**HOME ASSIGNMENT – 02**

Priyanka Shah (121038)

**Q1.**

|  |  |
| --- | --- |
| Transition | Event |
| Ready -> Run | When it is the next process in the ready queue to be executed. |
| Ready -> Non-resident | When the memory is full and some process needs to be swapped out. |
| Run -> Ready | It occurs when the time for the process gets over before the whole process is executed. |
| Run -> Blocked | If there is a request of I/O by the process and it needs to wait. |
| Blocked -> Ready | When the request for the I/O for the process gets satisfied. |
| Blocked -> Non-Resident | When the memory gets full and some process needs to be swapped out. |

**Q2.**

|  |  |  |  |
| --- | --- | --- | --- |
| Process | T = 22 | T = 37 | T = 47 |
| P1 | Blocked for I/O | Ready Running | Ready Running |
| P3 | Blocked for I/O | Ready Running | Ready Running |
| P5 | Ready Running | Blocked for I/O | Ready Suspended |
| P7 | Blocked for I/O | Blocked for I/O | Blocked for I/O |
| P8 | Ready Running | Ready Running | Exit |

**Q3**. The output if the fork succeeds will be:

0

<child pid>

**Q4.** Following are the reasons why switching between threads is easier than switching between processes:

* Memory management for threads is much simpler than that for the processes.
* Filling the thread control block is faster than the process control block because it maintains less fields.
* The amount of information to be moved for threads is much less than that for the processes.
* Threads share file so switching between them is faster while process don’t so they have to switch taking the files, taking more time to switch.

**Q5.** The three advantages of UTL (User Level Thread) over KLT (Kernel Level Thread) are:

* They are much faster than the KLT.
* It is cheaper and simpler to make threads at user level.
* It is easier to access and modify threads at user level.

**Q6.** The disadvantages of UTL over KTL:

* They need non-blocking system.
* There is a lack of kernel support.

**Q7.** The user process and kernel process are separate so the user thread structure will not be visible to the kernel. The kernel is responsible for scheduling the processes, and it will continue to schedule the process as a single unit till it assigns a single execution state to the whole process. So if one thread is blocked the rest of the threads of the process need to be blocked. Else they will be executed and not scheduled.

**Q8.**  In multithread programs if one of the KLT or ULT is blocked then the other threads can still run and execute while on a uniprocessor if one thread is blocked the other thread will have to wait till this thread is released. Thus multithread is faster than uniprocessor.

**Q9**. Threads of a process are governed by the process. If the process is exited then the threads under it will also exist as there will be no process to govern them.

**Q10.**

|  |  |
| --- | --- |
| Competing Process | Cooperating Process |
| Competing process are the process that work independent of any other process in the system. | Cooperating process are the processes that are dependent on the other processes in the system. |
| A competing process would compete for the resource that it needs to execute. | A cooperating process would share the resources it needs with the other process that needs the same resource for execution. |
| A competing process would finish its task on its own. | A cooperating process would work with other processes to finish its task. |
| All the processes are isolated from each other. | The processes would communicate and even share with each other. |

**Q11.**

|  |  |
| --- | --- |
| Strong Semaphore | Weak Semaphore |
| A strong semaphore would specify the order in which the processes would be removed from the waiting queue. | A weak semaphore will not specify the order in which the processes should be removed from the waiting queue. |
| It provides more efficiency. | It is random and does not provide good efficiency. |
| All operating systems majorly use strong semaphore technique. | There are very few places where the technique of weak semaphore is used. |

**Q12**. Monitor is a set of programs that together help achieve mutual exclusion through the lock system. It allows the threads to wait for some event to occur and assure mutual exclusion between them. With the help of monitors only one thread will be executed at a time in a monitor. For any other thread to execute it would first have to acquire the lock for the monitor. It is even helpful for multiprogramming.

**Q13.**

|  |  |
| --- | --- |
| Blocking | Non-Blocking |
| If the sender is blocked then they have to wait till the previous message that they sent is received by the receiver. | If the sender is non-blocking then it does not have to wait and can keep sending. |
| If the receiver is blocked then it has to wait till it receives the previous message before receiving any other message after that. | If the receiver is non-blocking then it does not have to wait and can keep sending. |

**Q14**. No, busy waiting will be more efficient than blocking wait in certain scenarios. When the expected wait time is less than the time needed to preempt a thread and re-schedule it the process of busy waiting would be much more efficient than the blocking wait.

**Q15.** Both the set of definition for the semaphore will give the same result and so they can be used in place of one another in the program.

**Q16.**

**Code for the Santa:**

Santa\_Sem.wait() //santa sleeping

mutex.wait()

if reindeer==9: // if all the 9 reindeer arrive

Prepare\_Sleigh() // Santa prepares the sleigh for his journey

Reindeer\_Sem.signal(9) //tells the reindeer they have to be prepared

else if elves= 3: // if 3 elves arrive for help

Help\_Elves() //give elves the help

mutex.signal()

**Code for reindeers:**

mutex.wait()

Reindeer += 1

if Reindeer == 9: //if 9 reindeers are ready

Santa\_Sem.signal() //signal the santa they are ready

mutex.signal()

Reindeer\_Sem.wait()

Get\_Hitched() //get hitched with the santa

**Code for elves:**

Elf\_Tex.wait()

mutex.wait()

Elves += 1

if Elves == 3: //if 3 elves ready

Santa\_Sem.signal() //tell santa 3 elves are ready

else Elf\_Tex.signal()

mutex.signal()

get\_Help() //get help from santa

mutex.wait()

Elves -= 1

if Elves == 0:

Elf\_Tex.signal()

mutex.signal()