CS471: Operating System Concepts

Homework #3
Solution
Points: 20

Question 1. [Points 8]

(a) Given the following solution for the critical section problem for two processes (the code shown is for processes Pi), show why it does not satisfy the mutual exclusion requirement. Here, **lock is a shared** variable initialized to **FALSE**. (Hint: Indicate one scenario where the mutual exclusion progress requirement is violated.)

```
do {
        while (lock);
        lock = TRUE;

            Critical section

        lock= FALSE;

            Remainder section
}
```

(b) What happens if lock is initialized to TRUE in the above code?

Answer:

(a) It does not satisfy the mutual exclusion requirement since multiple processes could execute the lock=TRUE statements and enter the critical section simultaneously as shown by the following scenario with P1 and P2.

Initially, lock=FALSE;
Time
0 P1 executes while(lock); since lock=FALSE; P1 now goes to the next statement; before it could execute lock=TRUE, P1 is preempted;
1 P2 executes while(lock); since LOCK=FALSE, P2 executes lock=TRUE and enters CS; P2 is now preempted;
2 P1 resumes execution; it executes lock=TRUE; and enters CS;

This violates mutual exclusion condition. In other words, since the entry section is not executed atomically, the given entry section code is not adequate to enforce mutual exclusion requirement.

(b) If lock is initialized to TRUE, then no process will be able to enter its critical section. In other words, all processes that execute the entry section will be blocked. The progress requirement will be violated.

Question 2. [Points 4] Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Answer: Spinlocks is where a process or thread continuously keeps on checking for a condition before it enters a critical section. In a single processor system, this wastes CPU time since it is stealing from the process/thread which is executing in its critical section. This further delays the waiting process from entering its critical section.

On the other hand, in a processor system, if one process executing on a processor is continuously checking on an entry section condition, it does not affect the process which is executing its critical section on a different processor.

Thus, spinlock or busy wait is suitable for multiprocessor systems and semaphores are appropriate for single processor systems.

Question 3. [Points 8] Consider a system consisting of processes P1, P2, ... , *P11,* each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.

Answer:

There could be several ways in which this code could be written. Since there are only three processes, for loop was not used in the acquire section to check on which printer is free. It is assumed that the signal() operation will check on the wait queue and signals that process with highest priority which is waiting on the condition variable printer.

Function acquire(int priority) returns a printer number to the calling process. It first checks which of the three printers is free and returns that number. If all are busy, it executes printer.wait(printer).

The release procedure releases the printer and then signals the printer condition variable.

The suggested solution is simple since it does not check whether a process is releasing a printer that it is holding or some other printer. But the procedure could be easily modified. For CS 471 HW, we do not need to complicate the solution any more than what is given below. It clearly shows ways to use a monitor in a multi resources context.

```
monitor PrinterAllocator
{
boolean busy[3];
condition printer;

int acquire(int priority) {
        boolean get=false;
        while(!get)
        {
        if (!busy[0]) {busy[0]=true; return 0;}
        if (!busy[1]) {busy[1]=true; return 1;}
        if (!busy[2]) {busy[2]=true; return 2;}
        printer.wait(priority);
        }
}
void release(int printerID) {
 busy[printerID]=false;
 printer.signal();
}

Initialization_code() {
busy[0]=false; busy[1]=false; busy[2]=false;}}
```