Assignment 3

Philip Dumaresq

40082638

COMP 346 - Operating Systems

Goswami Dhrubajyoti

Question 1

[10 marks] Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities.

Larger priority numbers indicate higher priority (e.g., priority 3 is higher priority than priority 1; priority 0 is higher priority than priority -1, etc.). When a process is waiting for the CPU in the ready queue, its priority changes at the rate of α per unit of time; when it is running, its priority changes at the rate of β per unit of time. All processes are assigned priority 0 when they enter the ready queue. Answer the following questions:

a. What is the scheduling algorithm that results when $\beta > \alpha > 0$? Explain.

This is going to be a First come first serve algorithm because when nothing is executing, the one that's been waiting longest will get to start, and it will run to completion since it's priority increases faster. This happens because both β and α are greater than 0, so they both increase in priority. However since α increases slower, a waiting process will never have higer priority than an active process.

b. What is the scheduling algorithm that results when $\alpha < \beta < 0$? Explain.

This one is going to be a last in, first out algorithm because new processes in this are going to be ran right away since they start at priority 0 and the value of α decreases. Older processes are going to start once once the active process finishes.

Question 2

[10 marks] An enhanced Round Robin CPU scheduling algorithm works as follows:

from the set of ready processes, pick the one to run which used the smallest fraction of the quantum during the last time it ran. A newly-arrived process is assumed to have used half its quantum. Also assume a constant steady stream of newly arriving processes. Answer the following questions:

a. Will the algorithm cause starvation? Explain your answer.

Yes, this can cause starvation. In a situation where you have a number of smaller processes that need to execute repeatedly, they would cause other larger processes to starve out. In

another scenario, if each process is assumed to have used half its quantum, then any process that's larger than half the quantum will never get to run since the biggest smallest process prossible is going to be every single new process coming in.

b. If your answer to part a. is yes, suggest an enhancement to the algorithm to resolve starvation

Question 3

[10 marks] Consider the version of the dining philosopher's problem in which the chopsticks are placed in the centre of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one chopstick at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

If a philosopher is not currently holding a chopstick, then when they make a request for one, check the state of the chopsticks. In the situation where there is 1 chopstick remaining and no philosopher holding two chopsticks, you do not pick up a chopstick. Any other time you can pick one up. But in this situation if you pick it up, you end up with N philosophers each holding 1 chopstick, and none of them will put it down until they eat.

Question 4

[10 marks] Consider Q.3 above. Assume now that each philosopher needs three chopsticks to eat. Resource requests are still issued one at a time. Describe some

simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers

The rule above still applies, but you need to add a second rule for when you're going to pick up your second chopstick. If you're going to pick up your second chopstick, you don't want to pick it up if there's only 1 remaining and no philosopher is currently eating. If there's only one left and no one is eating, then no chopsticks are going to get freed up and no one will get to eat.

Question 5

[10 marks] Consider a system consisting of m resources of the same type being shared by n processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

1. The maximum need of each process is between 1 resource and m resources.

2. The sum of all maximum needs is less than $m + n$.

We know that:

$$\sum \text{Need}[i] = \sum \text{Claim}[i] + \sum \text{Loan}[i] < m + n$$

If a deadlock were to occur, then all resources would need to be reserved:

$$\sum \text{Loan}[i] = m$$

and others would be waiting around for more idefinitely. In that case we would need to have

$$\sum \text{Claim}[i] < n$$

This means that there exists at least one process j that has acquired all its resources ($\text{Claim}[j] = 0$), and will be able to finish its task and release its resources again, which would ensure progress in the system - therefore, no deadlocks can occur.

Question 6

[10 marks] In a 64-bit computer system that uses pure paging with 16KB page size, if each page table entry is 4 bytes long then:

i. What is the maximum size of a page table?

$16KB = 16,000B$
$16,000/4 = 4,000$

ii. What is the maximum size of user-space main memory that can besupported by this scheme?

asd

iii. If hierarchical paging is used then what is the total level of hierarchies required? Assume that 2-level of hierarchy corresponds to an outer page table and an inner page table. Show all relevant calculations.

iv. If the inverted paging scheme is used instead of paging, with a 16KB page/frame size, then what is the maximum size of the inverted page table considering the same memory size calculated in (ii) above?

Assume that each entry of the inverted page table is 4 bytes long.

$$\begin{aligned}\text{Number of frames} &= \text{Physical mem size} \,/\, \text{Frame size} \\ &= 2^{64}/2^{14} \\ &= 2^{50} \text{ frames}\end{aligned}$$

$$\begin{aligned}\text{Inverted table size} &= \text{Number of Frames * Page table entry size} \\ &= 2^{50}/2^{2} \\ &= 2^{52} \text{ bytes}\end{aligned}$$

Question 7

[10 marks] Consider a demand-paged system where the page table for each process resides in main memory. In addition, there is a fast associative memory (also known as TLB which stands for Translation Look-aside Buffer) to speed up the translation process. Each single memory access takes 1 microsecond while each TLB access takes 0.2 microseconds. Assume that 2% of the page requests lead to page faults, while 98% are hits. On the average, page fault time is 20 milliseconds (includes everything: TLB/memory/disc access time and transfer, and any context switch overhead). Out of the 98% page hits, 80 % of the accesses are found in the TLB and the rest, 20%, are TLB misses. Calculate the effective memory access time for the system

$$\text{Let sma} = \text{Single memory access} = 1\mu\text{s}$$
$$\text{Let ta} = \text{TBL access} = 0.2\mu\text{s}$$
$$\text{Let at} = \text{access time} = \text{sma} + \text{ta} = 1.2\mu\text{s}$$
$$\text{Let prf} = \text{page faults} = 2\%$$
$$\text{Let pft} = \text{page fault time} = 20,000\mu\text{s}$$
$$\text{Let th} = \text{TLB hit} = 80\%$$
$$\text{Let tm} = \text{TLB miss} = 20\%$$

$$\begin{aligned}\text{Let eat} &= \text{Effective Access Time} \\ &= \text{prf * pft} + (\text{100-prf}) * ((\text{th * at})) + (\text{tm * 2 * at})) \\ &= 0.02 * 20,000 + 0.98 * ((0.8 * 1.2) + (0.02 * 2 * 1.2)) \\ &= 400 + 0.98 * (0.96 + 0.048) \\ &= 400.98 * 1.008 \\ &= 404.18784\end{aligned}$$

Question 8

[10 marks] Consider the two dimensional integer array A:

```
int A[][] = new int[100][100];
```

Assume that page size of the system = 200 bytes,and each integer occupies 1 byte. Also assume that `A[0][0]` is located at location 200 (i.e., page 1) in the paged memory system, and the array is stored in memory in row-major order (i.e., row by row). A small process that manipulates the array resides at page 0 (locations 0 to 199). Thus every instruction fetch is from page 0. The process is assigned 3 memory frames: frame 0 is already loaded with page 0 (i.e., process code), and the other two frames are initially empty.

a. Consider the following code fragment for initializing the array:

```
for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    A[i][j] = 0;
```

If the LRU (Least Recently Used) page replacement scheme is used then what is the total number of page faults in executing the previous initialization code?

The total number of page faults here would be 50. Since we have 50 pages (Array size / page size $= 10,000/200 = 50$ pages) and we're going in row-major order, we need to altenrate the page that we're on. With 100 elements, alternating pages, we'd get 50 page faults.

b. Now, instead, consider the following code fragment for initializing the array:

```
for (int j = 0; j < 100; j++)
  for (int i = 0; i < 100; i)
    A[i][j] = 0;
```

If the LRU page replacement scheme is used then what is the total number of page faults in executing the above initialization code?

In this one the number of page faults is going to go up to 5,000. Since the array is stored in row-major order, we'll get a fault for each alternating `A[i][j]` access, so the total number of faults will be $100 * 100/2 = 5,000$

Question 9

[10 marks] Which of the following programming techniques and data structures (in a user-level program) are good for a demand-paged environment, and which are bad? Explain your answer.

i. Sequential search through a linked list

This is going to be bad for a demand-paged environment because linked-lists are a non-contiguous data structure which means that the following element in the linked list may or may not be one a page that's currently loaded.

ii. Sequential search through an array

This is going to be good for a demand-paged environment because arrays are contiguous data structures, so unless the array is really really big and goes off a page, then you're going to sequentially access indexes no problem.

iii. Binary tree search

This is going to be bad for a demand-paged environment because like the linked list, trees are non-contiguous data structures, so you have no guarantees about where the next node is going to be. If it ends up being on a page that's not loaded, then that's going to give you a performance hit.

iv. Hashing with linear probing

This is going to be good for a demand-paged environment because hashmaps are implemented using arrays as an underlying data struture, which means the memory for the hashmap is going to be contiguous. Since linear probing is used instead of separate chaining, we don't need to consider what data structure is being used to avoid collisions. If our hash table gets really big and we need more than one page, then we'll get a hit to performance.

v. Queue implemented using a circular array

This is going to be good for a demand-paged environment because the underlying data structure is another array. Since arrays are contiguous data structures, the same thing as the sequential search and hashing with linear probing apply to the circular array.