Real-Time Programming with Pthread

In shared-memory multiprocessor architectures, threads are a useful tool to implement parallelism within an application or real-time system[1]. Threads also allow applications to concurrently perform multiple tasks[2]. Characteristics that offer practical advantages, and improved performance relative to non-threaded applications. Advantages, such as the handling of asynchronous events, and the ability to implement priority scheduling[1]. A thread is defined as a sub-procedure that can be independently scheduled to run by the operating system; entities capable of running independently of the main program because they contain the bare essential resources required to exist as executable code[1][3]. Relative to the management and creation of processes, threads are lightweight code structures that require far less computational overhead[1]. Task independence is rare in concurrent programming, consequently, the behavioral success of a concurrent program is critically dependent on the synchronization, and communication of tasks [4]. Synchronization ensures the integrity of shared data, enabling the safe interactions of different resources by protecting critical sections of a program through mutual exclusion[3][4]. Mutual exclusion prevents multiple tasks from simultaneously accessing a resource, protecting the critical sections that deal with a shared resource from being interrupted by any other task trying to access it[4]. Within pthread programming, only one thread can lock (or own) a mutex variable at any given time. If several threads try to lock a mutex at the same time, only one thread will be successful, and no other thread can access said mutex until the owning thread unlocks it. Thus, all threads can only access the protected data "in turns"[1] This report explores the effects of mutual exclusion using mutex variables, in addition to investigating multithreading and scheduling, using Portable Operating System Interface (POSIX) threads (pthreads).

Task 1

The main objective of critical.c is to demonstrate the effects of mutual exclusion using a mutex variable. Initially, the program comprises of a single default thread, the main() thread. Two additional executable threads are created using the thread_create command. Once created the routines can be called any number of times from any section of the code[1].

In this report, the x-axes for all the timing diagrams have been offset from the origin, starting at the creation of the subthreads rather than the beginning of the main() thread. Prior to the creation of the aforementioned subroutines, a scheduling policy must be set. A scheduler is the programming component that decides which executable thread the CPU will run next[2]. A First-In, First-Out (FIFO) scheduling policy is set; setting the appropriate policy is key in any real-time system as it directly influences the responsiveness of the system to external stimuli. If a FIFO process is preempted by another higher priority process, it will stay at the head of the list for its own priority level until the higher priority process is blocked, thereafter resuming execution. When a FIFO process transitions to the runnable condition, it is inserted at the end of its equal priority list. Once "running" a FIFO process will continue execution until it is blocked by either an I/O request or is preempted by a process of higher priority[2]. To fully demonstrate the interleaving of threads, the processor affinity is set such that all the threads execute on a single CPU core. A thread's CPU affinity mask determines which set of CPUs it is permitted to run on [2]. On a multiprocessor system, setting a CPU affinity mask regime can enhance computational performance. Dedicating a single CPU to an individual thread ensures the maximum achievable execution speeds for that particular thread. Additionally, restricting a thread to a single CPU reduces performance cost due to cache invalidation, which is when a thread stops execution on one CPU and begins on another[2]. Initially, both threadA and threadB are assigned the same priority levels. Once the main() thread creates and calls threadA and threadB, threadA begins by executing its non-critical output, printing "a" five times. When it begins to execute, the thread moves from the "ready" condition in the task queue to the "running" condition,therefore it is currently the only task executing on the CPU. The routine then moves on to the critical section of its code, printing "A" five times. During this time threadB remains second in the task queue having already been created and called by the main routine. Next, threadA increments the priority of threadB. Now, threadB has a higher priority than threadA. One of the conditions of the FIFO scheme is that a process will run until it is preempted by a higher priority process, thus threadB having the higher priority preempts threadA and begins execution. Sub-

sequently, threadA moves from the "running" condition to the "waiting" condition, behind threadB in the task queue. The basic framework of threadB mirrors threadA. First, threadB executes its non-critical section of the code, printing "b" five times, moving on to its critical section, printing "B" five times. It then increases the priority of threadA, and is then preempted by threadA. Once, threadA has finished executing the remainder of its routine, the remaining routine of threadB executes to completion.

```
Start time is: 1614444926 s 639415829 ns
main() waiting for threads
aaaaaAAAAAbbbbbBBBBBAAAAAaaaaaBBBBBbbbbb
main() reporting all threads have terminated
```

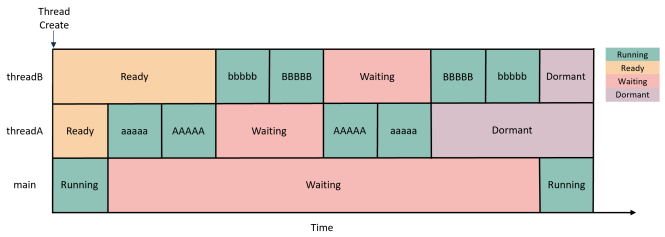

Figure 1: Task 1: No Mutual Exclusion

The current structure of the program does not protect the critical sections of each thread. Each thread is capable of executing its own critical section as well as the critical section of the other thread simply by increasing the other thread's priority. It should also be noted that once threadA and threadB begin to execute, in their respective orders, the pthread_join() function is called which forces the main() thread to "wait", until both threadA and threadB are finish executing before it can finish it's own execution.

Task 2

The program is modified to demonstrate the use of a mutex variable. First, threadA begins by executing its non-critical code block, printing "a" five times. Next, the mutex object is locked by calling the pthread_mutex_lock() function. Once the program enters threadA's critical section, it prints "A" five times. Next, threadA increments the priority of threadB, which causes threadB to execute under the preemption condition for the FIFO policy. Now, threadB executes its first non-critical section, printing "b" five times. When complete, the subroutine moves to the next section, which is a mutex lock. If a mutex is already locked the calling thread is blocked until the mutex is unlocked[5].

Therefore, the program moves back to executing the remaining critical section of threadA, printing "A" five times. The mutex is then unlocked using the pthread_mutex_unlock() function, which releases the mutex object. As threadB still has a higher priority than threadA, the remainder of threadB begins to execute. First, the mutex lock is called by threadB, thus locking its critical section. Subsequently threadB begins to execute the first part of its critical section, printing "B" five times. Following this, threadB increments the priority of threadA, causing threadA to preempt threadB, and execute the remainder of its non-critical section by printing "a" five times. Once finished, threadB being the next thread in the queue executes the remainder of its critical section, printing "B" five times. The pthread_mutex_unlock() function is once again called, which releases the mutex object, and threadB prints what is left of its non-critical section (i.e., it prints " b" five more times).It should be noted that the pthread_mutex_destroy() function is used at the end of the main() thread to destroy the mutex object, causing it to uninitialise at the end of the main () thread routine[1].

```
Start time is: 1614446342 s 518025023 ns
main() waiting for threads
aaaaaAAAAAbbbbbAAAAABBBBBaaaaaBBBBBbbbbb
main() reporting all threads have terminated
```

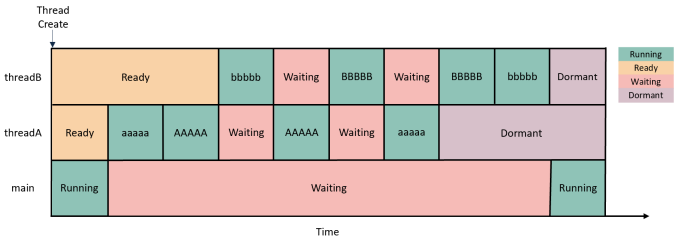

Figure 2: Task 2: Effects of Mutual Exclusion

Task 3

The program multithread.c has, a main() parent thread and three subthreads, threadA, threadB, and threadC. All the threads have equal priority, executing in chronological order, from threadA to threadC; once called each subthread simply prints out its respective letter ten times.

```
Start time is: 1614447968 s 618138695 ns
main() waiting for threads
aaaaaaaaaabbbbbbbbbbccccccccccc
main() reporting all threads have terminated
```

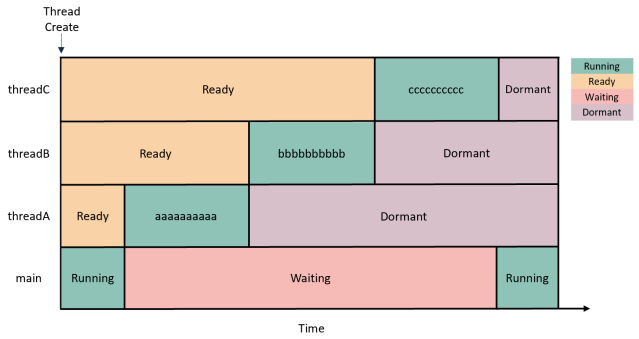

Figure 3: Task 3 - Multithreading

Task 4

The original program is modified such that threadA prints out half of its letters, then increases the priority of threadB. Since, threadB now has a priority greater than threadA, threadB preempts threadA halfway through execution, and threadA moves to the head of the queue for its priority level (i.e., in front of threadC).

```
Start time is: 1614968171 s 647919226 ns
main() waiting for threads
aaaaabbbbbbbbbbaaaaacccccccccc
main() reporting all threads have terminated
```

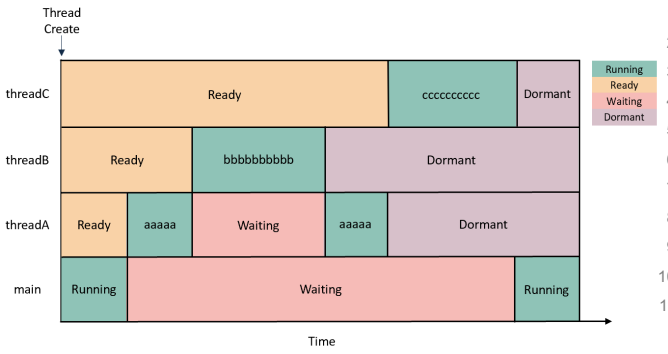

Figure 4: Task 4: Priority Switch

Finally, the original code was altered such that the "current running" thread increases its own priority before execution of its "For" loop. The output was the same as the unmodified program. The only effect increasing the priority of the "current running" thread is that it will have a higher priority than all the other dormant threads, and therefore, execute before them. The program was once again modified so that the "current running" thread now decrements its own priority before the execution of its "For" loop, the output was the same as the unmodified program. Initially, threadA is called, it then decreases its priority, thus moving to the back of the queue. Next, threadB is called, it also decreases its own priority, and is now the same priority level as threadA, however, threadB moves to the queue position behind threadA. Finally, threadC is called, the aforementioned process repeats, and threadC moves to the back of the queue. Now, threadA is once again at the front of the queue, and because it has already executed the code section that decrements its own priority, it moves onto its next section of code which is its "For" loop. Thus, the ordering is unchanged in this instance as each thread decrements its own priority by an equal amount. The applied modification is presented below:

```
Void *threadA(void *arg){
  int j;
  int status;
  param.sched_priority= priority_min-2;
  status = pthread_setschedparam(threadA_id,
    policy,&param);
  for(j=1;j<=10;j++){
        printf("a");}
  return (NULL);
```

Task 5

The original program was once again modified such that threadA sleeps for 1 ms after printing half of its letters using the nanosleep () function.

```
void *threadA(void *arg){
  int j;
  for(j=1;j<=5;j++){
        printf("a");}
  struct timespec ts;
  ts.tv_sec=0;
  ts.tv_nsec= 1000000L;
  nanosleep(&ts,NULL);
  for (j=1;j<=5;j++){
        printf("a");}
  return (NULL);}
```

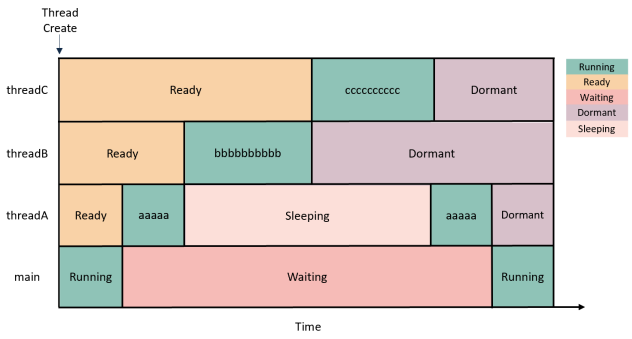

Figure 5: Task 5: Nanosleep

```
Start time is: 1614975431 s 675005847 ns
main() waiting for threads
aaaaabbbbbbbbbbccccccccccaaaaaa
main() reporting all threads have terminated
```

The nanosleep() function works by suspending the execution of the calling thread until the specified time has elapsed. Once asleep, threadA is superseded by threadB and threadC. After 10 ms, threadA wakes up, and executes the remainder of its code.

Task 6

Finally, the effects of a Round-Robin (RR) scheduling scheme were assessed. The goal was to increase the execution time of each thread by using a "For" loop counter, causing it to exceed its allotted CPU time slice. First, the original program was modified from a SHED_FIFO scheduling policy to a SHED_RR policy. Under the RR policy, threads of equal priority execute in a round-robin time-sharing fashion, whereby each thread is allowed a fixed-length time slice of the CPU. A scheduled thread will maintain control of the CPU until it has reached the end of its allotted time slice. A thread may also voluntarily relinquish its time on the CPU. It also may be forced to yield its time if preempted by a higher-priority thread[2]. When a thread reaches the end of its allotted time slice, it moves to the back of its queue for its priority level. In the event, a thread is preempted by a higher-priority thread, and said higher-priority thread has ceased execution, the preempted thread continues execution, consuming the remainder of its time slice (i.e., a preempted thread will remain at the head of the queue for its priority level)[2]. If the same procedure was applied using the FIFO scheduler, the output would be the same as the first output in task 4. However, under the RR scheme each thread does not have enough time to complete a single execution in the allotted CPU time, and thus the results in figure 6 are obtained. An unusual behaviour was also noticed during testing. The output of the threads re-order themselves after the first RR cycle. So the iteration count was increased serveral times, yet the same output was observed. The expected output was "abcabcabcabcabcabcabcabcabc" as all three tasked have the same priority level.

```c
void *threadA(void *arg){
  int j;
  int i;
  for(j=1;j<=10;j++){
    printf("a");
    for (i=1; i<= 10000000;i++);}
  return (NULL);}
```

```
Start time is: 1614979903 s 792209515 ns
main() waiting for threads
abccabcabcabcabcabcabcabcabcab
main() reporting  all threads have terminated
```

Finally, the priority of threadA is set higher than the remaining two subthreads as seen in the output below. It is clear from the obtained output that much like the FIFO policy, the RR policy also prioritises CPU space based on priority levels. Both the FIFO policy and RR policy are relatively similar. However, the FIFO does not allow time slicing. In both policies, the current running thread may be preempted for several reasons. For instance of the policy of another thread was raised to a higher priority level than the currently executing thread. Moreover, if the policy of the currently running thread was decreased to a lower value than some other runnable thread, it would also be preempted under both the FIFO and RR policy schemes. Finally, if a higher priority thread that was previously blocked suddenly becomes unblocked (e.g. an I/O operation it was waiting for was completed) the "current running" thread would be preempted[2]. Under the FIFO once a thread gains access to the CPU, it will continue to execute unless it voluntarily relinquishes the CPU or is preempted by a higher priority task. In the first instance it will be placed at the back of its priority level queue. In the second, it remains at the front of its priority level queue until the thread that preempted it finishes executing. In the output below, threadA has the highest priority level, consequently it is allocated more time slices to complete execution first. The reverse would be observed if the threadA was given the lowest priority level.

```
Start time is: 1615074249 s 293546711 ns
main() waiting for threads
aaaaaaaaaacbcbcbcbcbcbcbcbcb
main() reporting all threads have terminated
```

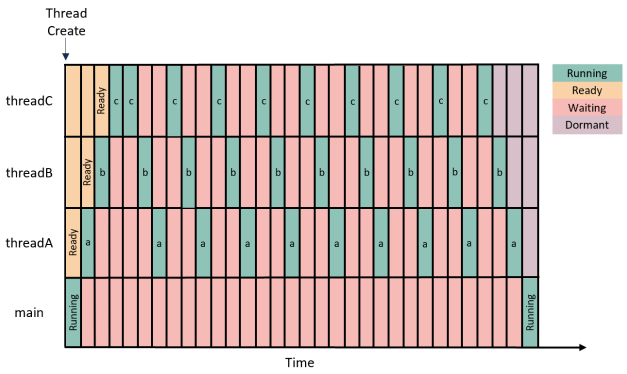

Figure 6: Task 6: Round-Robin Scheduler

References

[1] Barney,B.,(2021).POSIX Threads Programming. Available at: https://computing.llnl.gov (Accessed: 15 February 2021).

[2] Kerrisk, M.,(2010). The Linux programming interface: a Linux and UNIX system programming handbook.

[3] Howard, M., (2021).*Real Time Systems Control*, Lecture Notes, Real Time Systems Control 7CCSM-RTS. King's College London.

[4] Burns, A. and Wellings, A.J., (2001). *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education.

[5] The Open Group.,(2017).*The Open Group Base Specifications Issue 6*.Available at: https://pubs.opengroup.org (Accessed: 20 February 2021)