Tina Peng

CMPS111 Harrison

25 April 2018

Homework 2

(6 Marks) Question 1. Briefly outline the role of the Process Control Block (PCB), listing and describing three pieces of information an Operating System might choose to store in the PCB.

The operating system uses the Process Control Block to keep track of and manage all the processes (runtime context of an executing program) in the system. Therefore, the PCB holds information such as the memory, open files, threads and executable code of a program, the state (PC, registers, and addresses) of the program, and context switching information (switching between processes).

(4 marks) Question 2. If we assume that when processes are interrupted they are placed in a queue containing all non-running processes not waiting for an I/O operation to complete, briefly describe two strategies the Operating System might adopt to service that queue. One-word answers will not suffice.

Since the queue is full of non-running processes that are not dependent on another process's completion, the operating system can use simple strategies to service the queue. The first strategy can be First Come First Served, where the OS simply works on the first process entered in the queue. It produces highly variable wait times and bursts but it's an easy way to eventually service all the processes in the queue and gets rid of the CPU hogs first. The second strategy is Round Robin, where the OS iterates through the queue and prioritizes to service threads that do a modest amount of work before yielding or waiting for the I/O first. It's generally more efficient than FCFS and eliminates the convoy effect.

(5 marks) Question 3. Outline a mechanism by which counting semaphores could be implemented using the minimal number of binary semaphores and ordinary machine instructions. Include C or pseudo code snippets if you feel this will make your answer clearer and/or more concise.

Pseudocode:

Initialize int count = k value of counting semaphores, binary semaphores mutex = 1, binary semaphores gate = 0 (or 1 if k > 0)

```
P(counting_semaphores){
    P(gate);
    P(mutex);
    count--;
    if count >  0:
        V(gate);
    V(mutex);
}

V(counting_semaphores){

    P(mutex);

    count++;

    if count == 1:

        V(gate);

    V(mutex);

}
```

Since we decrease count at the end of P() and increase it at the end of V(), count should accurately record the right amount of counting semaphores.

(4 marks) Question 4. Define the terms “race condition”, “deadlock”, and “starvation” as they relate to Operating System design and outline the relationship between deadlock and starvation.

A race condition exists when a multithreaded program accesses a shared data and multiple threads try to change the data at the same time. Due to the “check-then-act” process, the simultaneous changes lead to undesired results. A deadlock is when multiple processes are waiting on each other to finish some needed resources to proceed but are unable to proceed due to the wait. I think the round table example in the lecture was a good example: everyone picked up a chopstick on their left- resulting there to be no more chopsticks available on the table, meaning that none of the philosophers could continue their process of eating, and were in a deadlock. Starvation would be a situation where a thread is unable to make progress because they cannot regularly access a needed shared resource. Referring to the round table example, it would be like when one philosopher has access to two chopsticks but are incredibly slow at eating. The philosopher on their right would come to a literal case of starvation because they would be holding one chopstick in hand but will be required to wait for the slow eater to finish before doing anything else (and will be forced to wait an extremely long and inefficient amount of time).

Threads wait indefinitely in the case of starvation, but deadlock is more like a circular loop where all threads cannot proceed unless there is external intervention. Therefore, while deadlock can grow into a case of starvation, the vice versa case is not possible.

(6 marks) Question 5. Can the “priority inversion” problem outlined in section (3) of the background information to Lab 2 occur if user-level threads are used instead of kernel-level threads? Explain your answer.

Priority inversion occurs when a high-priority thread is waiting for a low-priority thread to finish processing. In the case of kernel-level threads, the system gets stuck (and will run forever) since low-priority threads will never finish before high-priority threads. However, the problem is avoided with user-level threads since there are no preemptive process switches to higher priority threads, and therefore any thread, regardless of priority, must be finished before the next thread can start. (User level threads can run into other problems though and can get blocked for another reason).