Theory Assignment 3 Solutions

Solution 1.

(a) The scheduling algorithm is First-Come-First-Served (FCFS). When a process enters the ready queue its priority is set to 0, which is the lowest one in the queue. Since $\beta > \alpha$ so the priority of a running process is always more than that of a process in the ready queue. It means that a newly entered process in the ready queue always has to wait to get executed until all the previous processes finish their CPU bursts.

(b) The scheduling algorithm is Last-Come-First-Served (LCFS). When a process enters the ready queue its priority is set to 0, which is the highest among all the processes in the queue and the one executing. It means the new entered process always gets immediate CPU time since its priority is the maximum. This could result in starvation.

Solution 2.

(a) Yes, it can cause starvation. Consider a process P that used a very large fraction of the quantum (say 90%) the last time it ran. If a constant stream of new processes keep arriving then, based on the assumption that each new process is assumed to have used 50% of the quantum, the process P could starve as no bound can be put on when it can run next.

(b) The way to resolve starvation is by introducing the concept of "aging" in the algorithm. Here is a possible solution: the more time a process spends on the ready queue waiting to run, the more its "fraction of quantum used in last run" reduces. For instance, every τ time units of wait the fraction reduces by 10%. So, as a result, a process unable to run for a while will ultimately catch up with the other processes.

Solution 3.

In the following, it is assumed that a philosopher holding two chopsticks will eventually finish eating and put back the chopsticks. We also assume that there are minimum 2 chopsticks.

A request for a chopstick is granted as long as a *safe sequence* exists after the allocation, i.e., the resulting state is a *safe state*. To ensure that, follow these rules: (i) if at least two chopsticks are available then grant the request, otherwise (ii) If only one chopstick is available then grant the request only if either a) the requesting process already holds one chopstick, or b) at least another process holds two chopsticks. Otherwise the requesting process will have to wait even though the chopstick is available.

Solution 4.

Again, a request for a chopstick is granted as long as a safe sequence exists after the allocation, i.e., the resulting state is a safe state. We assume there are minimum 3 chopsticks. To ensure a safe state, follow these rules:

i) If at least 3 chopsticks are available then grant the request, otherwise (ii) if two chopsticks are available then grant the request only if either a) the requesting process already holds at least one chopstick, or b) at least another process holds two chopsticks, otherwise iii) if one chopstick is available then grant the request only if either a) the requesting processor already holds two chopsticks, or b) one of the processes holds three chopsticks.

Solution 5.

We will prove by contradiction. Let us assume that the system is in deadlock which involves *p* of the *n* processes, $p \geq 2$. Since, according to the question, a process can request only one resource at a time, so it must be that the deadlocked *p* processes are holding all the *m* resources (i.e., there are no available resources) because otherwise some of the *p* processes will not be in deadlock. So the sum of the maximum needs of the *p* processes is at least $p + m$. Of the remaining $n - p$ processes, sum of their maximum needs is at least $n - p$ because as the questions says each process needs between 1 and *m* resources. So, the sum of the maximum needs of all the n processes is at least $(p + m) + (n - p) = m + n$. This is a contradiction, because according to the question the sum of the maximum needs of the *n* processes must be less than $m + n$. Hence, the system cannot be in deadlock.

Solution 6.

(i) Given that the computer system's logical address space is 64 bits and a page size is 16 KB = 2^{14} bytes, the maximum number of entries in a page table = $\frac{2^{64}}{2^{14}} = 2^{50}$. Since each page table entry is 4 bytes long, hence the maximum size of a page table is $2^{50} \times 4$ bytes, which is 4 PB (peta bytes)

(ii) It is given that each page table entry is 4 bytes = 32 bits long. Noting that each page table entry contains a frame number, hence the total number of frames that can be addressed by 4 bytes is 2^{32}. Since each frame is 16KB in size, so the total main memory size (in user's address space) that can be addressed by this scheme is = $2^{32} \times$ 16KB = 64TB (Tera bytes)

(iii) Continuing from (i) above, the page table size is 2^{52} bytes, which is quite big to hold inside a page. So the page table is further divided into pages and information about these pages is kept inside an outer page table. The number of entries in the outer page table = $\frac{2^{52}}{2^{14}} = 2^{38}$ and size of the outer page table = $2^{38} \times 4$ bytes = 2^{40} bytes = 1 TB. This is diagrammatically shown below:

```
outer page       page    offset
←38 bits→  ←12 bits→  ←14 bits→
```

Noting that the outer page table itself is quite big to hold inside a page, it is further divided into pages and the whole process is repeated as follows:

```
  5th level    4th level    3rd level    2nd level    1st level     offset
←2 bits→←12 bits→←12 bits→←12 bits→←12 bits→←14 bits→
```

As the diagram shows, the total levels of hierarchies required = 5

(iv) Now, considering that inverted paging scheme is used and the user-addressable main memory size is 64TB (from (ii) above) which is composed of 2^{32} frames, so the total number of entries in the inverted page table = total number of frames = 2^{32}.

Since each entry of the inverted page table is 4 bytes long, so the maximum size of the inverted page table = $2^{32} \times 4$ bytes = 16GB.

Solution 7.

Effective memory access time = page fault rate $\times$ page fault service time +
Page hit rate $\times$ effective memory access time in page hit (1)

In the above, page fault rate = 0.02
Page fault service time = 20 milliseconds = 20000 microseconds
Page hit rate = 0.98
Effective memory access time in page hit = TLB hit rate $\times$ memory access time in TLB hit + TLB miss rate $\times$ memory access time in TLB miss
= 0.8 $\times$ (TLB access time + memory access time) +
0.2 $\times$ (TLB access time + page table access time + Memory access time)
= 0.8 $\times$ (0.2 + 1) + 0.2 $\times$ (0.2 + 1 + 1) microseconds
= 1.4 microseconds
Substituting in equation (1) above:
Effective memory access time = 0.02 $\times$ 20000 + 0.98 $\times$ 1.4 microseconds
= 401.372 microseconds

Solution 8.

a) Since the array is stored in row-major order, the first two rows reside on page 1, the next two rows reside on page 2, and so on. Since there are altogether 100 rows, so 50 pages are required to store the array.

When the array is initialized row-by-row, the following is going to be the page reference string in initializing the array:
1 1 1 …(200 times) 2 2 2 …(200 times) ……….50 50 50 (200 times)
Reference to every new page will result in a page fault, and hence total number of page faults = 50.

b) When the array is initialized column-by-column, every alternate access is going to reside on a new page and will result in a page fault. The following is the page reference string in traversing one column:

1 1 2 2 3 3…..50 50
and there are 50 page faults in the above. There are 100 columns altogether and hence the total number of page faults = 50 $\times$ 100 = 5000.

Solution 9.

(i) Not good. The nodes of a linked list can be located non-contiguously in the process' address space. It is highly possible that two adjacent nodes reside in different pages. In demand-paged environment, pages always are swapped into memory on demand. So, accessing subsequent nodes can lead to frequent page faults, if the nodes reside on different pages.

(ii) Good. An array is a contiguous memory block. The adjacent entries of the array are more likely to reside on the same page and hence will lead to fewer page faults.

(iii). Not good. Suppose a binary tree is implemented by using linked nodes, then searching through it will lead to the same situation as in part (i) above. If the tree is implemented using an array, then searching through it will require accessing non-contiguous entries of the array. If the array is large and scattered over multiple pages, then accessing non-contiguous entries of the array on a binary tree search could lead to more page faults.

(iv). Good. Hashing is based on an array based implementation and accessing this array is sequential in the case of linear probing.

(v). Good. Successive enqueue or dequeue operations will lead to contiguous accesses to the array (except at the two boundaries of the array) and hence will lead to fewer page faults.