Stephen Woodbury - 1429496 - swoodbur
HW2 - CMPS 111 - Spring 2018 - 4/25/2018 - Wednesday

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.*
   A: A Process Control Block(PCB) is attached to each live process for saving and restoring state when context switching. When an operating system switches processes, it updates the current thread's PCB and then it refers to the PCB of the process it's switching to. Though what exactly a PCB stores is system dependent, it's safe to assume the PCB stores the following:
   I) Process ID(PID): Stores the process's unique number which identifies this process.
   II) Scheduling Information: Stores things like a pointer to the scheduling queue(Holds the processes waiting to run on the CPU), the priority of the process to which the PCB belongs to, and scheduling parameters, things that affect the process's priority.
   III) Program Counter: Stores the next program instruction to execute for this process.

Stephen Woodbury - 1429496 - swoodbur
HW2 - CMPS 111 - Spring 2018 - 4/25/2018 - Wednesday

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.*

A: We could sort this queue many ways. Naturally, we could do first come first serve, but that's non-premptive and those processes that require a lot of CPU time may hog the CPU, even if they're not that important. It's inefficient. Listed below are some better ways to service the queue:

I) Round Robin: Each process running on the CPU is given a fixed amount of time to run, usually around 10 to 100 milliseconds. Once time is up, regardless as to if the process is done running on the CPU or not, we preempt(kick off) the process and send it back to the queue of programs ready to run on the CPU that do not need IO. This way those processes that only require a little CPU time can finish in a reasonable amount of time while those that need a lot of CPU time can still chip away at the time they need. There are weaknesses however, such as if all jobs are the same length, we waste time context switching.

II) Shortest Job First: The process with the shortest needed CPU time will run first ahead of those with longer CPU time needed. When a new process arrives that has a shorter CPU run time needed than the currently running Process, we could preempt the running process to put in place the shorter CPU time needed process. Though this is more effective than Round Robin at getting short processes completed and out of the way and for keeping mean waiting time down for the queue, it requires we know how much time the process will need on the CPU, which is hard to do as well as it may be unfair on bigger jobs.

Stephen Woodbury - 1429496 - swoodbur
HW2 - CMPS 111 - Spring 2018 - 4/25/2018 - Wednesday

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.*

A: To replace a counting Semaphore, we can create an ADT outlined below in pseudo code.

```
countingSemaphore:
  int available = n
  BinarySemaphore wait = 0
  int waitingNum = 0
  BinarySemaphore mutex = 1
```

Our countingSemaphore can be implemented with two binary semaphores and two integers.

+*'available'* is used to keep track of available resources (As a counting semaphore is typically used to control access to a limited resource). It is initially set to *'n'*, *'n'* being the number of resources we have available to us(We'd know this upon creation of our counting semaphore).

+*'wait'* is used to make the accessing process wait until a resource becomes available(Given there are none available, something that is checked for in the *'test'* call). This is why it's initially set to zero, or down, so that when we test/call P on our wait Binary Semaphore, it'll wait until we increment/call V on *'wait'* (which is done in *'increment'),* in which case, the waiting process will wake up and proceed.

+*'waitingNum'* is used to keep track of the number of processes waiting for a resource to become available. Ie, the processes that are currently waiting for V to be called on the *'wait'* binary semaphore. This is important for deciding when to call V on *'wait'*.

+*'mutex'* is used to control access/ensure mutual exclusion to code inside *'test'* and *'increment'* calls. We initially set it to one to allow the first process to get in.

Stephen Woodbury - 1429496 - swoodbur
HW2 - CMPS 111 - Spring 2018 - 4/25/2018 - Wednesday

To simulate our P and V calls for our fabricated counting semaphore, we'll create *'test'* and *'increment'* functions outlined below in psuedo-code.

```
test(countingSemaphore S):
  P(S.mutex)
  if S.available == 0:
    S.waitingNum++
    V(S.mutex)
    P(S.wait)
  else:
    S.available--
    V(S.mutex)
```

We must pass in the countingSemaphore we're working with. We first ensure we have exclusive access to the countingSemaphore we're working with, ie, that no other thread is working on countingSemaphore. Once that's ensured, we check if the number of available resources is zero, if so, we increase the number of threads waiting for a resource, we give up exclusive access to the countingSemaphore, then we set our thread to wait (Does this by calling P on 'wait' which is always zero at this stage of our *'test'* call). Given there is an available resource, we decrease the number of available resources and then give up exclusive access to our countingSemaphore.

```
increment(countingSemaphore S):
  p(S.mutex)
  if S.waitingNum > 0:
    S.waitingNum--
    V(S.wait)
    V(S.mutex)
  else:
    S.available++
    V(S.mutex)
```

We must pass in the countingSemaphore we're working with. We first ensure we have exclusive access to the countingSemaphore we're working with, ie, that no other thread is working on countingSemaphore. Once that's ensured, we check if there are any waiting threads (waiting for a resource to become available) If there are some waiting, we then decrease the number waiting by one and then increment *'S.wait'* which will let one thread proceed past the P call on *'wait'* based on first come first serve. We then relinquish exclusive control to our countingSemaphore. If there are no threads waiting, we increase the number of available resources before giving up exclusive control to our countingSemaphore.

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: Occurs when two or more threads/processes that share data want to change the data at the same time. Though the CPU scheduler decides the order in which the threads/processes access the data, we say that the threads/processes are "racing" to the data. Though technically the existence of a race condition itself causes no problem on its own, it does allow for the possibility of data inconsistency and for deadlocks to happen.
Deadlock: When a set of processes has all of its processes or threads waiting for an event which is only caused by another threads or process in the set.
Starvation: When a process or thread is waiting forever to use the CPU. It's "starved".

Relation: If a Deadlock occurs, it is because there was a race condition between the threads involved in the deadlock. Though a Deadlock infers there was a race condition, a race condition does not infer that there is a deadlock. When threads/processes are in a deadlock, the threads/processes involved are said to be starved as they will be waiting forever to access the CPU. Though a deadlock infers that the threads/processes involved are starved, it is not the case that when a thread/process is starved, that it's involved in a deadlock.

Stephen Woodbury - 1429496 - swoodbur
HW2 - CMPS 111 - Spring 2018 - 4/25/2018 - Wednesday

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.*
A: With kernel-level threads, preemption can happen as the OS can see with threads are more important. This means that our inversion problem can occur. However, when it comes to user-level threads, there is no preemption allowed. This is assuming there are no special calls developed for allowing a user-level thread to call to it's associated kernel-level thread(Given it's in a model where the user-level threads are tied to at least one kernel-level thread) to allow user-level threads to preempt. Linux does have some calls that allow this, like SIGALARM, a timer-signal. However, assuming there are no special calls like this in our OS, then a user-level thread cannot preempt.

This is important because if a thread cannot preempt, then the inversion problem can never occur. The Inversion problem occurs when a thread(B) that doesn't share a Critical section with another higher priority thread(A) runs on the CPU before that higher level thread. This can only happen when (A) is stuck waiting on a lower-priority thread(C)(Even lower priority than (B)) that it shares a Critical Section with as it runs on the CPU ((A) Can't run on CPU until the critical section is open). When (B) comes into the ready queue and given the queue is priority sorted, if it's a kernel-level thread, it will preempt (C) since it doesn't share a critical section with it. Ie, it'll run before (A), causing the Priority Inversion Problem.

This shows that our priority inversion problem cannot occur if there is no preemption allowed, which is what we are assuming for user-level threads (This bars special calls).