**Home Assignment 2**

**1.**

* RUN to READY can be caused by a time-quantum expiration
* READY to NONRESIDENT occurs if memory is overcommitted, and a process is
* temporarily swapped out of memory
* READY to RUN occurs only if a process is allocated the CPU by the dispatcher
* RUN to BLOCKED can occur if a process issues an I/O or other kernel request.
* BLOCKED to READY occurs if the awaited event completes (perhaps I/O
* completion)
* BLOCKED to NONRESIDENT - same as READY to NONRESIDENT.

**2.**

At time 22

P1: blocked for I/O

P3: blocked for I/O

P5: ready/running

P7: blocked for I/O

P8: ready/running

At time 37

P1: ready/running

P3: ready/running

P5: blocked suspend

P7: blocked for I/O

P8: ready/running

At time 47

P1: ready/running

P3: ready/running

P5: ready suspend

P7: blocked for I/O

P8: exit

**3.**

0

<child pid>

or

<child pid>

0

**4. List reasons why a mode switch between threads may be cheaper than a mode switch between processes.**

Switching process requires OS to process more information.

Memory is shared by threads, so there's no need to exchange memory or data during thread creation or switching.

Thread switching does not require kernel to get involved, which in turn saves time on switching user to kernel mode.

Threads do not have to worry about accounting, etc, so do not have to fill out all the information about accounting and other process specific information in their thread control block, 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 (similar to accounting information) and that makes the mode switch much faster.

**5. List three advantages of ULTs over KLTs.**

**1.** Thread switching does not require kernel mode privileges because all of the

thread management data structures are within the user address space of a single

process. Therefore, the process does not switch to the kernel mode to do thread

management. This saves the overhead of two mode switches (user to kernel; kernel

back to user).

**2.** Scheduling can be application specific. One application may

benefit most from a simple round-robin scheduling algorithm, while another

might benefit from a priority-based scheduling algorithm. The scheduling

algorithm can be tailored to the application without disturbing the underlying OS

scheduler.

**3.** ULTs can run on any operating system. No changes are required to

the underlying kernel to support ULTs. The threads library is a set of application level

utilities shared by all applications.

**6. List two disadvantages of ULTs compared to KLTs.**

**1.** In a typical operating system, many system calls are blocking. Thus, when a ULT

executes a system call, not only is that thread blocked, but all of the threads within

the process are blocked.

**2.** 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.

**7. In the discussion of ULTs versus KLTs, it was pointed out that a disadvantage of ULTs is that when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked. Why is that so?**

Because, with ULTs, the thread structure of a process is not visible to the operating

system, which only schedules on the basis of processes.

**8. Consider an environment in which there is a one-to-one mapping between user-level threads and kernel-level threads that allows one or more threads within a process to issue blocking system calls while other threads continue to run. Explain why this model can make multithreaded programs run faster than their single-threaded counterparts on a uniprocessor computer.**

The issue here is that a machine 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 the other KLTs can continue to run. On uniprocessors,

a process that would otherwise have to block for all these calls can continue to run

its other threads.

**9. If a process exits and there are still threads of that process running, will they continue to run?**

No. When a process exits, it takes everything with it—the KLTs, the process

structure, the memory space, everything—including threads.

**10. What is the distinction between competing processes and cooperating processes?**

Competing processes need access to the same resource at the same time, such as a

disk, file, or printer. Cooperating processes either share access to a common object,

such as a memory buffer or are able to communicate with each other, and

cooperate in the performance of some application or activity.

**11. What is the difference between strong and weak semaphores?**

A strong semaphore requires that processes that are blocked on that semaphore

are unblocked using a first-in-first-out policy. A weak semaphore does not dictate

the order in which blocked processes are unblocked.

**12. What is a monitor?**

A monitor is a programming language construct providing abstract data types and

mutually exclusive access to a set of procedures

**13. What is the distinction between *blocking* and *nonblocking* with respect to messages?**

There are two aspects, the send and receive primitives. When a *send* primitive is

executed in a process, there are two possibilities: either the sending process is

blocked until the message is received, or it is not. Similarly, when a process issues

a *receive* primitive, there are two possibilities: If a message has previously been

sent, the message is received and execution continues. If there is no waiting

message, then either (a) the process is blocked until a message arrives, or (b) the

process continues to execute, abandoning the attempt to receive.

**14. Is busy waiting always less efficient (in terms of using processor time) than a blocking wait? Explain.**

On average, yes, because busy-waiting consumes useless instruction cycles.

However, in a particular case, if a process comes to a point in the program where it

must wait for a condition to be satisfied, and if that condition is already satisfied,

then the busy-wait will find that out immediately, whereas, the blocking wait will

consume OS resources switching out of and back into the process

**15. Consider the following definition of semaphores:**

**Compare this set of definitions with one given below. Note one difference: With the preceding definition, a semaphore can never take on a negative value. Is there any difference in the effect of the two sets of definitions when used in programs?**

**That is, could you substitute one set for the other without altering the meaning of the program?**

The two are equivalent. In the definition of Figure 5.8, when the value of the

semaphore is negative, its value tells you how many processes are waiting. With

the definition of this problem, you don't have that information readily available.

However, the two versions function the same.