**CS Division – EECS Department**

**University of Central Florida**

**CGS 3763 – Operating System Concepts**

**Spring 2013 – dcm**

**Homework 5 Solutions (100 points)**

**Totally, there are 10 problems. So there are 10 points for each problem.**

Problem 5.1

Explain how volatile, nonvolatile, , and stable storage differ in cost.

*Volatile storage refers tomain and cachememory and is very fast. However, volatile storage cannot survive system crashes or powering down the system.*

*Nonvolatile storage survives system crashes and powered-down systems. Disks and tapes are examples of nonvolatile storage. Recently, USB devices using erasable program read-only memory, EPROM, have appeared providing nonvolatile storage.*

*Stable storage refers to storage that technically can never be lost as there are redundant backup copies of the data, usually on disk.*

Problem 5.2

Explain the concept of transaction atomicity.

*The concept of transaction atomicity means that the entire transaction is either successfully committed or rolled back.*

Problem 5.3

What is the meaning of the term busy-waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answers.

*Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquish the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.*

Problem 5.4

Explain why spin-locks are not appropriate for a single processor, single-core systems yet are used in multi-core and multi-processor systems.

*On a single-processor system, the thread holding a lock cannot be running while another thread is testing the lock, because only one thread/process can be running at a time.  Therefore the thread will continue to spin and waste CPU cycles until its time slice end.  That is why threads on a uni-processer should always sleep rather than spin, if they encounter a lock that is held, because it would never be released until after they were done spinning, anyway.  
  
On a multiprocessor system, multiple threads can actually be running at the same time.  Therefore, it is appropriate for a thread waiting for lock to go into a spin wait, because the lock could be released while it is waiting.*

Problem 5.5

Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single processor, single core system if the synchronization primitives are to be used in user-level programs.

*If a user-level program is given the ability to disable interrupts, then it candisable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes to execute.*

Problem 5.6

Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

*Depends on how interrupts are implemented, but regardless of how, it is a poor poor choice of techniques.  
  
Case 1 -- interrupts are disabled for ONE processor only -- result is that threads running on other processors could ignore the synchronization primitive and access the shared data  
  
Case 2 -- interrupts are disabled for ALL processors -- this means task dispatching, handling I/O completion, etc. is also disabled on ALL processors, so threads running on all CPUs can grind to a halt*

Problem 5.7

Describe two kernel structures in which race conditions are possible and show how a race condition can occur.

*Some kernel data structures include a process id (pid) management system, kernel process table, and scheduling queues.With a pid management system, it is possible two processes may be created at the same time and there is a race condition assigning each process a unique pid. The same type of race condition can occur in the kernel process table: two processes are created at the same time and there is a race assigning them a location in the kernel process table. With scheduling queues, it is possible one process has been waiting for IO which is now available. Another process is being context-switched out. These two processes are being moved to the Runnable queue at the same time. Hence there is a race condition in the Runnable queue.*

Problem 5.8

Describe how the *Swap()* instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirements.

*do {  
waiting[i] = TRUE; key = TRUE;  
while (waiting[i] && key) key = Swap(&lock, &key);  
waiting[i] = FALSE;  
/ critical section /  
j = (i+1) % n; while ((j != i) && !waiting[j])  
j = (j+1) % n;  
if (j == i) lock = FALSE; else waiting[j] = FALSE;  
n/ remainder section /  
while (TRUE);  
}*

Problem 5.9

Show that monitors and semaphores are equivalent insofar as they can be used to implement the same type of synchronization problems.

*A semaphore can be implemented using the following monitor code:  
  
monitor semaphore  
{  
int value = 0;  
condition c;  
semaphore increment() {  
value++;  
c.signal();  
}  
semaphore decrement() {  
while (value == 0)  
c.wait();  
value--;  
}  
}*

Problem 5.10

Discuss how the following scheduling algorithms discriminate in favor of short processes: (1) FCFS; (2) Round-Robin; (3) Multilevel feedback queues.

*a. FCFS—discriminates against short jobs since any short jobs arriving after long jobs will have a longer waiting time.*

*b. RR—treats all jobs equally (giving them equal bursts of CPU time) so short jobs will be able to leave the system faster since they will finish first.*

*c. Multilevel feedback queues—work similar to the RR algorithm— they discriminate favorably toward short jobs.*