CMPS111-HW2

Jinxuan Jiang
jjiang17@ucsc.edu

Question 1:

Role of Process Control Block:

At any instance, a process will be having various information associated with it like identifier, state, priority, program counters, memory pointers, accounting information etc. Such information are stored in a data structure called as **Process Control block (PCB)**.

- It is an important tool that helps the OS support multiple processes and provide for multiprocessing.
- It contains sufficient information such that if an interrupt occurs, the process can begin from the point where it left later as if nothing had happened.
- The blocks are read and/or modified by every module in the OS including
    - Scheduling
    - Resource allocation
    - Interrupt processing
    - Performance monitoring and analysis
- It can be said that the process control block defines the state of OS.
- However, there comes the issue of Security; a bug in a single routine say the interrupt handler could damage the PCB which in turn can make it difficult for OS to manage the affected process.
- Changes made to the Process control block (like semantics or design changes) can affect various modules.

Three pieces of information an Operating System might choose to store in PCB:

1. Process ID: unique number identifying this process.
2. Program Counter: indicates the next program instruction to execute.
3. Registers: Stack pointer, index registers, and various other system dependent registers.

Question 2:

1. **Deadlock**:

   A set of process is in a deadlock state when every thread or process in the set is waiting for an event that can only be caused by another thread or process in the set.

2. **Race Condition**:

   A flaw in a system or process whereby the output of the process is unexpected and critically dependent on the sequence or timing of other events.

3. **Starvation:**

   A thread or process waiting forever.

Relationship between deadlock and starvation:

Deadlock => Starvation

Starvation ≠> Deadlock

Question 3:

Scheduling algorithms make sure that there is no starvation in the system and all the processes gets the fair chance to be executed. The technique is called preemption which is applied in several algorithms. For example:

1. **Shortest Remaining Time Next**, also known as shortest remaining time first (SRTF), is a scheduling method that is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute.
2. **Round-Robin Scheduling**, is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices (also

known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive).

Question 4:

Threads are the light weight process. A single process can be divided into multiple threads. The priority inversion problem will **not** occur for the threads of user level. This is because the threads with lower priority cannot be suddenly preempted. The kernel threads can have the priority inversion problem.

Question 5:

Implementing counting semaphore using binary semaphores and ordinary machine instructions. We have binary semaphore, so we can use mutex, we can guarantee thread safety for a sequence of code and we can implement a ‘protected’ code area that will only be executed by one thread at a time.

Counting semaphore using binary semaphore is shown as follows:

/*

1. P() indicates wait
2. V() indicates signal
3. A binary semaphore is one, which takes only 0 and 1 as values
4. down() operation checks to see if the value is greater than 0 and decrements the value by 1
5. up() operation increments the value of semaphore and increments the count and wakes up the process.
6. Initialize mutex = 1, blocking = 0 and count = some value n

*/

```
P(semaphore){
        Mutex = down();
        Count = count-1;
        if(count >=0){
                Mutex = up();
                Return;
        }
                Mutex = up();
                Blocking = down();
}
V(semaphore){
        Mutex = down();
        Count = count+1;
        if(count>0){
                Mutex = up();
                Return;
        }
        Blocking = up();
        Mutex = up();
}
```