**OS Home Assignment – 2**

Name - Aakanksha Agarwal

Roll no. – 121002

1. Events that can cause the above transitions

* **READY to RUN:** If a process is allocated to the CPU by a dispatcher.
* **READY to NONRSIDENT:** If memory is over committed and a process is temporarily swapped out of the memory
* **RUN to READY:** This is caused by Thin Quantum expiration
* **RUN to BLOCKED:** If a process issues on I/O and other kernel requests.
* **BLOCKED to READY:** If the awaited event
* **BLOCKED to NONRESIDENT:** If memory is over committed and a process is temporarily swapped out of the memory. It is same as the transition from READY to NONRESIDENT.

1. For time 22 -

* P1: Blocked for I/O.
* P3: Blocked for I/O.
* P5: Ready/Running.
* P7: Blocked for I/O.
* P8: Ready/Running.

For time 37 -

* P1: Ready/Running.
* P3: Ready/Running.
* P5: Blocked/Suspended.
* P7: Blocked for I/O.
* P8: Ready/Running.

For time 47 -

* P1: Ready/Running.
* P3: Ready/Running.
* P5: Ready/Suspended.
* P7: Blocked for I/O.
* P8: Exits.

1. On execution of the code provided, the ***pid*** of the child process is returned in the parent’s thread of execution, and a ‘**0**’ is returned in the child process’s thread of execution.  
     
   Output:

0 OR <child ***pid***>

<child ***pid***> 0

1. Mode switch between threads may be cheaper than mode switch between processes because of the following reasons:

* The control blocks for processes are larger than for threads, so the amount of information to move during the thread switching is less than for process context switching.
* The major reason is that the memory management is much simpler for threads than for processes.
* Threads do not have to do accounting, so keeping the thread control block consistent is much faster.
* Threads share files, so when mode switch happens in threads, these information stay the same and threads do not have to worry about it and that makes the mode switch much faster.

1. Advantages of ULTs over KLTs are:

* Thread switching does not require kernel mode privileges because all the thread management data structure are stored within the user address space of a single process which saves the overhead of the mode switches i.e. from user to kernel and vice-versa.
* Scheduling can also be application specific. The algorithm for scheduling can be tailored to the application without disturbing the underlying OS schedules.
* ULTs can run on any Operating system without making any changes to the underlying kernel to support the ULTs. The thread library is a set of the application level functions shared by all the applications.

1. Disadvantages of ULTs over KLTs are:

* In a typical OS, many system calls are blocking. As a result, when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked.
* In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time.

1. ULT thread structure of a process is not visible to the OS/kernel which schedules on the basis of the process. The kernel continues to schedule the process as a unit and assigns execution state (Ready, Running, Blocked) to that process. Hence, once a thread is blocked the entire process is blocked and consequently all the threads in that process ae also blocked.
2. One to one mapping between the ULT and KLT allows on or more threads within the process to issue blocking system call while continue to run, because KLT in multi-threaded program enables at least on thread to issue a blocking system call independently without the influence of other threads and thereby allowing other threads to uninterruptly continue with their execution, however in single threaded system counter parts of the multi-threaded program machine generally spends a lot of time waiting for the I/O operations.
3. Yes, the thread will also terminate if the process it is running in terminates. The thread is dependent upon the processes it is running. If the processes die the thread dies.
4. Competing processes -They compete for resources. For example, two independent applications may both want to access the same disk or file or printer. The OS must regulate these accesses.

Cooperating Processes – They share resources. May or may not be aware of each other. Some processes are designed to cooperate together (jointly) on the same activity and share resources. They may also be aware of each other by process id.

1. Difference between strong and weak semaphores is that the strong semaphores specifies in which order the processes are being removed from the waiting queue whereas the weak semaphores don’t. For example in FIFO.
2. A monitor is a synchronization construct which allows the threads to have mutual exclusion. It provides equivalent functionality as that of the semaphores and are easier to control.
3. Blocking send, blocking receive: Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a rendezvous. This combination allows for tight synchronization between processes.

Nonblocking send, blocking receive: Although the sender may continue on, the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that exists to provide a service or resource to other processes.

Nonblocking send, nonblocking receive: Neither party is required to wait.

1. Busy waiting can be more efficient if the expected wait time is shorter than the time is shorter than the time it takes to pre-empt and reschedule a thread. This is common for multiprocessors.
2. Both are equivalent in terms of functionality.

s.count is for the number of process that can run simultaneously. When it turns to 0 it implies every upcoming process which invoked semwait() should be blocked.

1. Santa Claus problem:

loop

{

semwait(santa);

if(all\_reindeer\_ready)

{

for all\_waiting\_reindeer

{

semSignal(reindeer\_wait);

}

for all\_ reindeer

{

semSignal(harness) ;

}

Deliver Tops;

for all\_reindeer

{

semSignal(unharness);

}

}

Else if(all\_elves\_ready)

{

for all\_waiting\_elves

{

semSignal(elf\_wait);

}

for all\_ reindeer

{

semSignal(invite) ;

}

consult;

for all\_reindeer

{

semSignal(unharness);

}

}

}