Real-time Programming with pthreads

1. When you use the command: sudo ./multithread
   a. **What does it output?** – Outputs a series of the letters a,b and c: aaaaaaaaaabbbbbbbbbccccccccccc
   b. **What does this program do?** – The program demonstrates the use of multi-threading and scheduling using POSIX threads (pthreads). It does this by creating three threads, threadA, threadB, and threadC. Each thread prints a character ten times to the console. The main thread sets the processor affinity, finds the priority limits, and sets the priority and policy of itself and the created threads. The main thread then waits for the three created threads to terminate using pthread_join() function, and then it exits the program. Overall, the program demonstrates the creation and management of threads, as well as scheduling and prioritization of threads using POSIX threads.
   c. **How scheduling gives rise to the observed behaviour?** - This program's observed behaviour is that each thread prints its sequence of characters in turn, with thread A printing the letter 'a,' thread B printing the letter 'b,' and thread C printing the letter 'c'. The threads print their characters in a deterministic and sequential order. This is because each thread has the same priority level and there are no other threads with a higher priority. As a result, the operating system scheduler allots an equal amount of CPU time to each thread. Because the threads are created in a sequential order, the first thread, thread A, receives the most CPU time. When thread A completes its task, CPU time is allocated to thread B, and so on. The threads are scheduled in a round-robin fashion, with each thread allowed to complete its character sequence before handing over the CPU to the next thread. This is why the program's output is deterministic and follows a specific order.
2. Modifying threadA with the provided instructions
   threadA was modified in the following way:

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

   a. **What does it output?** – Outputs a series of the letters a,b and c but in a different order: aaaabbbbbbbbbaaaaaaacccccccccc
   b. **Explain the effect of this change on the observed behaviour** – When j=5, it modified the priority of threadA, giving threadB priority over threadA. Until the end of its iteration, it kept printing ‘b’ before returning to threadA and then threadC.
3. Using the nanosleep command
   threadA was modified in the following way:

```
void *threadA(void *arg)
{
  int j;
  for(j=1;j<=10;j++){
  if(j==5){
    struct timespec delay = {0,1000000};
    nanosleep(&delay, NULL);
  }
  printf("a");
  }
  return (NULL);
}
```

a. What does it output? – aaaabbbbbbbbbbcccccccccccaaaaaa
b. Explain the effect of this change on the observed behaviour – The program completely stops threadA from printing ‘a’ mid process for the duration of 1 millisecond. In this time threadB and threadC are executed. When 1 millisecond is finished threadA is executed until its function is finished. This happens because nanosleep is used to the suspend the execution of the program for specific amount of time.

4. When you use the command: sudo ./critical
    a. **What does it output?** – aaaaaAAAAAbbbbbBBBBBAAAAAaaaaaBBBBBbbbbb
    b. **What does the program do?** – This application shows how to use C language mutexes for mutual exclusion. It produces threadA and threadB, two threads. Uppercase and lowercase letters are printed to the console by both threads. Critical output is indicated by uppercase letters, whereas non-critical output is shown by lowercase letters. A mutex is used to synchronise the threads and guarantee that they execute the crucial portions of the code atomically. The threadA routine prints five lowercase letters "a", locks the mutex to enter the critical section, prints five uppercase letters "A", raises the priority of threadB above threadA, prints five more uppercase letters "A", unlocks the mutex to leave the critical section, and then prints five more lowercase letters "a". The threadB function is similar to threadA except that it prints five lowercase characters "b" instead of "a" and then five uppercase letters "B" instead of "A". Five more uppercase letters "B" are printed after that, followed by five lowercase letters "b". The main routine generates threadA and threadB, initialises the mutex, sets the priority and policy of the main thread, detects the priority limits, sets the priority and policy of the processor affinity, and waits for the threads to finish before deleting the mutex.
    c. **How scheduling gives rise to the observed behaviour?** - The two threads are scheduled using a First In First Out (FIFO) scheduling mechanism and have different priority, which results in the observed behaviour. The operating system schedules both threads in a round-robin method because they both start out with the same priority level (priority min). This allows each thread to run for a set length of time before moving to the other thread. While threadB blocks and waits for the mutex to become available, threadA acquires the mutex when it enters the critical zone. ThreadB is blocked and can no longer run until threadA releases the mutex. At this time, threadB's priority is increased, making it more important than threadA. Because of this, the operating system favours threadB and lets it run for a longer time before switching back to threadA. Given that threadB is running for a longer amount of time, it has a higher chance of acquiring the mutex before threadA. There are therefore more uppercase "B" outputs and fewer uppercase "A" outputs because threadB is more likely to enter the crucial zone before threadA. The order in which the threads execute and the amount of time each thread is given to run are both impacted by the operating system's scheduling behaviour. The output's frequency and order are consequently impacted by this.
5. Modify the code so that the mutexes are no longer commented out.
    a. **What does it output?** – aaaaaAAAAAbbbbbAAAAABBBBBaaaaaBBBBBbbbbb
    b. **Explanation of execution of the modified program** – The programme began by locking the mutex when thread A entered the critical region. When thread A keeps running, it increases thread B's priority and switches to executing thread B. This time, the mutex hasn't been unlocked, thus thread B must wait while thread A unlocks the mutex before it may access the critical region. As a result, uppercase As are shown, and after thread A has finished executing, it opens the mutex to let thread B enter its own critical region, which is where uppercase Bs are shown. This is the same for when the process switches to thread B. Thread B has locked the mutex so thread A cannot access its critical region so it doesn’t end up displaying uppercase As and the program finished but printing out the last uppercase and lowercase Bs.