Homework 2 Solution

Do the following Problems:
1. What is a critical section? What are the three requirements for correctly solving a critical section problem?

Answer:
A section of code in which only 1 process can be executing at any given time because it uses shared variables. To solve this problem, you are essentially solving a locking/mutual exclusion problem, so you must be able to 1) lock the process before entering the critical section, 2) unlock the process when you are done executing the critical section, and 3) wait to use the critical section of code if it is already locked (4) never have to wait ‘forever’ when trying to get to your critical section.

2. Both busy waiting and blocking methods can be used as means to address critical section problems and process synchronization. Describe how blocking is different from busy waiting, and discuss the advantages and disadvantages of each.

Answer:
Busy waiting means a process simply spins (does nothing but continue to test its entry condition) while it is waiting to enter its critical section. This continues to use (waste) CPU cycles, which is inefficient. Additionally, if a low priority process is in its critical section, and a high priority process goes into busy waiting, it may never get to its critical section. On a single CPU, busy waiting becomes a circular issue, so blocking would make more sense, as it only requires a single call to the OS to change state to Ready/Run, once it is notified that blocking process is complete. For blocking solutions, we would need to be careful because there is a chance that an incremental counter might be corrupted; processes can go into a sleep state and never awake. Additionally, while busy waiting may waste cycles on a single processor, blocking may waste cycles on a multiprocessor.

3. Do semaphores solve the problem of process synchronization completely? What are some of the things that might still go wrong in a solution that requires synchronization?

Answer:
The primary outstanding concern with use of semaphores is that they can lead to race condition problems; additionally, they do not naturally protect against deadlock. Semaphores are tools only, and as such can be implemented incorrectly, such that they do not guarantee the sequence in which events occur, which may lead to a problem with synchronization.

4. What is a monitor? Explain how it can be used to solve a synchronization problem.

Answer:
A monitor is a self- contained data structure which contains atomic operations, condition variables, and shared data. Can be used to solve a synchronization problem, because the shared data can only be accessed by the procedures in the monitor, and the process can
control the sequence in which the procedures access the shared data, thereby enforcing synchronization (though not necessarily mutual exclusion).

5. It was pointed out that two advantages of using multiple threads within a process are (1) less overhead involved in creating a new thread within an executing process than in creating a new process (explain concisely why), and (2) communication among threads within the same process is simplified (explain concisely how). Is it also the case that a context switch between two threads within the same process involves less overhead than a switch between two threads in different processes?

Answer:
Yes, because more state information must be saved to switch from one process to another.

6. Suppose that the OS needs to choose and dispatch another process to execute and that, at this time, there are processes in both the ready state and the ready-suspend state. Furthermore, suppose that at least one process in the ready-suspend state has higher scheduling priority than any of the processes in the ready state. Two extreme policies for deciding which process to execute are: (1) Always dispatch a process in the ready state. (2) Always dispatch the highest-priority process. What is a concern with each of these two policies? Is there an intermediate policy that tries to balance these concerns?

Answer:
Obviously the concern with (1) above is that if a process is assigned a higher priority (e.g. an application calculation), it really should go before something of a lesser priority (sending something to a printer). The potential problem with (2) above is that if you are *always* dispatching highest priority process, you may run into issues with too much swapping, or you may end up with processes all needing to access the same data at the same time, therefore all blocking on each other.

Last Question:

- a. List three reasons for wanting to use multiple threads in a process.
- b. If a process exits and there are still threads of that process executing, will they continue to execute? Explain.
- c. What is one advantage and one disadvantage of a process using user-level threads over a process using kernel-level threads?
- d. Suppose two processes are executing, both of which are using multiple kernel-level threads. Is a context switch between two threads within one processes less work than a context switch between two threads in different processes? Explain.
- e. Suppose two processes are executing, both of which are using multiple user-level threads. Is a context switch between two threads within one processes less work than a context switch between two threads in different processes? Explain.
Answer:

- a. simplifies program structure
  improves performance in presence of blocking
  make use of multiple processing units
- b. No. When a process exits, it takes everything with it. This includes the process structure, the memory space, etc. including threads.
  (The process is the execution environment; without it, a thread can not execute.)
- c. advantage: don't need kernel to understand threads faster switching, since kernel not involved
  disadvantage: if one thread blocks, all threads blocked
  process can't take advantage of multiple processors
- d. PARTIAL ANSWER:
  YES! More state information needs to be switched when threads are in different processes.
  FULLER ANSWER:
  context switch between two threads in the same process:
  only need to store/load the TCB information
  context switch between two threads in different processes:
  above, plus need to store/load the PCB information
- e. PARTIAL ANSWER:
  YES! Process switch occurs when threads are in different processes.
  FULLER ANSWER:
  context switch between two threads in the same process:
  Handled by the dispatcher within your program
  OS doesn't need to do anything
  context switch between two threads in different processes:
  must process switch between two processes