ADJ Assignment 2
SER 334: Operating Systems

Swanson, Douglas
dswanso6@asu.edu

Speidel, Clay
cspeidel@asu.edu

November 27, 2019

1 Question 1: Process Synchronization 1

1.1 Problem (P)

[Silberschatz, Acuña] Assume that a context switch takes time T. Propose an algoithm to determine how long a process to hold a spinlock, based on the process load of a system and T. If the spinlock is held for long, switching to a mutex lock (where waiting threads are put to sleep) would give the system better CPU utilization by decreasing wasted cycles (i.e, the busy waiting in a spinlock).

1.2 Analysis (A)

A spinlock is basically a loop that will continually check a resource to see if it is available. This spinlock is a computationally expensive operation that could possibly turn into an infinite loop if the resource is never released. A mutex is an atomic operation that will stop a thread until a resource is realeased, the difference is that this lock is not a continous operation, it will wait until a message is received that the resource is available. It would be beneficial to use both, one in the short term to determine if the resource is available now and one for in the long term, if the resource has been unavailable for a time. The first would keep the process running and would prevent the CPU from switching processes, while the current process goes dorment, thus reducing the computational time incurred by switching or starting a new process.

A context switch is the amount of time a computer needs to switch between already loaded and operating system and will be denoted by the letter T, where a process load is how long a system needs to start executing a process and will be denoted by the letter S. The system will context switch when the current process it is working on becomes dormant, this is done to optimize the cpu comutational power across multiple concurrent processes.

1.3 Design (D)

The amount of time spent using a spinlock should be less than or equal to the sum of the time spent context switching and half the time spent starting a new process. The context switching time is used because if the process goes into the mutex lock section, the system will switch to another process which will involve waiting the context switch time. Half of the process load time is used because sometimes, during the process switch, another process may need to be started due to the lack of another process currently running. The half give a good estimation figuring in that half the time a process would be loaded and the other half another process will already be running.

So in short, the spinlock should be used for an amount of time, determined by the equation presented below, that would be less than or equal to the amount of time spent context switching and loading a new process. This would make the computational power and time spent of the spin lock to be worth while before switching to the more effcient mutex lock.

1.4 Justification (J)

This design takes into account the load and switching process time which are costly operation, if the algorithm allows the current process to use a spin lock to try and aquire a resource within the same amount of time that it would take to pause the current process, on a mutex lock, and switch to another. This would keep the same process running and not perform the switch and load action twice.

2 Question 2: CPU Scheduling

2.1 Problem (P)

[Acuña] Design an effcient (i.e., Big-Oh of a polynomial) algorithm for determining the time quantum t for a round robin scheduler. The algorithm must consider the current process load and compute t, where system throughput is maximized for some interval.

2.2 Analysis (A)

To determine an efficient algorithm for determining the time quantum that a round robin scheduler will take, we must thoroughly examine the scheduler itself. A round robin scheduler is a type of CPU scheduler that organizes the processes that the CPU is tasked with running. The idea behind the round robin scheduler is that each process gets a set amount of time, the time quantum, for the CPU to perform on it before the process is interrupted. The CPU interrupts the process and switches tasks, working on the next process in the list. The process that is interrupted, if it has not finished its task, will move to the tail of the list. This ensures that each process is worked on and no process starves.

There are consequences to an improperly timed schedule as there is overhead in switched the processes that the CPU is working on. If the time quantum is too long, then it becomes more akin to a "first-come first-served" scheduler, where each process is worked on in the order it arrives in the CPU's queue. If the time quantum is too short, then the CPU will constantly be switching between tasks and will slow the progress of all processes accordingly. It is a widely agreed-upon rule of thumb that 80 percent of the processes worked on should take a shorter amount of time than the time quantum.

2.3 Design (D)

The goal is to find the time quantum that maximizes the throughput, or the number of processes completed, for an interval of time. We must first find the total time required for each process to be completed as it enters our list of processes. This would be our worst case, as it is equivalent to first-come first-served. If we let n be the number of processes and T to be our total time, then:

We let a be the average. According to our rule of thumb, 80 percent of the processes should be done within their first time quantum. Thus, if we allow t to be our time quantum, then:

Our time quantum is equal to 80 percent of our average completion time.

2.4 Justification (J)

This algorithm for finding the time quantum works for our round robin scheduler because as we add more processes, the average time for completing a process is altered, depending on the value of the time for completion. Since our time quantum is dependent on the average, it will be altered as well. This is more optimal than trying to determine the time to complete each process, sort them by said time and determine the time quantum that is greater than 80 percent of those processes, which is another, time consuming alternative.

3 Question 3: Virtual Memory

3.1 Problem (P)

[Acuña] Considering the code presented in the assignment. What page replacement scheme (of FIFO, OPR, LRU, MFU) should be used for this code?

3.2 Analysis (A)

The FIFO, first in first out page replacement scheme removes the page that has been loaded the longest in order to make room for a page that needs to be loaded. Now when a page is requested the queue of pages is checked to see if the page is already loaded.

Another page replacement scheme is the OPR, or optimal page replacement, and this scheme checks to see if a new page needs to be loaded and if it does it whould replace a page in the queue that will not be for the longest period of time. This allows the queue to keep the potentially highest used pages whilst replacing the potentially untouched pages in favor for pages that will be used more often. The downside is that foresight of how the pages will be used is required.

LRU, which stands for least recently used, is page replacement scheme that is similar to OPR but it will replace the page that has been the oldest access time, historically. On the other hand we have the final replacement scheme is the MFU, most frequently used, which will replace the page that has been used the most.

The code presented in the assignment is comparing two matrices, whose data could be spread across many different pages in memory depending on the size of the matrices in question.

3.3 Design (D)

The page replacement scheme used should depend on the best fit for the data that will cause the fewest page faults. With first-in first-out (FIFO), we run the risk of encountering Belady's anomaly. Belady's anomaly occurs in the worst-case scenario of the FIFO method, when we replace a page and then immediately need it again, causing an increase in the page-fault rate as we increase the number of frames for the pages. With FIFO, we do not know the number of frames that are allocated for this page-replacement algorithm and we also do not know the data in the matrix being examined.

Optimal Page Replacement (OPR) is the process of replacing the page that will not be used for the longest period of time. This does not work for us because the page-replacement algorithm cannot see into the future and we cannot easily tell the code what matrices to expect. If it were feasible to program, this method would cause the least amount of page faults.

Least Recently Used (LRU) replaces a page that has not been used in a while, typically measuring the time a page was last used using counters or a stack. Utilizing a stack is easier, using a doubly linked list that has a head and tail pointer. Whenever a page is used it is placed on the top of the stack. If the page is already in the middle of the stack when used again, it is simply moved to the top. When a page needs to be replaced, the page that the tail is pointing to is replaced. This is usually close to the OPR in terms of the number of page faults.

The Most Frequently Used (MFU) is like the LRU except it follows the opposite logic: that a page with the smallest count was just introduced and has not been used yet. This method is not typically used as the algorithm is expensive and the number of page faults that occur can be random.

3.4 Justification (J)

Using the descriptions of the algorithms, the LRU is the best fit for the code. Of the choices, the LRU typically has the least number of page-faults and since our code is examining matrices that we do not know the values of, we need the algorithm to do most of the work for us. We can easily implement a stack of numbers from the matrices and remove the least recently used number from the tail.