CMSC 412 Midterm #2 (Spring 1996)

1.) (16 points) Define (or explain) the following terms:

a) Belady's anomaly

The property of some page replacement algorithms that, for some reference strings, adding additional page frames increases the number of page faults.

b) shortest job first scheduling

A scheduling policy that uses knowledge about how long jobs will take to select the shortest job to run first. It requires knowledge of the future, and so is not possible to implement exactly.

c) cylinder (as it applies to disks)

The collection of all tracks on all platters equal distant from the center of the disk drive.

d) file pointer vs. direct access I/O

file pointers are an implicit way to specify the next location to be read or written. Direct I/O provides explicit offsets with each request.

2.) (9 points) In project #3, the update to semaphore data structures has to be done atomically, yet the implementation of P and V in the `system_service` routine can run with interrupts enabled. Why is this possible?

P and V are atomic since one invocation of system_service can not be interrupted by another. Timer interrupts can go off, but they only set a flag if system_service is active.

3.) (25 points) Consider an OS running on the following machine:

a two level page table (the pointer to the page directory is stored in a register)
no cache or TLB
a memory reference takes 100 nanoseconds

a) If 0.001% of attempts to access physical memory cause a page fault that takes 10msec to service, what is the effective access time of user space (paged) memory. (You answer need only be accurate to 3 significant digits).

Assume that page directories and tables are pinned down (always) in memory. We need to access the page dir, the page table, and then the data reference. The reference to data could cause a page fault. EAT = 100ns + 100ns + 10^{-5}(10ms + 100ns) + 100ns ≈ 400ns.

b) If we add a TLB and 75% of all page-table references are found there, what is the effective access time of user space memory.

Adding a page table implies that 75% of the time we can skip the translation memory accesses. Also if the item is in the TLB, we know it is in memory (not paged out).
So EAT = .75(100ns) + .24999 (3 * 100ns) + 10^{-5} (10ms + 3 * 100 ns) ≈ 250ns.

c) Someone suggests that doubling the page size might improve the system performance. What factors would you need to consider to evaluate if this change would help or not?

- internal fragmentation goes up
+ fewer page faults if spatial locality is large enough
- longer time to service a single fault since there is more data to move
+ faster to process one large page fault than two smaller ones
(due to fault handler time & maybe seek time)
- more data to write back for dirty pages

4.) (15 points) Explain how virtual memory schemes serve to protect memory from unauthorized access.

> Per process page tables provide only way to access memory. If a frame lacks a PTE, there is no way for a process to access it
>
> Only the OS can write to or update the page tables, so processes can't map frames that are not theirs.
>
> Attempts to access PTE items that are not valid result in a page fault trap into the OS.

5.) (15 Points) Consider the problem of making water. To make water, two hydrogen atoms and one oxygen atom need to be combined. Assume that atoms are processes, and that exactly two hydrogen and one oxygen process must call the procedure `makeWater` at once to create water. Use semaphores to provide a starvation and busy wait free solution to this problem. Make sure you indicate the initial values of any semaphores you use. You may assume the semaphores are "strong" (i.e. processes blocked on a P are released in FIFO order).

```
Oxygen process
while (1) {
   P(oxygen);
   P(mutex);
   waiting++;
   if (waiting == 3) then
      V(go); V(go); V(go);
   V(mutex);
   P(go);
   // makeWater();
   V(oxygen);
}
```

```
Hydrogen process
while (1) {
   P(hydrogen);
   P(mutex);
   waiting++;
   if (waiting == 3) then
      V(go); V(go); V(go);
   V(mutex);
   P(go);
   // makeWater();
   V(hydrogen);
}
```

```
Initial Values
semaphore mutex = 1
semaphore hydrogen = 2
semaphore oxygen = 1
semaphore go = 0
variable int waiting = 0
```

6.) (20 points) Disks and filesystems

a) A well known computer scientist is fond of saying 'to get good performance out of disks you have to realize they are really sequential not random access devices'. What does he mean by this statement?

> Disks have heads that need to seek to reach the desired blocks. Thus the time to access random locations is not uniform and independent of the previous accesses.

b) If you knew that most accesses to the filesystem were either backward sequential reads, or append writes to very large files (several gigabytes each) which filesystem implementation would you prefer linked allocation (MS-DOS), or indexed allocation (UNIX)? Explain your answer.

> Indexed allocation is better since it is possible to reach any location in a file with only a constant number of pointer accesses (through the direct, indirect, double indirect, or triple indirect blocks). With linked allocation we need to walk the forward though the links to find a specific location in a file. This applies to reverse sequential reads. For appends, most linked schemes maintain a pointer to the last block to make append faster.

c) You want to extend your operating system to permit very large files (files bigger than will fit on one disk). Explain how you would need to change the filesystem data structures. How could this extension impact mean time to data loss (failure) of the aggregate filesystem compared to a single disk filesystem?

> Need to change all data structures that have a block field to have a block plus disk field. This would reduce the mean time to data loss since the probability of failure for n dsks is $(1-p)^n$ where p is the probability that one disk has not failed.