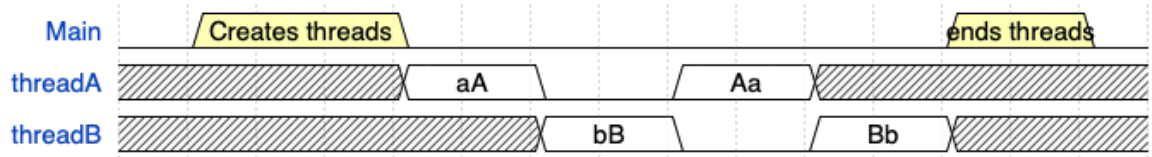
Paul Beglin

Coursework 1: Real-time Programming with pthreads

1. Understanding ‘critical.c’

```
pi@raspberrypi:~/Downloads $ sudo ./critical
Start time is: 1614946926 seconds 927449953 nano_seconds
main() waiting for threads
aaaaaAAAAAbbbbbBBBBBAAAAAaaaaaBBBBBbbbbb
main() reporting that all threads have terminated
```

This is a FIFO stack (First in First out). The code is being written sequentially 5 ‘a’ then 5 ‘A’ (I will write “aA” from now on to be more concise) “Aa”→ “bB” → “Bb”. But as we can see from the terminal screenshot above, it doesn’t output like this; because between the two “aA” we change the priority level of threadB; therefore despite being lower in the queue, it jumps up and gets returned before the second “Aa”; we then increment the priority of threadA, like this both threads have the same priority level; therefore the queue goes back to “normal” and outputs “Aa” → “Bb”.

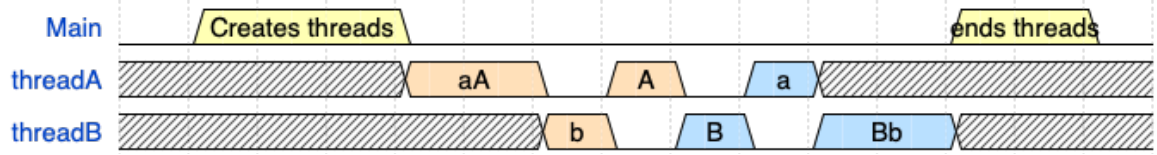



(The ‘Main’ thread does a lot of things for the initialization, among them: defining time, setting processor, setting priority, creating the threads, joining the threads. Then it reports the threads are terminated and deletes the mutex)

2. ‘critical.c’ with Mutexes

```
pi@raspberrypi:~/Downloads $ sudo ./critical1
Start time is: 1615077108 seconds 33435128 nano_seconds
main() waiting for threads
aaaaaAAAAAbbbbbAAAAABBBBBaaaaaBBBBBbbbbb
main() reporting that all threads have terminated
```

After “commenting-out” the mutexes, we have a first step after ‘a’ where we acquire the mutex lock (we enter a “critical region”); as the program goes on, it prioritizes threadB (like last time) but before it reaches ‘B’ it tries to lock the mutex again (except it’s already locked) so it has to wait until ‘A’ is written (which then unlocks the mutex), enabling threadB to lock the mutex, enter “critical region” and finish the normal queue (like last time, giving priority to threadA to equalize importance).

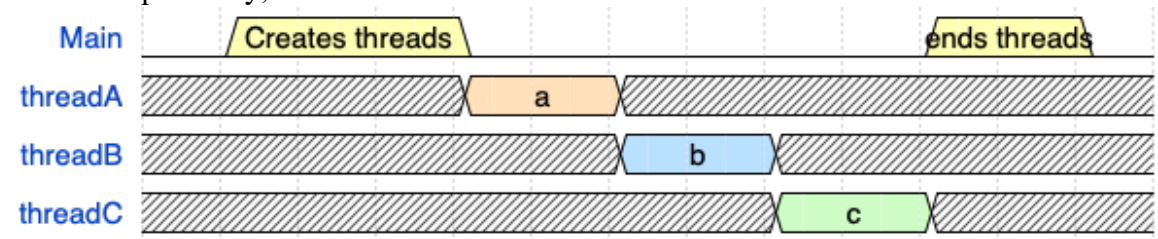



(orange is the first time the mutex is locked; blue is the second time the mutex is locked; this is simplified because the status is not locked on the first ‘a’ and final ‘b’)

3. Understanding ‘multithread.c’

```
pi@raspberrypi:~/Downloads $ sudo ./multithread
Start time is: 1615133963 seconds 357798336 nano_seconds
main() waiting for threads
aaaaaaaaaabbbbbbbbbbccccccccccc
main() reporting that all threads have terminated
```

In this program, the main() setup is similar to ‘critical.c’, afterwards the threads are just executed sequentially, first 10 ‘a’ then 10 ’b’ then 10 ‘c’.




4. Modified ‘multithread.c’

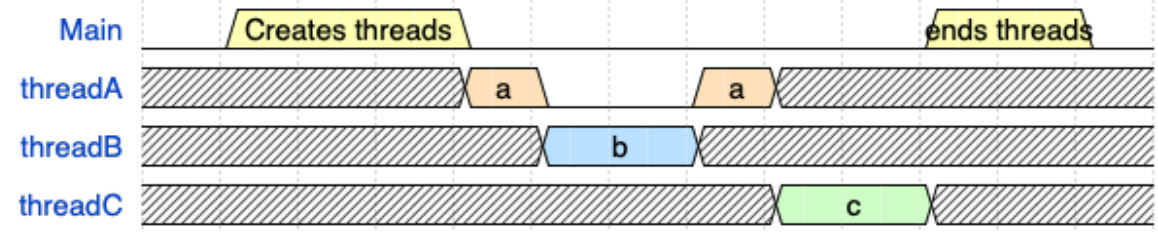
```
pi@raspberrypi:~/Downloads $ sudo ./multithread1
Start time is: 1615193920 seconds 782548513 nano_seconds
main() waiting for threads
aaaaabbbbbbbbbbaaaaaccccccccccc
main() reporting that all threads have terminated
```

I changed the code in threadA like so:

```
void *threadA(void *arg)
{
  int j;

  for(j=1;j<=10;j++){
  printf("a");
  if (j==5){
    param.sched_priority = priority_min+2;
    pthread_setschedparam(threadB_id,policy,&param);
  }
  }
```

Here we are increasing the priority of threadB after printing the first 5 ‘a’ (at the end of the fifth loop); therefore, threadA has to wait as threadB gets prioritized and executes its printing of 10 ‘b’ until it terminates and resumes threadA (and then threadC).

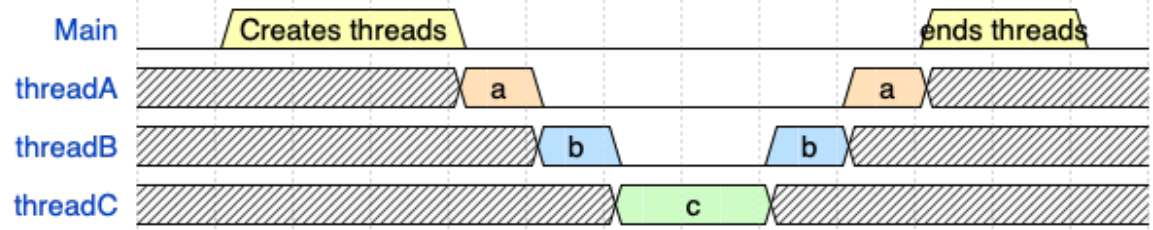



If we decrease the priority of threadB, we can see how threadC “jumps ahead” and gets prioritized over threadB (once threadA has finished) as shown here:

```
pi@raspberrypi:~/Downloads $ sudo ./multithread2
Start time is: 1615194934 seconds 130720537 nano_seconds
main() waiting for threads
aaaaaaaaaaccccccccccbbbbbbbbbb
main() reporting that all threads have terminated
```

Finally, I understand the last part of this question as "Modify the priority of the executing thread after it printed half (5) letters" therefore, I will modify the code to implement when the priority is increased (on every thread after 5 prints) and once when it is decreased.
When we modify to decrease:

```
pi@raspberrypi:~/Downloads $ sudo ./multithreadmin
Start time is: 1615195574 seconds 815483188 nano_seconds
main() waiting for threads
aaaaabbbbbccccccccccbbbbbaaaaa
main() reporting that all threads have terminated
```

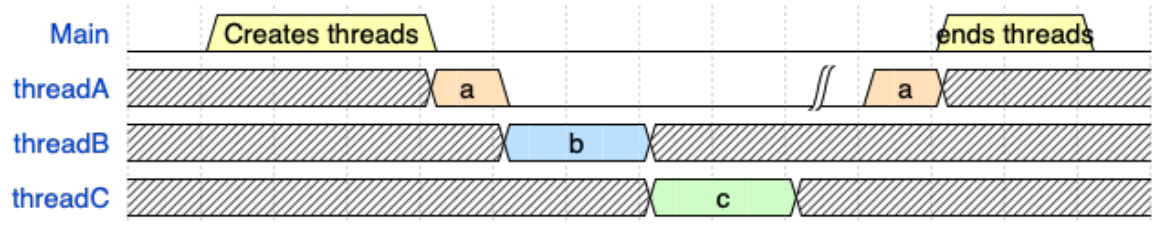



As we can see in this diagram, first threadA executes; after 5 'a' its priority decreases, therefore threadB gets prioritized and executes; but after 5 'b' its priority also decreases, which leaves threadC to be prioritized, and even though its priority decreases, it's equal to the 2 other threads, and since threadC is currently executing, it's at the front of the queue and therefore keeps its place. After threadC terminates, we have the normal queue order resuming with first threadB (since it executed after threadA, it's at the front of the queue) then threadA.

On the other hand, increasing the priority of the currently executing thread is quite uninteresting; since the executing thread is already at the front of the queue (being executed), increasing its priority won't change anything; just giving the sequential output 'a'→'b'→'c'.

5. Modified 'multithread.c'

```
pi@raspberrypi:~/Downloads $ sudo ./multithreadnano
Start time is: 1615196766 seconds 912562238 nano_seconds
main() waiting for threads
aaaaabbbbbbbbbbccccccccccaaaaa
main() reporting that all threads have terminated
```



As we can see in the timing diagram, since threadA has to wait, it leaves the time for threadB and threadC to execute sequentially. Then after the waiting time is over, threadA terminates.

```
void *threadA(void *arg)
{
  int j;

  for(j=1;j<=10;j++){
  printf("a");
  if (j==5){
    struct timespec ts = {0, 1000000L };
    nanosleep(&ts, NULL);
```

6. Scheduling policy 'multithread.c'

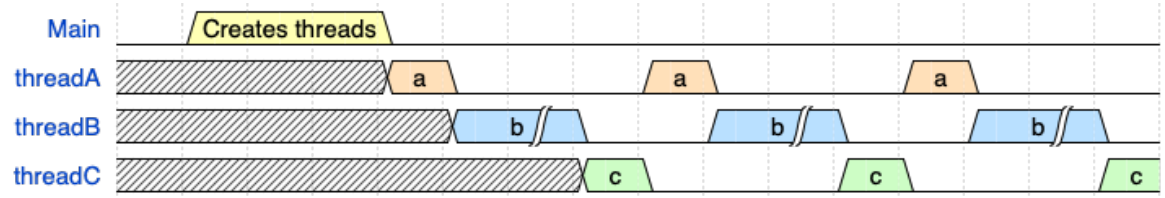
What is the difference between Round-Robin and FIFO (or FCFS):
Round-Robin is technically an enhancement to FIFO, where tasks of similar priority values get a maximum time (quantum) to run on the processor, after that it's on to the next task/thread, all repeating in a cycle until each task is finished.

In order to show the difference in practice between FIFO and RR, we need to make a task that takes a longer time than the maximum time quantum allowed in RR; like this we will be able to see how it automatically switches to the next task (as the currently executing thread is stopped and put at the bottom of the queue in RR).

```
void *threadC(void *arg)
{
  int j;
  j=0;
  while(1){
    j++;
  if (j==60000){
    j=0;
    printf("c");
```

For this code in RR, the output every cycle is going to be (a bunch of) 'a' → 'b' → 'c' → 'a' → 'b' → 'c'…
Whereas in FIFO it's going to get stay on the infinite while loop of 'a'.

Another test we can make, is to show that when we have a thread with a higher priority than any other, even in RR, this thread will get prioritized and will have no time constraint; it will either finish its task or change its priority to a lower level where RR scheduling takes place.




Here we demonstrate this by having threadB prioritized for a longer period than the maximum time (quantum), before reducing its priority to an equal level. (or just letting it stay at a high priority level forever, showing the maximum time constraint doesn't intervene). In FIFO it's going to do 'a' → 'b' → 'c' in one go.

```
while(1){
  j++;
if (j==100000){
  j=0;
  printf("b");
  param.sched_priority = priority_min+2;
  pthread_setschedparam(threadB_id,policy,&param);
  i++;
}
if (i==300){
  param.sched_priority = priority_min;
  pthread_setschedparam(threadB_id,policy,&param);
  i=0;
```