**Home Assignment**

**Erali Shah**

**121012**

1. States:

(2) RUN to READY can be caused by a time-quantum expiration or priority

Interrupt

(5)READY to NONRESIDENT occurs if memory is overcommitted, and a

process is temporarily swapped out of memory

(1)READY to RUN occurs only if a process is allocated the CPU by the dispatcher

(3) RUN to BLOCKED can occur if a process issues an I/O or other

kernel request.

(4) BLOCKED to READY occurs if the awaited event completes

(perhaps I/O completion)

(6) BLOCKED to NONRESIDENT - same as READY to NONRESIDENT.

1. T = 22: P5, P8 ready/running  
   P1, P3, P7: blocked for I/O  
   T = 37: P1, P3, P8: ready/running  
   P5: blocked suspend (swapped out)  
   P7: blocked for I/O  
   T = 47: P1, P3, P5 ready/running  
   P7: blocked for I/O  
   P8: exit
2. The outputs are:
   1. Pid = 0, if child process is created
   2. Pid> 0, if parent process is created
   3. Pid< 0, if child process can be created.
3. (1) Memory management is much simpler for threads than for processes.

(2) Time required is less

(3) Lesser information is transferred during thread switching.

1. (1) Thread switching does not require kernel mode privileges.

(2) ULTs can run on any OS.

(3) Scheduling can be application specific.

1. (1) In ULT, when a thread is blocked, it blocks whole process. (2) In pure ULT, a multithreaded application cannot take advantage of multiprocessing.
2. User process functions separately from Kernel processes. The kernel continues to schedule the process as a unit and assigns a single execution state(Ready, Running, Blocked, etc.) to that process "Hence once one thread is blocked, the whole process is blocked and consequently all threads in that process are blocked.
3. The problem is process spends a considerable amount of its waking hours waiting for I/O to complete. In a multithreaded program, one KLT can make the blocking system call, while other KLT can continue to run on uniprocessors, a process that would otherwise have a block for all those calls that can continue to run its other thread.
4. No, if process exits, all of its threads exit.
5. Competing processes compete for resources while cooperating processes share resources
6. Strong semaphore specifies in which order processes are removed from the waiting queue while weak semaphore doesn’t.
7. a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.
8. Blocking send or receive for messages means that the sender and reciever both are blocked until the message is delivered. In a nonblocking send/recieve, neither the sender orreciever has to wait.
9. Busy waiting can be more efficient if the expected wait time is shorter than the time it takes to preempt and re-schedule a thread.
10. They are equivalent in the perspect of functionality. Here s.count stands for number of process which can run simultaneously when it turns to 0, it means every upcoming process which invoked semwait should be blocked. The difference is semsignal primitive. The semaphore determining if to unblock process, depends on process queue instead of value of s.count. when all blocked process are released, the s.count may start to increase if semsignal was invoked. Hence the codes cannot be substituted in each other’s place.