set
   sudo cset proc -m -f root -t system
   Now, move all Kernel-Level Tasks from the **root** set to the **system** set
   sudo cset proc -k -f root -t system
   CPUset will notify you that for safety reasons, certain Kernel-Level Tasks should not be moved. You can force this, but it is that crucial to leave just a few Taks capable of being scheduled on CPU 7.
3. At the moment what we have achieved is to have (almost) all Tasks on cpuset **system** that is associated with CPUs 0-6, and we have an (almost) free cpuset **dedicated** that is associated with CPU 7.
   Now we can run a Process, find its PID, and move it to the **dedicated** set, effectively having it owning (almost) exclusively CPU 7:
   sudo cset proc -m -p <pid> -t dedicated
   (where **<pid>** the PID of a Process we wish to move to the set)
4. Note: You can finally delete cpusets with
   sudo cset set -d dedicated
   sudo cset set -d system
   Deleting them will just reclaim their Tasks and assign them back to the **root** set

You will not be required to use Method a), but it is encouraged to experiment with its workings and checking out its side-effects. You will be required to 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 used-controlled amount 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.

Note: For further fine-grained control of the Scheduler you can look at this resource: https://documentation.suse.com/sles/15-SP1/html/SLES-all/cha-tuning-taskscheduler.html. You are not required to tweak any of the settings mentioned here, but it is noteworthy how certain principles seem to violate the above mentioned Policies (e.g. **`sched_rt_runtime_us`**).

General Directions:

Name your program **`sched.c`** . You will turn in C code for this. To build a program with **`pthread`** support, you need to provide the following flags to gcc: **`-pthread`**

You may only use the following libraries, and not all of them are necessary:

```
<stdio.h>
<string.h>
<stdlib.h>
<sys/wait.h>
<sys/types.h>
<unistd.h>
<fcntl.h>
<errno.h>
<sys/stat.h>
<pthread.h>   // for pthreads
<time.h>  // for clock_gettime
```

You will also need the following struct declaration in your code:

```
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. Just reading from these values (although uninitialized) is fine and won't affect the program's behavior. Create and initialize to 0 a **`long long int totalSum`** variable which will hold the total array sum. Create and initialize a **`pthread_mutex_t`** variable that will be used by any created Threads to implement required locking, using [**`pthread_mutex_init`**] (https://linux.die.net/man/3/pthread_mutex_init). At this point, the program should construct an array of **`thread_data_t`** objects, as large as the number of Threads requested by the user. Then it should 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.

The program should now make an array of **pthread_t** objects (as large as the number of Threads requested by the user), and run a loop of as many iterations, and call within it **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 (the one that contains per-Thread arguments such as the start and end index that the specific Thread is responsible for).

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***
  It is assumed to operate on **thread_data_t*** input data (but 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*. This means it will still be reading values from the **thread_data_t->data** array, calculating a locally defined **long long int threadSum** with their sum, and then updating a the **thread_data_t->totalSum** (remember to appropriately Lock and Unlock the **thread_data_t->lock** mutex while modifying shared data), but now the function:
  a) Will be working on the entire array length (**thread_data_t->numVals**)
  b) Will be repeating the same calculation over and over without terminating, i.e. the **for** loop doing the **threadSum** calculation and the update of the **thread_data_t->totalSum** should be inside a **while(1)** loop.
  c) Will be timing the latency for each iteration of the sum **for** loop, and extracting the maximum latency that took place for each execution of the **while** loop, i.e.:
    i) within the inner **for** loop, in the beginning you should create a **struct timespec start** value and use [**clock_gettime**](https://linux.die.net/man/3/clock_gettime) to store its value, and at the end of the for loop you should create another **struct timespec end** value and using **clock_gettime** again update it, and use these to calculate a **long latency** which will be the time difference between them (in **nanoseconds**).
    ii) in the outer **while** loop, at the beginning of it you should create a **latency_max** variable, which within each iteration of the inner **for** loop should be updated to reflect the maximum observed **latency**.
    iii) at the end of each iteration of the outer **while** loop, you should use the **latency_max** and report it on the terminal.
    You are provided with a sample function **print_progress(int local_tid, int 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 **latency_max** you just calculated. It will print out the Process PID of the Thread that called it, as well as progress-bar visualization of the **value** (the **latency_max** 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 the same calculation over and over (Busy-Work). What we are interested in 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 text document of your choice (.pdf, .doc, .docx, .odt, .txt., etc.):

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, 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 (using a colleague's machine to just run these is fine with respect to Plagiarism policies).

1) Run the program with an amount of Threads as many as the number of CPUs of your system (if you don't know you can use the **`nproc --all`** command in your terminal, or the **`sysconf(_SC_NPROCESSORS_ONLN);`** syscall in C.
Write your observations

2) Run the program again with the same amount of Threads. Open a different terminal and issue the command:
`watch -n.5 grep ctxt /proc/<pid>/status`
This allows you to see 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 its Scheduling Policy to a Real-Time one. Try both Policies, and 1-2 different Priority Levels.
Write the sequence of commands you used for this question, and your observations.

3) Run the program again with the same amount of Threads.
Create a **`system`** cpuset with all CPUs except 1 (as described in Method b) in the section about *Linux Multi-Processor Control*) and move all Tasks (all User-Level and Kernel-Level ones that are possible to move) into that set. Create a **`dedicated`** cpuset with only the CPU that is excluded from the **`system`** one.
Move one of the Threads of **`sched.c`** to the **`dedicated`** cpuset.
Write the sequence of commands you used for this question, and your observations.

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.
Write the sequence of commands you used for this question, and your observations.

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 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 due date and time. Projects that are up-to 24 hours late will receive a 15% penalty, ones that are up-to 48 hours late will receive a 35% penalty, ones that are up-to 72 hours late will receive a 60% penalty, and anything turned in 72 hrs after the due date will receive a 0.

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 will receive an automatic 0.