# Midterm *Solutions*

CS 414 Operating Systems, Spring 2007
March 8th, 2007
Prof. Hakim Weatherspoon

Name: ___________________________________NetId/Email:____________________________

**Read all of the following information before starting the exam:**
Write down your name and NetId/email NOW.

This is a **closed book and notes** examination. You have 90 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points given to the question; there are 100 points in all. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. ***Make your answers as concise as possible.*** If a question is unclear, please simply answer the question and state your assumptions clearly. If you believe a question is open to interpretation, then please ask us about it!

**Good Luck!!**

| Problem | Possible | Score |
|---|---|---|
| **1** | 26 | |
| **2** | 24 | |
| **3** | 16 | |
| **4** | 12 | |
| **5** | 22 | |
| **Total** | **100** | |

1. (26 points total) Short answer questions (***NO answer should be longer than two or three sentences***).

a. (2 points) Name three ways in which the processor can transition form user mode to kernel mode? Can the user execute arbitrary code after transition?

*1) The user process can execute a trap instruction (e.g. system call). A trap is known as a synchronous software interrupt.*
*2) The user process can cause an exception (divide by zero, access bad address, bad instruction, page fault, etc).*
*3) The processor can transition into kernel mode when receiving an interrupt.*

*No, the user cannot execute arbitrary code because entry to kernel mode is through a restricted set of routines that ensure only the kernel is running, not the user process.*

b. (6 points) Threads

i) (2 points) What needs to be saved and restored on a context switch between two threads in the same process? What if two are in different processes? Be brief and explicit.

*Need to save the registers, stack pointer, and program counter into the thread control block (TCB) of the thread that is no longer running. Need to reload the registers, stack pointer, and program counter from the TCB of the new thread.*

*When the threads are from different processes, need to not only save and restore what was given above, but also need to load the pointer for the top-level page table of the new address space. (Note that this top-level page table pointer from the old process does not need to be saved since it does not change and is already contained in the process control block (PCB)).*

ii) (2 points) Why is switching threads less costly than switching processes?

*Less state needs to be saved and restored. Furthermore, switching between threads benefits from caching; whereas, switching between processes invalidates the cache and TLB.*

iii) (2 points) Suppose a thread is running in a critical section of code, meaning that it has acquired all the locks through proper arbitration. Can it get context switched? Why or why not?

*Yes, a process holding a lock can get context switched. Locks (especially user-level locks) are independent of the scheduler. (Note that threads running in the kernel with interrupts disabled would not get context-switched).*

c. (6 points) Deadlock

i) (2 points) Name the four conditions required for deadlock and give a brief (one sentence) description of each.

*Mutual exclusion: a resource can be possessed by only one thread.*
*Hold and wait: A thread can hold a resource such as a lock while waiting for another.*
*No preemption: The resource cannot be taken away from the thread.*
*Circular wait: Two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain holds*

ii) (2 points) Does a cyclic dependency always lead to deadlock? Why or why not?

*No. If multiple equivalent resources exist, then a cycle could exist that is not a deadlock. The reason is that some thread that is not part of the cycle could release a resource needed by a thread in the cycle, thereby breaking the cycle.*

iii) (2 points) What is the difference between deadlock prevention and deadlock avoidance? What category does Bankers algorithm falls in and why?

*Deadlock prevention prevents deadlock by preventing one of the four conditions required for deadlock to occur.*

*Deadlock avoidance ensures the system is always in a safe state by not granting requests that may move the system to an unsafe state. A is consisered safe if it is possible for all processes to finish executing (i.e. a sequence exists such that each process can be given all its required resources, run to completion, and return resources allocated, thus allowing another process to do the same, etc until all process complete). Deadlock avoidance requires the system to keep track of the resources such that it knows the allocated, available, and remaining resource needs.*

*The Bankers algorithm is a deadlock avoidance scheme since it defines an algorithm and structure to ensure the system remains in a safe state before granting any new resource requests.*

d. (2 points) What are exceptions? Name two different types of exceptions and give an example of each type.

*Exceptions are events that stop normal execution, switch the execution mode into kernel mode, and begin execution at special locations within the kernel. Examples include system calls, divide by zero errors, illegal instructions, and page faults.*

e. (2 points) Why would two processes want to use shared memory for communication instead of using message passing?

*Performance. Communicating via shared memory is often faster and more efficient since there is no kernel intervention and no copying.*

f. (2 points) What is internal fragmentation? External fragmentation? Give a brief example of each.

*Internal fragmentation: allocated space that is unused. For example, a process may have allocated a 1KB page, but uses only 10 bytes.*

*External fragmentation: unallocated "free" space that lies between allocated space such that contiguous regions of free space is insufficient to satisfy subsequent space allocation requests even though sufficient free space exists in aggregate.*

g. (6 points) For each of the following thread state transitions, say whether the transition is legal *and* how the transition occurs or why it cannot. Assume Mesa-style monitors.

i) (2 points) Change from thread state BLOCKED to thread state RUNNING

*Illegal. The scheduler selects threads to run from the list of ready (or runnable) threads. A blocked thread must first be placed in the ready queue before it can be selected to run.*

ii) (2 points) Change from thread state RUNNING to thread state BLOCKED

*Legal. A running thread can become blocked when it requests a resource that is not immediately available (disk I/O, lock, etc).*

iii) (2 points) Change from thread state RUNNABLE to thread state BLOCKED

*Illegal. A thread can only transition to BLOCKED from RUNNING. It cannot execute any statements when still in a queue (i.e. the ready queue).*

***EXTRA CREDIT***

h. (3 points) Suppose you have a concurrent system with locks: Lock.acquire() blocks until the Lock is available and then acquires it. Lock.release() releases the Lock. There is also a Lock.isFree(), that does *not* block and returns true if the Lock is available; otherwise, returns false .

What can you conclude about a subsequent Lock.acquire(), based on the result of a previous call to Lock.isFree() ?

*Nothing. A separate thread could acquire the Lock in between the two calls. This is why we don't implement Lock.isFree().*

2. (24 points total) CPU Scheduling. Here is a table of processes and their associated arrival and running times.

| **Process ID** | **Arrival Time** | **Expected CPU Running Time** |
|---|---|---|
| Process 1 | 0 | 5 |
| Process 2 | 1 | 5 |
| Process 3 | 5 | 3 |
| Process 4 | 6 | 2 |

a. (12 points) Show the scheduling order for these processes under First-In-First-Out (FIFO), Shortest-Job First (SJF), and Round-Robin (RR) with a quantum = 1 time unit. *Assume that the context switch overhead is 0 and new processes are added to the* ***head*** *of the queue except for FIFO.*

| **Time** | **FIFO** | **SJF** | **RR** |
|---|---|---|---|
| **0** | *1* | *1* | *1* |
| **1** | *1* | *1* | *2* |
| **2** | *1* | *1* | *1* |
| **3** | *1* | *1* | *2* |
| **4** | *1* | *1* | *1* |
| **5** | *2* | *3* | *3* |
| **6** | *2* | *3* | *4* |
| **7** | *2* | *3* | *2* |
| **8** | *2* | *4* | *1* |
| **9** | *2* | *4* | *3* |
| **10** | *3* | *2* | *4* |
| **11** | *3* | *2* | *2* |
| **12** | *3* | *2* | *1* |
| **13** | *4* | *2* | *3* |
| **14** | *4* | *2* | *2* |
| **15** | | | |

b. (12 points) For each process in each schedule above, indicate the queue wait time and turnaround time (TRT).

| Scheduler | Process 1 | Process 2 | Process 3 | Process 4 |
|---|---|---|---|---|
| FIFO queue wait | *0* | *4* | *5* | *7* |
| FIFO TRT | *5* | *9* | *8* | *9* |
| SJF queue wait | *0* | *9* | *0* | *2* |
| SJF TRT | *5* | *14* | *3* | *4* |
| RR queue wait | *8* | *9* | *6* | *3* |
| RR TRT | *13* | *14* | *9* | *5* |

The queue wait time is the *total* time a process spends in the wait queue.
The turnaround time is defined as the time a process takes to complete after it first arrives.

*We took points off for the following reasons:*
*-1 one or two process wrong*
*-1 if incorrect calculation due to swapping preempting process 3 with process 4 at time 6 Process 4 would not preempt process 3 because they have the same priority (2 time slots remaining); instead process 4 goes to the head of the queue then when runs when process 3 completes*
*-2 for each measurement combo incorrect*
*-12 more than four process wrong, appears does not understand concepts*

3. (16 points) Virtual Memory Page Replacement
Given the following stream of page references by an application, calculate the number of page faults the application would incur with the following page replacement algorithms. Assume that all pages are initially free.

Reference Stream: A B C D A B E A B C D E B A B

a. (4 points) FIFO page replacement with 3 physical pages available.

*11 page faults*

| *Refrenence stream:* | *A* | *B* | *C* | *D* | *A* | *B* | *E* | *A* | *B* | *C* | *D* | *E* | *B* | *A* | *B* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *oldest page* | *A* | *A* | *A* | *B* | *C* | *D* | *A* | *A* | *A* | *B* | *E* | *E* | *C* | *D* | *D* |
| | | *B* | *B* | *C* | *D* | *A* | *B* | *B* | *B* | *E* | *C* | *C* | *D* | *B* | *B* |
| *Newest page* | | | *C* | *D* | *A* | *B* | *E* | *E* | *E* | *C* | *D* | *D* | *B* | *A* | *A* |
| *page fault* | √ | √ | √ | √ | √ | √ | √ | | | √ | √ | | √ | √ | |

b. (4 points) LRU page replacement with 3 physical pages available.

*12 page faults*

| *Refrenence stream:* | *A* | *B* | *C* | *D* | *A* | *B* | *E* | *A* | *B* | *C* | *D* | *E* | *B* | *A* | *B* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *least recently used* | *A* | *A* | *A* | *B* | *C* | *D* | *A* | *B* | *E* | *A* | *B* | *C* | *D* | *E* | *E* |
| | | *B* | *B* | *C* | *D* | *A* | *B* | *E* | *A* | *B* | *C* | *D* | *E* | *B* | *A* |
| *most recently page* | | | *C* | *D* | *A* | *B* | *E* | *A* | *B* | *C* | *D* | *E* | *B* | *A* | *B* |
| *page fault* | √ | √ | √ | √ | √ | √ | √ | | | √ | √ | √ | √ | √ | |

c. (4 points) OPT page replacement with 3 physical pages available.

*8 page faults*

*On a page fault, replace the page used furthest in the future.*

| *Refrenence stream:* | *A* | *B* | *C* | *D* | *A* | *B* | *E* | *A* | *B* | *C* | *D* | *E* | *B* | *A* | *B* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *A* | *B* | *C* | *D* | *D* | *D* | *E* | *E* | *E* | *E* | *E* | *E* | *E* | *E* | *E* |
| | | *A* | *B* | *B* | *B* | *B* | *B* | *B* | *B* | *B* | *B* | *B* | *B* | *B* | *B* |
| | | | *A* | *A* | *A* | *A* | *A* | *A* | *A* | *C* | *D* | *D* | *D* | *A* | *A* |
| *page fault* | √ | √ | √ | √ | | | √ | | | √ | √ | | | √ | |

d. (4 points) True or False. If we increase the number of physical pages from 3 to 4, the number of page faults always decreases using FIFO page replacement. Briefly explain.

*False due to Belady's anomaly. For instance, for this problem if we increase the number of physical pages available from 3 to 4, then the number of page faults increases from 11 to 12 using a FIFO page replacement algorithm.*

*We took points off for problems* a, b*, and* c *for the following reasons:*
*-1 perfect stream but counted wrong (count close)*
*-1 correct number of page faults, but not completely correct sequence*
*-2 incorrect number of page faults, but correct process and close (off by 1)*
*-2 perfect stream, but counted wrong (count way off)*
*-3 incorrect number of page faults, but reasonable (process not clear)*
*-3 only off by 1, but no reasoning*
*-4 did not answer or way off*

*We took points off for problem* d *for the following reasons:*
*-2 correct answer, but incorrect reasoning (or no reasoning)*
*-2 correct answer, but says "never increases"*
*-4 incorrect answer*

4. (12 points) Memory Management

a. (6 points) Consider a memory system with a cache access time of 10ns and a memory access time of 200ns. If the *effective access time* is 10% greater than the cache access time, what is the hit ratio H? (Fractional answers are okay).

*Effect Access Time:* $T_e = H \times T_c + (1 - H)(T_m + T_c)$,
*where* $T_c = 10ns$, $T_e = 1/1 \times T_c$, *and* $T_m = 200ns$.

*Thus,* $(1.1) \times (10) = H \times 10 + (1 - H)(200 + 10)$
$11 = 10H + 210 - 210H$

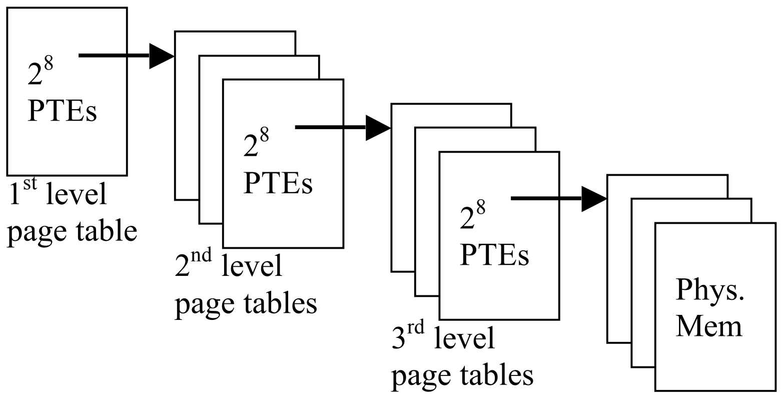
$H = 199/200$

*We took points off for the following:*
*-1 memory access time not including cache access time*
*-1 wrong effective access time*
*-1 wrong cache access time*
*-1 wrong miss calculation*
*-3 wrong equation*
*-2 memory access time not including cache miss time, plus arithmetic error*
*-5 only calculate effective access time*

b. (6 points) Assuming a page size of 1 KB and that each page table entry (PTE) takes 4 bytes, how many levels of page tables would be required to map a *34-bit* address if every page table fits into a single page. Be explicit in your explanation.

*Since a page is* $2^{10}$ *bytes (1 KB) and each PTE is* $2^2$ *bytes (4 bytes), a 1-page page table contains 256 or* $2^8$ *PTEs* ($2^{10}/2^2=2^8$). *Each entry points to a page that is* $2^{10}$ *bytes (1 KB). With one level of page tables we can address a total of* $2^8 \times 2^{10} = 2^{18}$ *bytes). Adding another level yields another* $2^8$ *pages of page tables, addressing a total of* $2^8 \times 2^8 \times 2^{10} = 2^{26}$ *bytes. Finally, adding a third level* $2^8$ *pages of page tables, addressing a total of* $2^8 \times 2^8 \times 2^8 \times 2^{10} = 2^{34}$ *bytes. So, we need 3 levels.*

| | | | |
|---|---|---|---|
| 8 bits | 8 bits | 8 bits | 10 bits offset |



*We took points off for the following reasons:*
*-1 wrong calculation of bits from 256 -2 page index only 2 bits*
*-2 right answer, wrong reasom ( 2 pages too full, 1 too empty)*
*-2 wrong calculation of PTE per page*
*-2 KB to Kb for not reason*
*-2 forgot to take page index bits into consideration*
*-3 divide number of pages by PTE on a page*
*-4 only describe 10 bit index*
*-5 2 bits per page index, no real idea*
*-6 very large number of strange multiplications*
*-6 arbitrarily deciding that all 3 level page tables can hold 32 bits*

5. (22 points) Concurrency: the "$H_2O$" problem
You have just been hired by Mother Nature to help her out with the chemical reaction to form water, which she does not seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure *hReady* when it is ready to react, and each O atom invokes a procedure *oReady* when it is ready. For this problem, you are to write the code for *hReady* and *oReady*. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the threads must call the procedure *makeWater* (which just prints out a debug message that water was made). After the *makeWater* call, two instances of *hReady* and one instance of *oReady* should return. Your solution should avoid starvation and busy-waiting.

You may assume that the semaphore implementation enforces FIFO order for wakeups—the thread waiting longest in P() is always the next thread woken up by a call to V().

a. (4 points) Specify the correctness constraints. Be succinct and explicit.

- *No call to* makeWater *should be made when fewer than 2 H atoms have called hReady and no O atom has called oReady.*
- *Exactly two H atoms return and one O atom returns for every call to* makeWater.

*We took points off for the following:*
*-2 presented half of the correctness constraints above.*
*-3 just presented a list of conditions for critical section(i.e. mutual exclusion, progress, bounded waiting).*

b. (18 points) provide the pseudo implementation of *hReady* and *oReady* using semaphores.

```
Semaphore mutex = 1;
Semaphore h_wait = 0;
Semaphore o_wait = 0;
int count = 0;

hReady() {                          oReady()
    P(mutex);                       {
    count++;                           P(o_wait);
    if(count %2 == 1) {                V(h_wait);
        V(mutex);                      V(h_wait);
        P(h_wait);                     makeWater();
    } else {
        V(o_wait);                     return;
        P(h_wait);                  }
        V(mutex);
    }

    return;
}
```

---

Alternative equivalent solution:

```
Semaphore mutex = 1;
Semaphore h_wait = 0;
Semaphore o_wait = 0;

hReady() {                      oReady()
    V(o_wait)                   {
    P(h_wait)                      P(mutex)
                                   P(o_wait);
    return;                        P(o_wait);
}                                  V(h_wait);
                                   V(h_wait);
                                   makeWater();
                                   V(mutex)

                                   return;
                                }
```

*We deducted points for the following mistakes:*
*-1 inconsistent testing, incrementing, or decrementing of a counter variable.*
*-3 intricate setup that can cause a busy wait.*
*-4 atoms return before call to makeWater.*
*-4 starvation/deadlock due to sensitivity of arrival order of atoms.*
*-4 no initial values for semaphores.*
*-6 does not call makeWater with exactly two H atoms and one O atom*
*-6 single violation unprotected modification/testing of a shared variable outside critical section.*
*-8 solution deadlocks*
*-10 immediate deadlock due to incorrect initialization of semaphores.*
*-10 multiple violations of accessing shared variable outside critical section. Also, busy waiting.*

*No credit or almost no credit if did not present a viable solution.*