Benajmin Barnes
CIS 415
Assignment 2
Joe Sventek

1. **Suppose that a short-term CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O bound processes and yet not permanently starve CPU-bound programs?**
   I/O bound processes are known as short-burst CPU processes, in that they are completed relatively quickly once they receive the processor. When a new I/O process is scheduled, it will (unsurprisingly) have no processor time from the recent past and get scheduled. This is what leads to the algorithm favoring I/O bound processes. Although, because these processes are completed quickly (relative to CPU-bound programs), the CPU is then able to work on CPU-bound programs until it receives more I/O bound processes. This is why the algorithm will never starve the CPU-bound programs.
2. **Discuss the following pairs of scheduling criteria conflict in certain settings:**
   a. **CPU utilization and response time**
      CPU utilization is trying to keep the CPU as busy as possible. Obviously we need tasks to do, but once we have a number of tasks, we increase the utilization of the CPU by doing as few context switches as possible because these are essentially time wasted. Although by utilizing the CPU and having few context switches, this would increase response time as we work on strictly one process until we need to switch.
   b. **Average turnaround time and maximum waiting time**
      This case is similar to the two Gantt charts on page 267 in the book. To lower the average turnaround time, we schedule the short tasks first as these we complete quicker. As the book points out, unless we incorporate aging, there will most likely be some processes that starve as the shortest processes always take priority. Without aging, this would make the maximum waiting time very high.
   c. **I/O device utilization and CPU utilization**
      Similar to part A, by doing I/O device utilization we are executing many short CPU bursts, which is the exact opposite of what we want to for CPU utilization. With CPU utilization, we are trying to do as few context switches to keep the CPU busy and minimize time lost in context switches. Trying to do both I/O device utilization and CPU utilization is near impossible.
3. **In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, and new resources are purchased and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?**

a. **Increase Available (new resources added)**
   This change can be made safely under any condition. Even if all our resources are being used currently, adding resources will only help completely the processes.
b. **Decrease Available (resources are permanently removed from the system)**
   This change can be made only in specific instances. As the book said, a process can only be added if its max amount of resources needed is lower than the number of resources the system can offer. To decrease the resources by x, every process must have a max resource requirement of (number of current system resources) – (x) <= (all processes' max requirement). Also, the number of resources currently available needs to be greater than or equal to the number of resources we wish to remove. Both of these need to be present to decrease resources.
c. **Increase Max for one process (the process needs or wants more resources than allowed)**
   This is only allowed if the new max is less than or equal to the number of total resources the system can offer. If we only have 10 resources and we wish to increase one processes max to 11 from 9, we cannot do it.
d. **Decrease Max for one process (the process decides it does not need that many resources)**
   If we want to decrease a processes max by x, we can only do this when the process needs a number of resources that is greater than or equal to x. For example, say a process has a max of 5, but currently only has 1 resource. This mean it has 4 that it is waiting on. If we want to decrease its max to 3, we can do this because 2 (5-3) >= 4(resources it is currently waiting on)
e. **Increase the number of processes**
   We can do this safely whenever we want, as long as the processes being added have a max less than or equal to the total number of resources the system can offer.
f. **Decrease the number of processes**
   We can only do this safely when the processes we wish to remove are not using and resources currently.

4. **Compare the memory organization schemes of contiguous memory allocation, pure segmentation, and pure paging with respect to the following issues:**
   a. **External fragmentation**
      Contiguous memory allocation suffers from external fragmentation, particularly when using the best-fit and first-fit strategies for choosing free memory blocks.
      Pure segmentation can suffer from external fragmentation as a segment of memory is given to a process similar to the way it is in contiguous memory allocation. When the process is finished and a new one takes it's space, there is the possibility that we will have a small, unusable fragment left over.

Pure paging does not suffer from external fragmentation because each page is given the same amount of memory to store data. This comes at the expense of internal fragmentation though.

b. **Internal fragmentation**
Contiguous memory allocation can suffer from internal fragmentation when there is more overhead in keeping track of small amount of open memory. To counter for this, it ultimately takes up more memory than necessary, hence the internal fragmentation.
Pure segmentation does not allow internal fragmentation as each segment is sized just as necessary. It differs from contiguous memory allocation in that it doesn't take into account the possibility of leaving a small, unusable chunk of memory.
Pure paging suffers from internal fragmentation due to all pages being of a fixed size. It is very likely that not every process will use every spot in every page but because the space is already allocated, this leads to internal fragmentation.

c. **Ability to share code across processes**
Contiguous memory allocation does not allow sharing across processes, as each process has its base and limit, which only allow it to modify data relevant to its process.
Pure segmentation allows the ability to share code, as there may be multiple processes that are using a specific segment. For example, if one segment is a "stack", multiple processes may be working on it. While the segment may be shared, each process would have a different offset to access its particular data.
Similar to pure segmentation, pure paging allows for the sharing of code between processes.

5. **Consider a paging system with the page table stored in memory.**
   a. **If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?**
   It would take 100 nanoseconds; we must first index to the page table to get the frame number, which we then use to get the actual address. Both of these tasks require a memory access, getting us 50 x 2 = 100 nanoseconds.
   b. **If we add TLBs, and 75% of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds if the entry is present.)**
   It takes 50 nanoseconds to access the TLB, and then assuming we have a hit, we get a total of 52 nanoseconds. This happens 75% of the time. The other 25% it takes a total of 100 seconds, as we saw in part a. This gets us: 52 x .75 + 100 x .25 = 39 + 25 = 64 nanoseconds as an effective memory reference time.
6. **Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)**
   a. **TLB miss with no page fault**

The page is in the page table but is not in the TLB.

b. **TLB miss and page fault**
   The page is not in the page table and not in the TLB so we must go to the disk to get it.
c. **TLB hit and no page fault**
   The page is present in the TLB so we don't even check the page table; this explains no page fault.
d. **TLB hit and page fault**
   No such scenario can occur because the page table wouldn't need to be checked if we have a TLB hit.

7. **Consider the following page reference string: 7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1. Assume demand paging with 3 frames, how many page faults would occur for the following replacement algorithms?**
   a. **LRU replacement**
      18 page faults.
      F represents a page fault.
      7(F), 2(F), 3(F), 1(F), 2, 5(F), 3(F), 4(F), 6(F), 7(F), 7, 1(F), 0(F), 5(F), 4(F), 6(F), 2(F), 3(F), 0(F), 1(F)
   b. **FIFO replacement**
      17 page faults.
      F represents a page fault.
      7(F), 2(F), 3(F), 1(F), 2, 5(F), 3, 4(F), 6(F), 7(F), 7, 1(F), 0(F), 5(F), 4(F), 6(F), 2(F), 3(F), 0(F), 1(F)
   c. **Optimal replacement**
      13 page faults.
      F represents a page fault. This one requires more visualization than the previous two, so next to some (F)'s is a series of 3 numbers – this represents the pages currently in the table after replacement.
      7(F), 2(F), 3(F), 1(F)(2,3,1), 2, 5(F)(5,3,1), 3, 4(F)(5,4,1), 6(F)(5,6,1), 7(F)(5,7,1), 7, 1, 0(F)(5,0,1), 5, 4(F)(4,0,1), 6(F)(6,0,1), 2(F)(2,0,1), 3(F)(3,0,1), 0, 1