Midterm Exam
CS 111, Principles of Operating Systems
Fall 2016

This is a closed book, closed note test. Answer all questions.

Each question should be answered in 1-3 paragraphs. DO NOT simply write everything you remember about the topic of the question. Answer what was asked. Extraneous information not related to the answer to the question will not improve your grade.

General guidelines on grading: Each question is worth 10 points, but there should be partial credit possibilities for all of them. Give only integral numbers of points, not fractional points. Except where noted in the answers, precisely matching the wording of the provided answer is not necessary. The major points need to be hit, as indicated. Students should not lose point for poor writing, grammar, or penmanship, except when these issues make it impossible to determine what they meant.

1. What is the difference between the invalid bit in a page table entry and the invalid bit in a translation lookaside buffer entry? What happens in each case if an address translation attempts to use that entry? Is it possible for both to be set? Why?

The invalid bit in the page table entry is set if the owning process does not have a page at that address allocated. Thus, an attempt to translate that address will lead to an exception. The invalid bit in a translation lookaside buffer entry is set if that TLB entry is not currently valid. Translating the address in question may still go ahead, but the actual page table, rather than the translation lookaside buffer entry, must be consulted. Both could be set. The TLB entry invalid bit might be set because the TLB entry refers to a different process' address space, while the page table entry invalid bit might be set because the process has not allocated that page.

Grading guidelines:

- *2 points each for correctly describing the purpose of each invalid bit*
- *2 points each for correctly describing the actions taken when an address translation encounters the invalid bit*
- *2 points for proper discussion of both being set*

Reference: Arpaci-Dusseau chapter 19, page 8 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf)

2. There are many difficult issues that arise due to uncontrolled concurrent executions. Why do we not simply turn off interrupts to prevent such problems from arising? Why not always? Why not just for all critical sections of code?

First, on multicore machines and other machines with genuine parallel operations (such as symmetric multiprocessors), concurrent execution can occur even when interrupts are turned off. Also, many system activities require interrupts to proceed properly, such as

handling I/O events and performing preemptive scheduling. If interrupts are always off, we cannot run a preemptive scheduler and we cannot guarantee that I/O events like keystrokes, mouse movements, message arrivals, or disk reads will be handled in a timely manner. Some interrupts may be lost while interrupts are disabled. As a result, the system may run very slowly and provide poor responsiveness if interrupts are kept off for a long time. Thus, running with interrupts turned off always is not desirable.

Turning interrupts off for critical section is less harmful, but unless the critical sections are guaranteed to be short, the same undesirable effects can occur. Again, merely disabling interrupts won't prevent all concurrent operations on multicore machines. Finally, turning interrupts off and on is a privileged operation and cannot be performed by user code, unless you provide a system call for that purpose.

Grading guidelines:

- *7 points for proper discussion of downsides of turning off interrupts. Loss of 2 point each for not discussing multiprocessor issue, preemptive scheduling, and handling I/O events. Remaining point for general discussion of consequences.*
- *3 points for discussion of turning off during critical sections. Either here or earlier, should mention not being able to disable interrupts from user code, with loss of 1 point if not mentioned at all.*

Reference: Arpaci-Dusseau chapter 28, page 4 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf), Lecture 7 pages 47-51 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_7.pdf)

3 What issues arise in management of thread stacks that do not arise in management of process stacks? Why do these issues arise and what is typically done to handle them? What are the disadvantages of this approach and why is the approach used despite those disadvantages?

A process has its own virtual memory space and only one stack. This stack can be placed at one end of the essentially one-dimensional memory and allowed to grow towards the other end, while the data segment can be located at that other end and grow in the opposite direction. Unless all memory space is used up, they will not run into each other. But threads of a single process share the same address space. Since the address space is still one dimensional, multiple thread stacks cannot all grow towards a single "hole" in the middle of the address space. Therefore, thread stacks might run into each other or other memory segments if they grow too large. The typical solution is to place thread stacks at one place in the virtual address space and limit their maximum size. This solution implies that a thread cannot make deeply nested function calls, as each will use up part of its stack space, and too many will overflow its allocated space. The approach is used largely because there are few alternatives and most threads tend to be relatively simple, with little likelihood of making deep sequences of function calls. So they don't need arbitrarily growing stack sizes.

Grading guidelines:

- *5 points for identifying and discussing the basic issue of how to place thread stacks in a virtual address space.*
- *2 points for properly describing the usual approach used to handle this problem*
- *2 points for the disadvantage of limiting how deep thread calls can nest*
- *1 point for good arguments for why this is used anyway*

References: Arpaci-Dusseau chapter 26 pages 1-2 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf), Lecture 7 slides 9-11 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_7.pdf)

4 In MLFQ scheduling, processes are moved from one scheduler queue to another based on their behavior. Each such queue has a particular length of time slice for all processes in that queue. Why might a process be moved from a queue with a short time slice to a long time slice? How can the operating system tell that it should be moved?

MLFQ scheduling tries to find a good time slice for each process, where "good" is defined as a time slice that is suited for the process' needs. Processes that perform a lot of I/O need short time slices, since they will probably block on I/O after running for a short time. They will need to be scheduled fairly often, though, so their I/O is handled promptly once it is ready. Processes that perform a lot of computation and little I/O would work best with long time slices, since that will allow them to get much work done before they are context switched out due to time slice expiration. But if we allow them to run for such long slices very often, they will hog the processor. So they get scheduled for longer slices, but less often. A process would move from the short time slice queue to the long time slice queue if it appeared to require more processing time than it got on the short time slice queue. That would be detected by the process frequently running to the end of its short time slice and rarely blocking on I/O requests. The OS can simply count how often the process finishes executing for each reason and move the process to the longer time slice queue if it usually uses up its entire short time slices.

Grading guidelines:

- *5 points for correct description of why the move should be performed*
- *5 points for correct description of how the OS knows that the move is needed*

References: Arpaci-Dusseau chapter 8 pages 2-4 (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf), lecture 4 slides 50-52 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_4.pdf)

5 Will binary buddy allocation suffer from internal fragmentation, external fragmentation, both, or neither? If it does suffer from a form of fragmentation, how

badly and why? If it does not suffer from a form of fragmentation, why not?fragmentation, external fragmentation, or both? How badly? Why?

Binary buddy allocation starts with all of free memory in one segment. On request of n bytes, it looks for a free segment of at least n bytes, but no more than 2n. If it has no such segment, it halves the next largest segment and continues halving one of the segments until it has produced a segment in the desired range. That segment is given to the requestor. Since the possible segment sizes are fixed, many allocations will be met with somewhat larger segments, resulting in internal fragmentation. The internal fragmentation will always be less than 50%, since if a segment more than twice the size of the request is available, it will be split in half before granting the request.

External fragmentation is also possible, since segments keep getting split into two smaller segments. As a result, we could end up with lots of very small segments scattered through memory, totaling more than N bytes, but with none of them large enough to fulfill a request for N bytes. That would be external fragmentation. It might not be too bad, since one only splits a segment when one is asked for a segment of a particular size. Thus, while one could have small segments, they are likely to be usable, since segments of those sizes were usable before.

Grading guidelines:

- *5 points for internal fragmentation discussion. 3 points for correct analysis of why internal fragmentation will occur. 2 points for good description of how much internal fragmentation there will be.*
- *5 points for external fragmentation discussion. 3 points for correct analysis of why external fragmentation will occur. 2 points for good description of how much external fragmentation there will be.*

References: Arpaci-Dusseau chapter 17 pages 14-15 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf)

6 What is the purpose of a trap table in an operating system? What does it contain? When is it consulted? When is it loaded?

The trap table is used to specify what actions the operating system should take when a particular exception occurs. It will contain pointers to exception handling routines, indexed by a number specifying the particular exception that occurs. It is consulted whenever an exception actually occurs. It is loaded at boot time and typically will not change until the next reboot.

Grading guidelines:

- *4 points for correct description of purpose*
- *2 points for correct description of contents*
- *2 points for correct description of when it is consulted*
- *2 points for correct description of when it is loaded. Suggesting it can be changed after boot time is OK provided there is some*

reasonable discussion of why it would be, such as mentioning hot plugging of a new device and altering the trap table to deal with its device driver

References:Arpaci-Dusseau chapter 6 pages 4-5 (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf), lecture 3 slides 45-47 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_3.pdf)

7 Assuming a correct implementation, do spin locks provide correct mutual exclusion? Are they fair? Do they have good performance characteristics? Explain why for all three of these evaluation criteria (correctness, fairness, performance).

Correctly implemented spin locks do provide correct mutual exclusion. The holder of the lock is permitted to access the critical section and no one else is. When the lock holder releases the lock, if any other process is spinning on that lock, one of the spinning processes will obtain the lock. So not only will we prevent multiple processes from simultaneously accessing the critical section, but we will ensure that if anyone wants to access the critical section, someone will be able to.

Spin locks are not fair. They give no guarantee that a process spinning on a lock will ever obtain the lock. If multiple processes are all spinning on the lock, only one will be the next to get it. That isn't necessarily the one that has waited longest, and there is no way to ensure that a process' waiting time while spinning is bounded.

Spin lock performance is poor. Spinning burns cycles, so instructions are being spent merely to determine if the lock can be obtained. Further, if the process holding the lock is preempted by someone spinning on the lock, no progress will be made until the process holding the lock is re-scheduled. This is worst on single core systems. On multi-core systems, it's not quite as bad, since the lock-holding process is less likely to be preempted by spinning processes. If the critical section is short, the spinning process might not need to wait long and preemptions might not occur.

Grading guidelines: For all parts, yes or no answers are not sufficient. There should be an explanation of why. Just answering yes or no (correctly) will only get 1 point for that part.

- *4 points for correctness discussion*
- *3 points for fairness discussion*
- *3 points for performance discussion. 1 of those points for discussing the single core vs. multicore issue*

References : Arpaci-Dusseau chapter 28 page 9 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf), Lecture 8 slide 23 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_8.pdf)

8 What is the purpose of using a clock algorithm to handle page replacement in a virtual memory system? How does it solve the problem it is intended to address?

A clock algorithm is intended to approximate an LRU page replacement algorithm without incurring the costs of true LRU. In true LRU, every page access requires saving a clock value of the time of that access, and choosing the least recently used page requires checking the access time values for all pages. (Alternately, one can keep an ordered list of pages by access time, but that requires updating this list on each page access, which is even more expensive than searching the access time values on pages.)

A clock algorithm avoids keeping the timestamp by instead having a single bit that is set when a page is accessed. The bit is set in hardware on each access, so there is little extra cost for this level of record keeping. The clock algorithm avoids the cost of checking all pages to choose the least recently used by simply choosing the first page it checks where this bit is not set.

Grading guidelines:

- *5 points for proper discussion of the purpose of the algorithm. No more than 2 points if no mention is made of its relationship to LRU. The answer need not explicitly say it is an LRU approximation, but must clearly discuss that relationship. It is not necessary to include the parenthetical thought in the answer above to get full points. Merely identifying reasons why LRU would be too expensive is enough.*
- *5 points for how the clock algorithm deals with the problem. This portion of the answer needs to touch on both the benefits of only using a single bit per page (3 points) and the benefits of not searching through all pages to find a replacement (2 points).*

References:Arpaci-Dusseau Chapter 22 pages 12-13, lecture 6 slides 41-43 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_6.pdf)

9 What are two fundamental problems faced by a user level thread implementation that are not faced by a kernel level thread implementation? Why do these problems arise in user level implementations? Why don't they arise in kernel level implementations?

User level threads generally cannot take advantage of multicore or other multiprocessor systems. The OS schedules the process containing the user level threads on one processor, not knowing that the process contains multiple potentially parallelizable threads. In kernel level thread implementations, the OS is aware of the existence of multiple threads and can choose to schedule them on more than one processor simultenously.

User level thread implementations block all threads in the process if any of the threads block, even though the other threads might be able to run properly. From the OS's perspective, the blocking request applies to the entire process, since it does not know that the process contains multiple threads of control. In kernel level thread implementations, the blocking request was issued by a particular thread known to the kernel, so it can block just that thread and allow other threads in the same process to continue execution.

Grading guidelines: Other answers may be possible, such as user level threads requiring a user level scheduler. Those problems are less fundamental, though. Depending on how well they are argued, they can be given up to three points. Identifying more than two issues is OK. If they include both of those mentioned in the sample answer and satisfy the other grading criteria, give such answers full points.

- *5 points for discussion of the multicore issue. Must describe both why it is a problem in the user level thread implementation and why it is not a problem for kernel threads.*
- *5 points for the discussion of the blocking issue. Must describe both why it is a problem in the user level thread implementation and why it is not a problem for kernel threads.*

References: Web page on user mode threads (http://lasr.cs.ucla.edu/classes/111_fall16/readings/user_threads.html)

10 What is the difference for a virtual memory system between segmentation and paging? Why might both be used in a single system?

Segmentation allows specification of arbitrarily sized ranges of the address space that are valid and are used for a particular purpose, such as holding the process' code or stack. Paging is used to divide allocated memory space into smaller pieces that allow the virtual memory management system to load, relocate, and otherwise manage the space more flexibly.

Segmentation and paging might be used together to both specify the portions of the theoretical virtual memory space that a process is actually to use while allowing more flexible management of RAM use within those portions. Segmentation would specify the set of address ranges that are legally addressable, while paging would allow the OS to locate small sections of those address ranges in arbitrary page frames, or to move them off to disk, as necessary.

Grading guidelines:

- *6 points for proper description of the difference between segmentation and paging.*
- *4 points for description of why both might be useful*

References: Arpaci-Dusseau chapter 16 pages 1-4 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf) and Arpaci-Dusseau Chapter 18 pages 1-5 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf), lecture 5 slides 58-62 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_5.pdf) and lecture 6 slides 6-9 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_6.pdf)

COMP 300E Operating Systems
Fall Semester 2011
Midterm Examination Solution

Time/Date: 5:30 – 6:30 pm Oct 19, 2011 (Wed)

Name: _____ Student ID: _____

Instructions:

1. Please write your *name*, *student ID*, and *section number* **on this page**.
2. This is a **CLOSED** book exam!
3. This examination paper consists of **5** questions and **10** pages (including this page).
4. You have 60 minutes to complete this exam.
5. Please answer all questions within the space provided on the paper. Be concise!
6. Please read each question very carefully and answer the question clearly to the point. Make sure that your answers are neatly written, legible, and readable.

Question	Points	Score
1	**20**	
2	**20**	
3	**20**	
4	**20**	
5	**20**	
Total	**100**	

1. **(20 pts) Short Q&A**

1.1 (3 pts) What are the optimal scheduling algorithms for minimizing average response time, for the case of non-preemptive scheduling and the case of preemptive scheduling, respectively?

Shortest Job First (SJF) among non-preemptive algorithms, Shortest Remaining Time First (SRTF) among preemptive algorithms

1.2 (3 pts) If the time quantum for Round-Robin scheduling is set to be very large (approaching infinity), then it becomes equivalent to what scheduling algorithm?

FCFS or FIFO

1.3 (3 pts) Name one common hardware instruction that is useful for implementing critical sections.

TestAndSet,or CompareAndSwap

1.4 (3 pts) The signal() operation is used with both semaphores (also called V() operation) and monitors. Explain the key difference in its runtime behavior in the two cases. (Hint: consider what happens when a thread signals, but no other thread is waiting.)

When the signal() operation is used in monitors, if a signal is performed and if there are no waiting processes, the signal is simply ignored and the system does not remember the fact that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks.

In semaphores, on the other hand, every signal results in a corresponding increment of the semaphore value even if there are no process waiting. A future wait() operation would immediately succeed because of the earlier increment.

1.5 (3 pts) Explain the key drawback of spinlocks with busy-waiting compared to binary semaphores with sleeping.

Spin locks waste CPU time.

1.6 (2 pts) List the four necessary conditions for deadlocks.

Mutual exclusion, Hold and wait, No preemption, Circular wait

1.7 (3 pts) What is the main limitation of Resource Allocation Graph used for deadlock detection, as compared to Banker's algorithm?

It can only detect deadlocks for the case of single-instance resources.

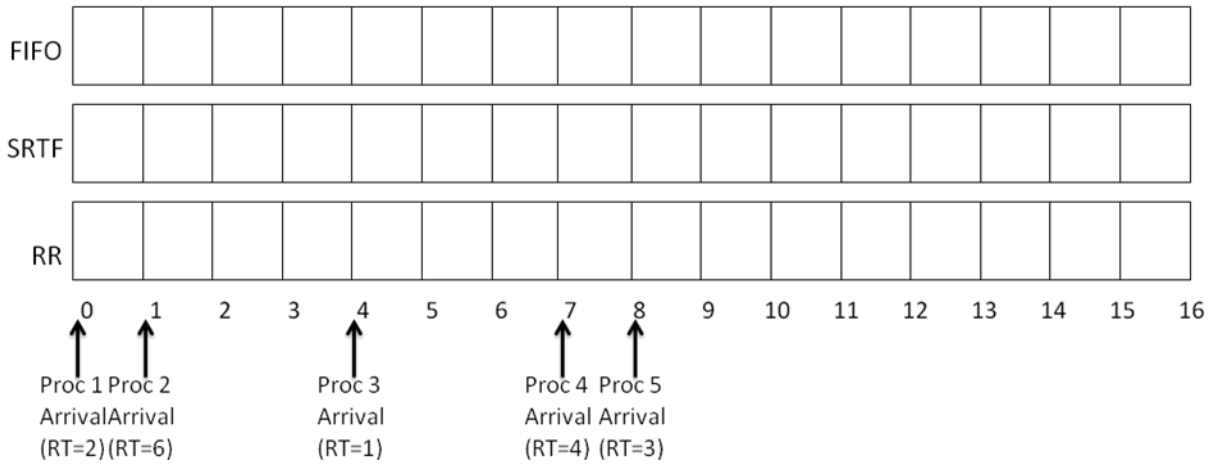
2. (20 pts) Processor Scheduling.

Here is a table of processes and their associated arrival and running times.

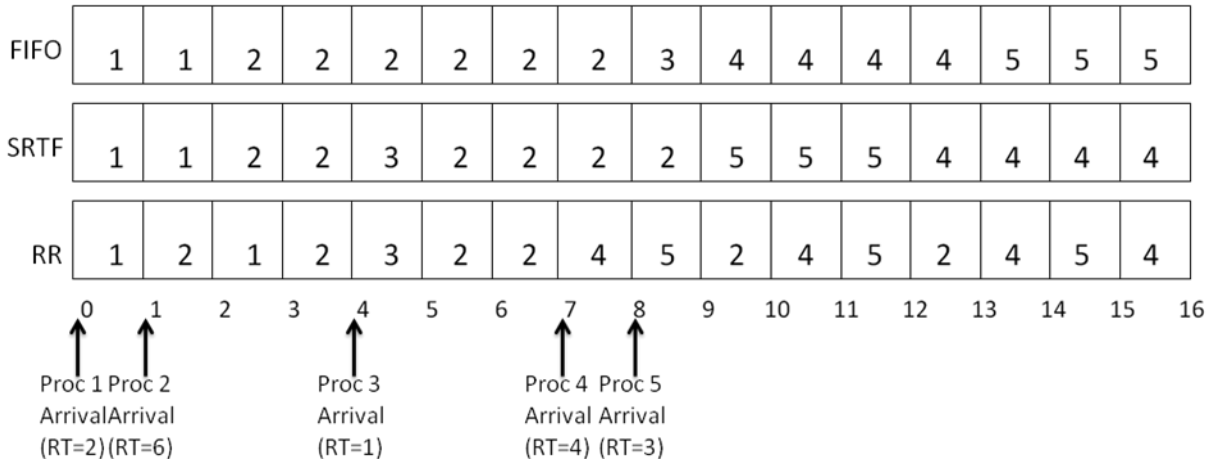
Process ID	**Arrival Time**	**Running Time**
1	0	2
2	1	6
3	4	1
4	7	4
5	8	3

a. (12 points)

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1, by filling in the Gantt chart with ID of the process currently running in each time quantum. *Assume that context switch overhead is 0 and that new RR processes are added to the **head** of the queue and new FCFS processes are added to the **tail** of the queue.*



Answer:



b. (8 points) Compute the response time for each process in each schedule above.

Scheduler	Process 1	Process 2	Process 3	Process 4	Process 5
FIFO	2	7	5	6	8
SRTF	2	8	1	9	4
RR	3	12	1	9	7

3. (20 pts) Synchronization

Consider the following semaphore-based solution to the readers-writers problem:
Common initialization section for both writer and reader:
semaphore wrt=1, mutex=1;
int readcount=0;

The code bodies for writer and reader::

Writer	**Reader**
Writer() { *wrt.P() ;* *//writing is performed* *wrt.V() ;* *}*	*Reader(){* *mutex.P() ;* *readcount ++ ;* *if (readcount == 1)* *wrt.P() ;* *mutex.V();* *// reading is performed* *mutex.P() ;* *readcount - - ;* *if (readcount == 0)* *wrt.V() ;* *mutex.V() ;* *}*

1) (10 pts) Fill in the underlined areas in Reader() to make it a complete solution to the readers-writers problem. Your solution should allow multiple concurrent readers, but only one writer, to access the shared database simultaneously.

First if statement: if (readcount == 1) wrt.P() ;
Second if statement: if (readcount == 0) wrt.V() ;

("wrt" guarantees mutual exclusion to the critical section in the presence of writers. "mutex" guarantees mutual exclusion when updating the shared variable readcount.)

2) (4 pts) Suppose a writer thread is inside its critical section performing writing, while another writer and n>1 readers are waiting outside their critical sections. Which semaphores are they waiting on, respectively?

writer is waiting on wrt; the 1st reader is waiting on wrt and the other n-1 readers are *waiting on mutex*.

3) (4 pts) Does this solution give preference to waiting readers or waiting writers? Why?

This solution gives readers preference over writers; once one reader is accessing the database, all readers are allowed to read the database without performing the P operations on wrt.

Note that we are not explicitly checking the number of waiting readers or waiting writers like the solution in class. For example, if there are both waiting readers and waiting writers when a writer calls wrt.V(), then either reader(s) or one writer can get ahead of each other nondeterministic ally. However, once one reader has already started reading, then other waiting and newly arrived readers have preference over any waiting writers.

Discussions (not part of the problem): Work out a monitor-based solution to this problem.

```
MonitorVar wrt=1;
Lock lock=FREE;
int readcount=0;
bool writing=FALSE;
```

The code bodies for writer and reader:

Writer	**Reader**

```
Writer() {
        lock.aquire();
        wrt.wait(&lock) ;
        writing=TRUE;
        lock.release();
        //writing is performed
        lock.aquire();
        writing=FALSE;
        wrt.signalAll();
        lock.release();
    }
```

```
Reader(){
        lock.aquire();
        readcount ++ ;
        while (writing)
            wrt.wait(&lock);
        lock.release();
        // reading is performed
        lock.aquire();;
        readcount - - ;
        if (readcount==0)
            wrt.signalAll() ;
        lock.release();
    }
```

Note: Compared to the solution in the class notes, this is a simplified solution that removes several checks to give writers preference, and uses a single condition variable instead of 2.

Suppose a writer thread is inside its critical section performing writing, while another writer and n>1 readers are waiting outside their critical sections. Then both the one writer and all n readers are waiting on wrt. *Recall that condition.wait(&lock) implicitly releases the lock upon going to sleep. Assuming Mesa style monitors, "while" needs to be used instead of "if" to check the boolean "writing".*

We should use signalAll() in both reader and writer code to wake up potentially multiple readers. If multiple writers are woken up, all but one will go to sleep again, so the program is still correct. Since only the last reader invokes wrt.signalAll(), using wrt.signal() here would not be correct, since multiple readers may be waiting on condition wrt, and they all need to be woken up.

4. (20 points total) Synchronization

In parallel programs (one multi-threaded process), a common design methodology is to perform processing in sequential stages. All of the threads work independently during each stage, but they must synchronize at the end of each stage at a synchronization point called a *barrier*. If a thread reaches the barrier before all other threads have arrived, it waits. When all threads reach the barrier, they are notified and can begin execution on the next phase of the computation.

There are three complications to barriers. First, there is no master thread that controls the threads, waits for each of them to reach the barrier, and then tells them to re-start. Instead, the threads must monitor themselves and determine when they should wait or proceed. Second, for many dynamic programs, the number of threads that will be created during the lifetime of the parallel program is unknown in advance, since a thread can spawn another thread, which will start in the same program stage as the thread that created it. Third, a thread may end before reaching the barrier. In all cases, all active threads (those that have not ended) must synchronize at the barrier before the processing is allowed to proceed to the next phase.

Provide the pseudo-code for a class called `Barrier` that enables this style of barrier synchronization. Your solution must support creation of a new thread (an additional thread that needs to synchronize), termination of a thread (one less thread

that needs to synchronize), waiting when a thread reaches the barrier early, and releasing waiting threads when the last thread reaches the barrier. *Implement your solution using monitors* (e.g., `wait()`, `signal()`, and `signalAll()`). Your class must implement the following three methods: `threadCreated`(), `threadEnd`(), `barrierReached`(). We have provided code skeletons below. Please fill in the blank areas.

```
Class Barrier (){
    condition barrier; Lock lock = FREE;
    int numThreads = 0; int numThreadsAtBarrier = 0;
    threadCreated(){
        lock.acquire();
        numThreads++;
        lock.release();
    }
    threadEnd(){
        lock.acquire();
        numThreads--;
        if (numThreadsAtBarrier == numThreads){
           numThreadsAtBarrier = 0;
           barrier.signalAll();
        }
        lock.release();
    }
    barrierReached(){
        lock.acquire();
        numThreadsAtBarrier++;
        if (numThreadsAtBarrier < numThreads){
            barrier.wait(&lock);
        } else {
            numThreadsAtBarrier = 0;
            barrier.signalAll();
        }
        lock.release();
    }
}
```

We subtracted points as follow:

- *Answers with semaphores received no credit*
- *Answers without locks lost 10 points, incorrect lock use lost 2 points for each error (max –10 points)*
- *If your threadEnd procedure didn't check the ending thread was the last one not at the barrier, you lost 4 points*
- *If threads could slip through your barrierReached, you lost 6 to 10 points*

- *If not all threads were released after the barrier, you lost 8 points*
- *Excessive thread wakeups (e.g., in threadEnd) cost you 2 points*
- *Answers with indefinite waiting (never finishing the barrier) lost 10 points*
- *Answers with busy waiting lost 10 points*

5. (20 pts) Deadlocks

Consider the following snapshot of a system:

	C (Current allocation matrix)					R (Total Request matrix)					A (Available resource vector)			
	A	**B**	**C**	**D**		**A**	**B**	**C**	**D**		**A**	**B**	**C**	**D**
P0	0	0	1	2		0	0	3	2		2	1	2	0
P1	2	0	0	0		2	7	5	0					
P2	0	0	3	4		6	6	5	6					
P3	2	3	5	4		4	3	5	6					
P4	0	3	3	2		0	6	5	2					

Answer the following questions using Banker's algorithm:

a. (12 pts) Determine if the system is currently in a safe state using Banker's algorithm. (Hint: first determine the matrix of additional resources needed by each process, then find a sequence of process executions that can complete successfully. We have provided the tables below, so you only need to fill in the blanks.)

Answer: R – C = Need (maxtrix)

	Still Needed			
	A	**B**	**C**	**D**
P0	0	0	2	0
P1	0	7	5	0
P2	6	6	2	2
P3	2	0	0	2
P4	0	3	2	0

The allocation should be safe right now, with a sequence of process executions.

Answer: Yes, with <P0, P3, P4, P1, P2>.

	Resources available after each process has finished			
	A	**B**	**C**	**D**
P0	2	1	3	2
P3	4	4	8	6

P4	4	7	11	8
P1	6	7	11	8
P2	6	7	14	12

b. (8 pts) If a request from process P2 arrives for resources (0, 1, 2, 0), can the requested be granted immediately? If yes, then show the sequence of successful process executions in tabular form; if no, then you do not need to show any further details.

Answer: No, this cannot be allocated. If this is allocated, the resulting Available() is (2, 0, 0, 0), there is no sequence of the process execution order that lead to the completion of all processes. This is an unsafe state.

EURECOM
Sophia Antipolis

Exam: Solutions
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 8th, 2013

Authorized documents: Nothing! The grading takes into account the fact that you don't have any document with you.
A grade is provided for every question (beware: do organize your time). 1 additional point is given as a general appreciation, including written skills and readability.

This is only an example of one possible solution to get the maximum grade. So, you can get the maximum grade with a different solution. Also, contact me if you think my solution can be improved

1 Understanding of the course (5 points, ∼30 minutes)

1. What is the main purpose of swapping? Can a process be run by an Operating System if some of its pages are swapped out? [2.5 points]

   The main purpose of swapping is to have a larger storage facility to store data allocated by processes. That larger storage is commonly a hard disk, which is much slower than the main memory.
   If we assume a virtual memory managed by the Operating System, and supported by a MMU, then, all pages under manipulation by a running process must be swapped in before they can be used. Thus, if a process does not need some of its pages, yes, it can be run with some of its currently-not-used-pages swapped out.

2. What are the two techniques that are commonly used by device drivers to exchange information with devices? Explain the two, and explain in which situations they are efficient, or not. [2.5 points]

   The two main techniques are polling and interrupts.

Polling consists in periodically reading a given register (e.g., status register) of a device to know whether the latter has completed its operation, or not.
In the interrupt technique, the device asserts a hardware signal - called interrupt - to the microprocessor. The corresponding interrupt service routine set up by the Operating System at boot up will then be called.
Polling can be used when the delay between two operations is short and known: it avoids interrupting the processor. When that delay is likely to be long, or is unknown, interrupts give a chance to the Operating System to schedule other processes while waiting for the interrupt to occur.

2 Memory allocation (6 points, ∼40 minutes)

Memory allocated by Operating Systems is usually a multiple of a given memory page size. Operating Systems commonly store references to allocated pages in linked lists. Yet, programmers like to allocate a memory chunk whose size is not necessarily a multiple of memory pages handled by the Operating System. To do so, user-level libraries manages more fine-grained chunks of memory. For example, *malloc()* is a user-level library function which handles random sizes of memory allocations. That is, that library function allocates necessary pages using system calls (e.g., *mmap()*, *brk()*), and manages allocations within pages using its own data structures (e.g., linked lists).

1. Why isn't it the Operating System directly handling fine-grained allocations? [1 point]

   For efficiency reasons, the Memory Management Unit should manage lookup tables of reasonable size. Therefore, an Operating system can only handle pages and segments whose size is at least a few kbytes. Finer-grained allocations are thus managed at user-level e.g., by the libC (malloc, etc.).

2. Let's now consider the following code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  char *str;

  str = (char *)malloc(sizeof(char)*4);
  strcpy(str, "12345678\n");
  printf("str=%s", str);
  free(str);
}
```

   a. At execution, an error may occur when executing *strcpy*: could you explain it precisely, i.e., what is the cause of the error, and how it is detected at run time? In particular, you should explain why sometimes no error occurs. [2 points]

      The program first allocates 4 characters at address str. Then, 10 characters are

copied at address str, including the NULL character (end of the string). Either the address str+10 is in a page already allocated by the process, and in that case, no error occurs. Or, str+10 is outside the process address space, so when the process tries to access to that address, the MMU generates a page fault exception, which result in the process to be killed by the Operating System.

b. Again at execution, an error may also occur when executing *free*: could you explain it precisely, i.e., what is the cause of the error, and how it is detected at run time? [3 points]

The process was able to execute the strcpy instruction without being killed by the operating system, i.e. str+10 is part of the process address space. Thus, if an error occurs when calling free, it means that the function free could not be correctly executed, which probably means that the data structure managing the memory at user-level was corrupted by the 5 extra characters written at address str.

3 POSIX programming (8 points, ∼50 minutes)

Let's consider the following code.
(Memo: pthread_yield() causes the calling thread to relinquish the CPU.)

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;
pthread_t a, b;

void *f(void *param) {
  while (1)
    {
      pthread_mutex_lock(&m);
      printf("%s", param);
      pthread_mutex_unlock(&m);
      pthread_yield();
    }
}

int main() {
  pthread_mutex_init(&m, NULL);
  pthread_create(&a, NULL, f, "Hello ");
  pthread_create(&b, NULL, f, "World\n");

  pthread_join(a, NULL);
  pthread_join(b, NULL);
}
```

1. Give two possible traces of execution. [1 point]

First trace: the World process executes first. We assume that the yield operation provokes a switch to Hello, and then to World, etc.

World
Hello World
Hello World
etc.

Second trace: Hello starts first, but World takes longer to start, and yield does not work as we would like to, i.e., the same thread is given several times the processor again. One possible trace of this would be: Hello Hello Hello World
World
Hello Hello World
etc.

Of course, the second trace is more likely to occur than the first one.

2. Modify this code so as to have "Hello World" printed on each line. [2 points]

We need to alternate between the two threads, and yield is not the right way to do: we should rather use a global variable to do so, and two conditions variables in order to avoid threads to busy wait. To do so, we have split the code of threads in two different functions ("hello", "world"), and declared two condition variables ("toWorld", "toHello") and the global "alternate" variable. Finally, the code is as follows:

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;
pthread_cond_t toWorld, toHello;
pthread_t a, b;

int alternate = 0;

void *hello(void *param)
{
  while (1)
    {
      pthread_mutex_lock(&m);
      while (alternate != 0) {
        pthread_cond_wait(&toHello, &m);
      }
      printf("%s", param);
      alternate = 1;
      pthread_cond_signal(&toWorld);
      pthread_mutex_unlock(&m);
    }
}

void *world(void *param)
{
  while (1)
    {
      pthread_mutex_lock(&m);
      while (alternate != 1) {
        pthread_cond_wait(&toWorld, &m);
      }
```

```
        printf("%s", param);
        alternate = 0;
        pthread_cond_signal(&toHello);
        pthread_mutex_unlock(&m);
      }
}

int main()
{
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&toHello, NULL);
  pthread_cond_init(&toWorld, NULL);
  pthread_create(&a, NULL, hello, "Hello ");
  pthread_create(&b, NULL, world, "World\n");

  pthread_join(a, NULL);
}
```

3. Enhance the code so as to print "Hello World" exactly 10 times. [1 point]

   We need a counter to do so. We have added a "cpt" counter to each thread, and we have defined "MAX_HELLO" to 10. Also, when the two threads terminate after 10 iterations, the main thread can do the join on the two threads. (I provide only part of the code)

```
...
#define MAX_HELLO 10

...

void *hello(void *param)
{
  int cpt = 0;
  while (cpt < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      ...
      pthread_mutex_unlock(&m);
      cpt ++;
    }
}

// similarly for world
...

int main()
{
  ...

  pthread_join(a, NULL);
  pthread_join(b, NULL);
}
```

4. Now, we would like to have two threads being able to print "Hello" and two printing "World". Synchronize those 4 threads so as to print exactly 10 times "Hello World". [4 points]

We need to start four threads. We also need also to ensure the synchronization between the four threads (for printing hello world correctly), and inside of each group of two, for not printing more that 10 times hello world. A first non working solution would be to simply execute the same code as previously with two variables nbOfWorld and nbOfHello (instead of the cpt variables) being global. Actually, this solution does not work because, when the last "hello" is being written, the other hello thread might already be waiting in the condition variable, and will thus be woken up later on by the last world thread, thus printing a 11th hello: we thus need to retest the number of hellos and worlds printed when waking up from waiting on the condition variable. finally, the solution is as follows.

```
#include <pthread.h>
#include <stdio.h>

#define MAX_HELLO 10

pthread_mutex_t m;
pthread_cond_t toWorld, toHello;
pthread_t a1, b1, a2, b2;

int alternate = 0;

int nbOfWorld = 0;
int nbOfHello = 0;

void *hello(void *param)
{
  while (nbOfHello < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      while (alternate != 0) {
        pthread_cond_wait(&toHello, &m);
      }
      if (nbOfHello < MAX_HELLO) {
        printf("%s", param);
        nbOfHello ++;
      }
      alternate = 1;
      pthread_cond_signal(&toWorld);
      pthread_mutex_unlock(&m);
    }
}

void *world(void *param)
{
  int cpt = 0;
  while (nbOfWorld < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      while (alternate != 1) {
        pthread_cond_wait(&toWorld, &m);
      }
      if (nbOfWorld < MAX_HELLO) {
        printf("%s", param);
        nbOfWorld ++;
      }
      alternate = 0;
```

```
      pthread_cond_signal(&toHello);
      pthread_mutex_unlock(&m);
    }
}

int main()
{
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&toHello, NULL);
  pthread_cond_init(&toWorld, NULL);
  pthread_create(&a1, NULL, hello, "Hello ");
  pthread_create(&a2, NULL, hello, "Hello ");
  pthread_create(&b1, NULL, world, "World\n");
  pthread_create(&b2, NULL, world, "World\n");

  pthread_join(a1, NULL);
  pthread_join(b1, NULL);
  pthread_join(a2, NULL);
  pthread_join(b2, NULL);
}
```

COMP 300E Operating Systems
Fall Semester 2011
Midterm Examination Solution

Time/Date: 5:30 – 6:30 pm Oct 19, 2011 (Wed)

Name: _____ Student ID: _____

Instructions:

1. Please write your *name*, *student ID*, and *section number* **on this page**.
2. This is a **CLOSED** book exam!
3. This examination paper consists of **5** questions and **10** pages (including this page).
4. You have 60 minutes to complete this exam.
5. Please answer all questions within the space provided on the paper. Be concise!
6. Please read each question very carefully and answer the question clearly to the point. Make sure that your answers are neatly written, legible, and readable.

Question	Points	Score
1	**20**	
2	**20**	
3	**20**	
4	**20**	
5	**20**	
Total	**100**	

1. **(20 pts) Short Q&A**

1.1 (3 pts) What are the optimal scheduling algorithms for minimizing average response time, for the case of non-preemptive scheduling and the case of preemptive scheduling, respectively?

Shortest Job First (SJF) among non-preemptive algorithms, Shortest Remaining Time First (SRTF) among preemptive algorithms

1.2 (3 pts) If the time quantum for Round-Robin scheduling is set to be very large (approaching infinity), then it becomes equivalent to what scheduling algorithm?

FCFS or FIFO

1.3 (3 pts) Name one common hardware instruction that is useful for implementing critical sections.

TestAndSet,or CompareAndSwap

1.4 (3 pts) The signal() operation is used with both semaphores (also called V() operation) and monitors. Explain the key difference in its runtime behavior in the two cases. (Hint: consider what happens when a thread signals, but no other thread is waiting.)

When the signal() operation is used in monitors, if a signal is performed and if there are no waiting processes, the signal is simply ignored and the system does not remember the fact that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks.

In semaphores, on the other hand, every signal results in a corresponding increment of the semaphore value even if there are no process waiting. A future wait() operation would immediately succeed because of the earlier increment.

1.5 (3 pts) Explain the key drawback of spinlocks with busy-waiting compared to binary semaphores with sleeping.

Spin locks waste CPU time.

1.6 (2 pts) List the four necessary conditions for deadlocks.

Mutual exclusion, Hold and wait, No preemption, Circular wait

1.7 (3 pts) What is the main limitation of Resource Allocation Graph used for deadlock detection, as compared to Banker's algorithm?

It can only detect deadlocks for the case of single-instance resources.

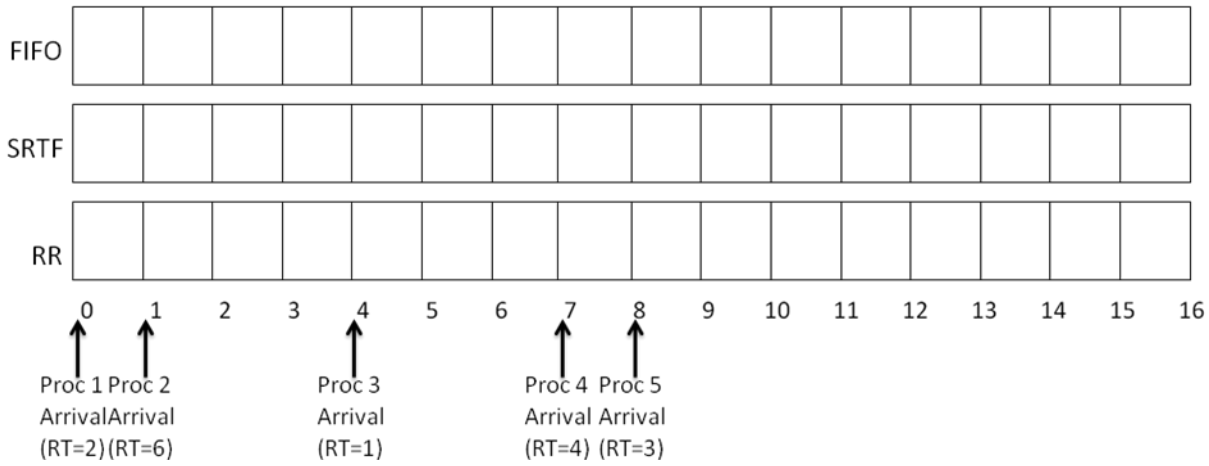
2. (20 pts) Processor Scheduling.

Here is a table of processes and their associated arrival and running times.

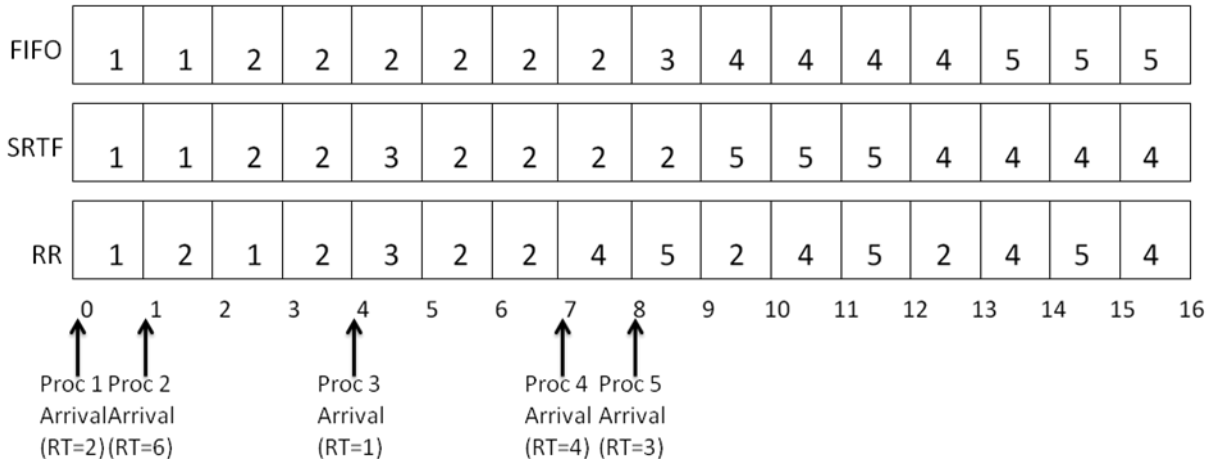
Process ID	**Arrival Time**	**Running Time**
1	0	2
2	1	6
3	4	1
4	7	4
5	8	3

a. (12 points)

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1, by filling in the Gantt chart with ID of the process currently running in each time quantum. *Assume that context switch overhead is 0 and that new RR processes are added to the **head** of the queue and new FCFS processes are added to the **tail** of the queue.*



Answer:



b. (8 points) Compute the response time for each process in each schedule above.

Scheduler	Process 1	Process 2	Process 3	Process 4	Process 5
FIFO	2	7	5	6	8
SRTF	2	8	1	9	4
RR	3	12	1	9	7

3. (20 pts) Synchronization

Consider the following semaphore-based solution to the readers-writers problem:
Common initialization section for both writer and reader:
semaphore wrt=1, mutex=1;
int readcount=0;

The code bodies for writer and reader::

Writer	**Reader**
Writer() { *wrt.P() ;* *//writing is performed* *wrt.V() ;* *}*	*Reader(){* *mutex.P() ;* *readcount ++ ;* *if (readcount == 1)* *wrt.P() ;* *mutex.V();* *// reading is performed* *mutex.P() ;* *readcount - - ;* *if (readcount == 0)* *wrt.V() ;* *mutex.V() ;* *}*

1) (10 pts) Fill in the underlined areas in Reader() to make it a complete solution to the readers-writers problem. Your solution should allow multiple concurrent readers, but only one writer, to access the shared database simultaneously.

First if statement: if (readcount == 1) wrt.P() ;
Second if statement: if (readcount == 0) wrt.V() ;

("wrt" guarantees mutual exclusion to the critical section in the presence of writers. "mutex" guarantees mutual exclusion when updating the shared variable readcount.)

2) (4 pts) Suppose a writer thread is inside its critical section performing writing, while another writer and n>1 readers are waiting outside their critical sections. Which semaphores are they waiting on, respectively?

writer is waiting on wrt; the 1st reader is waiting on wrt and the other n-1 readers are *waiting on mutex*.

3) (4 pts) Does this solution give preference to waiting readers or waiting writers? Why?

This solution gives readers preference over writers; once one reader is accessing the database, all readers are allowed to read the database without performing the P operations on wrt.

Note that we are not explicitly checking the number of waiting readers or waiting writers like the solution in class. For example, if there are both waiting readers and waiting writers when a writer calls wrt.V(), then either reader(s) or one writer can get ahead of each other nondeterministic ally. However, once one reader has already started reading, then other waiting and newly arrived readers have preference over any waiting writers.

Discussions (not part of the problem): Work out a monitor-based solution to this problem.

```
MonitorVar wrt=1;
Lock lock=FREE;
int readcount=0;
bool writing=FALSE;
```

The code bodies for writer and reader:

Writer	**Reader**

```
Writer() {
        lock.aquire();
        wrt.wait(&lock) ;
        writing=TRUE;
        lock.release();
        //writing is performed
        lock.aquire();
        writing=FALSE;
        wrt.signalAll();
        lock.release();
    }
```

```
Reader(){
        lock.aquire();
        readcount ++ ;
        while (writing)
            wrt.wait(&lock);
        lock.release();
        // reading is performed
        lock.aquire();;
        readcount - - ;
        if (readcount==0)
            wrt.signalAll() ;
        lock.release();
    }
```

Note: Compared to the solution in the class notes, this is a simplified solution that removes several checks to give writers preference, and uses a single condition variable instead of 2.

Suppose a writer thread is inside its critical section performing writing, while another writer and n>1 readers are waiting outside their critical sections. Then both the one writer and all n readers are waiting on wrt. *Recall that condition.wait(&lock) implicitly releases the lock upon going to sleep. Assuming Mesa style monitors, "while" needs to be used instead of "if" to check the boolean "writing".*

We should use signalAll() in both reader and writer code to wake up potentially multiple readers. If multiple writers are woken up, all but one will go to sleep again, so the program is still correct. Since only the last reader invokes wrt.signalAll(), using wrt.signal() here would not be correct, since multiple readers may be waiting on condition wrt, and they all need to be woken up.

4. (20 points total) Synchronization

In parallel programs (one multi-threaded process), a common design methodology is to perform processing in sequential stages. All of the threads work independently during each stage, but they must synchronize at the end of each stage at a synchronization point called a *barrier*. If a thread reaches the barrier before all other threads have arrived, it waits. When all threads reach the barrier, they are notified and can begin execution on the next phase of the computation.

There are three complications to barriers. First, there is no master thread that controls the threads, waits for each of them to reach the barrier, and then tells them to re-start. Instead, the threads must monitor themselves and determine when they should wait or proceed. Second, for many dynamic programs, the number of threads that will be created during the lifetime of the parallel program is unknown in advance, since a thread can spawn another thread, which will start in the same program stage as the thread that created it. Third, a thread may end before reaching the barrier. In all cases, all active threads (those that have not ended) must synchronize at the barrier before the processing is allowed to proceed to the next phase.

Provide the pseudo-code for a class called `Barrier` that enables this style of barrier synchronization. Your solution must support creation of a new thread (an additional thread that needs to synchronize), termination of a thread (one less thread

that needs to synchronize), waiting when a thread reaches the barrier early, and releasing waiting threads when the last thread reaches the barrier. *Implement your solution using monitors* (e.g., `wait()`, `signal()`, and `signalAll()`). Your class must implement the following three methods: `threadCreated`(), `threadEnd`(), `barrierReached`(). We have provided code skeletons below. Please fill in the blank areas.

```
Class Barrier (){
    condition barrier; Lock lock = FREE;
    int numThreads = 0; int numThreadsAtBarrier = 0;
    threadCreated(){
        lock.acquire();
        numThreads++;
        lock.release();
    }
    threadEnd(){
        lock.acquire();
        numThreads--;
        if (numThreadsAtBarrier == numThreads){
           numThreadsAtBarrier = 0;
           barrier.signalAll();
        }
        lock.release();
    }
    barrierReached(){
        lock.acquire();
        numThreadsAtBarrier++;
        if (numThreadsAtBarrier < numThreads){
            barrier.wait(&lock);
        } else {
            numThreadsAtBarrier = 0;
            barrier.signalAll();
        }
        lock.release();
    }
}
```

We subtracted points as follow:

- *Answers with semaphores received no credit*
- *Answers without locks lost 10 points, incorrect lock use lost 2 points for each error (max –10 points)*
- *If your threadEnd procedure didn't check the ending thread was the last one not at the barrier, you lost 4 points*
- *If threads could slip through your barrierReached, you lost 6 to 10 points*

- *If not all threads were released after the barrier, you lost 8 points*
- *Excessive thread wakeups (e.g., in threadEnd) cost you 2 points*
- *Answers with indefinite waiting (never finishing the barrier) lost 10 points*
- *Answers with busy waiting lost 10 points*

5. (20 pts) Deadlocks

Consider the following snapshot of a system:

	C (Current allocation matrix)					R (Total Request matrix)					A (Available resource vector)			
	A	**B**	**C**	**D**		**A**	**B**	**C**	**D**		**A**	**B**	**C**	**D**
P0	0	0	1	2		0	0	3	2		2	1	2	0
P1	2	0	0	0		2	7	5	0					
P2	0	0	3	4		6	6	5	6					
P3	2	3	5	4		4	3	5	6					
P4	0	3	3	2		0	6	5	2					

Answer the following questions using Banker's algorithm:

a. (12 pts) Determine if the system is currently in a safe state using Banker's algorithm. (Hint: first determine the matrix of additional resources needed by each process, then find a sequence of process executions that can complete successfully. We have provided the tables below, so you only need to fill in the blanks.)

Answer: R – C = Need (maxtrix)

	Still Needed			
	A	**B**	**C**	**D**
P0	0	0	2	0
P1	0	7	5	0
P2	6	6	2	2
P3	2	0	0	2
P4	0	3	2	0

The allocation should be safe right now, with a sequence of process executions.

Answer: Yes, with <P0, P3, P4, P1, P2>.

	Resources available after each process has finished			
	A	**B**	**C**	**D**
P0	2	1	3	2
P3	4	4	8	6

P4	4	7	11	8
P1	6	7	11	8
P2	6	7	14	12

b. (8 pts) If a request from process P2 arrives for resources (0, 1, 2, 0), can the requested be granted immediately? If yes, then show the sequence of successful process executions in tabular form; if no, then you do not need to show any further details.

Answer: No, this cannot be allocated. If this is allocated, the resulting Available() is (2, 0, 0, 0), there is no sequence of the process execution order that lead to the completion of all processes. This is an unsafe state.

EURECOM
Sophia Antipolis

Exam: Solutions
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 8th, 2013

Authorized documents: Nothing! The grading takes into account the fact that you don't have any document with you.
A grade is provided for every question (beware: do organize your time). 1 additional point is given as a general appreciation, including written skills and readability.

This is only an example of one possible solution to get the maximum grade. So, you can get the maximum grade with a different solution. Also, contact me if you think my solution can be improved

1 Understanding of the course (5 points, ∼30 minutes)

1. What is the main purpose of swapping? Can a process be run by an Operating System if some of its pages are swapped out? [2.5 points]

   The main purpose of swapping is to have a larger storage facility to store data allocated by processes. That larger storage is commonly a hard disk, which is much slower than the main memory.
   If we assume a virtual memory managed by the Operating System, and supported by a MMU, then, all pages under manipulation by a running process must be swapped in before they can be used. Thus, if a process does not need some of its pages, yes, it can be run with some of its currently-not-used-pages swapped out.

2. What are the two techniques that are commonly used by device drivers to exchange information with devices? Explain the two, and explain in which situations they are efficient, or not. [2.5 points]

   The two main techniques are polling and interrupts.

Polling consists in periodically reading a given register (e.g., status register) of a device to know whether the latter has completed its operation, or not.
In the interrupt technique, the device asserts a hardware signal - called interrupt - to the microprocessor. The corresponding interrupt service routine set up by the Operating System at boot up will then be called.
Polling can be used when the delay between two operations is short and known: it avoids interrupting the processor. When that delay is likely to be long, or is unknown, interrupts give a chance to the Operating System to schedule other processes while waiting for the interrupt to occur.

2 Memory allocation (6 points, ∼40 minutes)

Memory allocated by Operating Systems is usually a multiple of a given memory page size. Operating Systems commonly store references to allocated pages in linked lists. Yet, programmers like to allocate a memory chunk whose size is not necessarily a multiple of memory pages handled by the Operating System. To do so, user-level libraries manages more fine-grained chunks of memory. For example, *malloc()* is a user-level library function which handles random sizes of memory allocations. That is, that library function allocates necessary pages using system calls (e.g., *mmap()*, *brk()*), and manages allocations within pages using its own data structures (e.g., linked lists).

1. Why isn't it the Operating System directly handling fine-grained allocations? [1 point]

   For efficiency reasons, the Memory Management Unit should manage lookup tables of reasonable size. Therefore, an Operating system can only handle pages and segments whose size is at least a few kbytes. Finer-grained allocations are thus managed at user-level e.g., by the libC (malloc, etc.).

2. Let's now consider the following code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  char *str;

  str = (char *)malloc(sizeof(char)*4);
  strcpy(str, "12345678\n");
  printf("str=%s", str);
  free(str);
}
```

   a. At execution, an error may occur when executing *strcpy*: could you explain it precisely, i.e., what is the cause of the error, and how it is detected at run time? In particular, you should explain why sometimes no error occurs. [2 points]

      The program first allocates 4 characters at address str. Then, 10 characters are

copied at address str, including the NULL character (end of the string). Either the address str+10 is in a page already allocated by the process, and in that case, no error occurs. Or, str+10 is outside the process address space, so when the process tries to access to that address, the MMU generates a page fault exception, which result in the process to be killed by the Operating System.

b. Again at execution, an error may also occur when executing *free*: could you explain it precisely, i.e., what is the cause of the error, and how it is detected at run time? [3 points]

The process was able to execute the strcpy instruction without being killed by the operating system, i.e. str+10 is part of the process address space. Thus, if an error occurs when calling free, it means that the function free could not be correctly executed, which probably means that the data structure managing the memory at user-level was corrupted by the 5 extra characters written at address str.

3 POSIX programming (8 points, ∼50 minutes)

Let's consider the following code.
(Memo: pthread_yield() causes the calling thread to relinquish the CPU.)

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;
pthread_t a, b;

void *f(void *param) {
  while (1)
    {
      pthread_mutex_lock(&m);
      printf("%s", param);
      pthread_mutex_unlock(&m);
      pthread_yield();
    }
}

int main() {
  pthread_mutex_init(&m, NULL);
  pthread_create(&a, NULL, f, "Hello ");
  pthread_create(&b, NULL, f, "World\n");

  pthread_join(a, NULL);
  pthread_join(b, NULL);
}
```

1. Give two possible traces of execution. [1 point]

First trace: the World process executes first. We assume that the yield operation provokes a switch to Hello, and then to World, etc.

World
Hello World
Hello World
etc.

Second trace: Hello starts first, but World takes longer to start, and yield does not work as we would like to, i.e., the same thread is given several times the processor again. One possible trace of this would be: Hello Hello Hello World
World
Hello Hello World
etc.

Of course, the second trace is more likely to occur than the first one.

2. Modify this code so as to have "Hello World" printed on each line. [2 points]

We need to alternate between the two threads, and yield is not the right way to do: we should rather use a global variable to do so, and two conditions variables in order to avoid threads to busy wait. To do so, we have split the code of threads in two different functions ("hello", "world"), and declared two condition variables ("toWorld", "toHello") and the global "alternate" variable. Finally, the code is as follows:

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;
pthread_cond_t toWorld, toHello;
pthread_t a, b;

int alternate = 0;

void *hello(void *param)
{
  while (1)
    {
      pthread_mutex_lock(&m);
      while (alternate != 0) {
        pthread_cond_wait(&toHello, &m);
      }
      printf("%s", param);
      alternate = 1;
      pthread_cond_signal(&toWorld);
      pthread_mutex_unlock(&m);
    }
}

void *world(void *param)
{
  while (1)
    {
      pthread_mutex_lock(&m);
      while (alternate != 1) {
        pthread_cond_wait(&toWorld, &m);
      }
```

```
        printf("%s", param);
        alternate = 0;
        pthread_cond_signal(&toHello);
        pthread_mutex_unlock(&m);
      }
}

int main()
{
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&toHello, NULL);
  pthread_cond_init(&toWorld, NULL);
  pthread_create(&a, NULL, hello, "Hello ");
  pthread_create(&b, NULL, world, "World\n");

  pthread_join(a, NULL);
}
```

3. Enhance the code so as to print "Hello World" exactly 10 times. [1 point]

   We need a counter to do so. We have added a "cpt" counter to each thread, and we have defined "MAX_HELLO" to 10. Also, when the two threads terminate after 10 iterations, the main thread can do the join on the two threads. (I provide only part of the code)

```
...
#define MAX_HELLO 10

...

void *hello(void *param)
{
  int cpt = 0;
  while (cpt < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      ...
      pthread_mutex_unlock(&m);
      cpt ++;
    }
}

// similarly for world
...

int main()
{
  ...

  pthread_join(a, NULL);
  pthread_join(b, NULL);
}
```

4. Now, we would like to have two threads being able to print "Hello" and two printing "World". Synchronize those 4 threads so as to print exactly 10 times "Hello World". [4 points]

We need to start four threads. We also need also to ensure the synchronization between the four threads (for printing hello world correctly), and inside of each group of two, for not printing more that 10 times hello world. A first non working solution would be to simply execute the same code as previously with two variables nbOfWorld and nbOfHello (instead of the cpt variables) being global. Actually, this solution does not work because, when the last "hello" is being written, the other hello thread might already be waiting in the condition variable, and will thus be woken up later on by the last world thread, thus printing a 11th hello: we thus need to retest the number of hellos and worlds printed when waking up from waiting on the condition variable. finally, the solution is as follows.

```
#include <pthread.h>
#include <stdio.h>

#define MAX_HELLO 10

pthread_mutex_t m;
pthread_cond_t toWorld, toHello;
pthread_t a1, b1, a2, b2;

int alternate = 0;

int nbOfWorld = 0;
int nbOfHello = 0;

void *hello(void *param)
{
  while (nbOfHello < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      while (alternate != 0) {
        pthread_cond_wait(&toHello, &m);
      }
      if (nbOfHello < MAX_HELLO) {
        printf("%s", param);
        nbOfHello ++;
      }
      alternate = 1;
      pthread_cond_signal(&toWorld);
      pthread_mutex_unlock(&m);
    }
}

void *world(void *param)
{
  int cpt = 0;
  while (nbOfWorld < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      while (alternate != 1) {
        pthread_cond_wait(&toWorld, &m);
      }
      if (nbOfWorld < MAX_HELLO) {
        printf("%s", param);
        nbOfWorld ++;
      }
      alternate = 0;
```

```
      pthread_cond_signal(&toHello);
      pthread_mutex_unlock(&m);
    }
}

int main()
{
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&toHello, NULL);
  pthread_cond_init(&toWorld, NULL);
  pthread_create(&a1, NULL, hello, "Hello ");
  pthread_create(&a2, NULL, hello, "Hello ");
  pthread_create(&b1, NULL, world, "World\n");
  pthread_create(&b2, NULL, world, "World\n");

  pthread_join(a1, NULL);
  pthread_join(b1, NULL);
  pthread_join(a2, NULL);
  pthread_join(b2, NULL);
}
```

CSE 451 Midterm Exam
May 13th, 2009

Your Name: _____

Student ID: _____

General Information:

This is a closed book examination. You have **50 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 **8 pages** on this exam (check to make sure you have all of them), and there is a total of **100 points** in all. Write all of your answers directly on this paper. Make your answers as concise as possible. If there is something in the question that you believe is open to interpretation, then please go ahead and interpret, but state your assumptions in your answer.

Problem 1: (25 points)

Which of the following require assistance from hardware to implement correctly and/or safely? For those that do, circle them, and name the necessary hardware support. For those that don't briefly explain why (one or two sentences at most).

System call: **requires hardware support, in particular to set the CPU's protection bit to run in protected kernel model**.

Process creation: **requires HW support. must do a system call, so need a protection bit. Also, need to manipulate address spaces (i.e., create new page tables for the new process).**

Thread creation: [ok if you argued that kernel threads require system calls, and hence need a protection bit, but user-level threads do not need assistance from HW]

Process context switch: **requires HW support. need a timer interrupt to prevent starvation, and also need to manipulate address spaces (usually by changing a register to point to the page directory for the new process, e.g., cr3 on x86)**

Thread context switch: [ok if you argued that kernel threads require system calls, but user-level threads do not need hw assistance]

Lock acquisition : **requires hw support. need support for atomic instructions (e.g., test and set), or support for disabling/enabling interrupts.**
[note: there are software-only solutions to the critical section problem – see problems 6.9 and 6.10 in the textbook -- and therefore it is possible to use those to implement a (very slow) lock. If you claimed you don't need HW for lock acquisition, and justified this by arguing for a software-only solution to CS, we gave you full credit.]

Lock release: [even if you require HW support to grab a lock, in many cases you don't need it to release the lock. See, for example, how test-and-set based locks works – you don't use the test-and-set to release the lock.]

Problem 2: (25 points)

Three processes P1, P2, and P3 have priorities P1=1, P2=5, P3=10. ("10" is higher priority than "1".) The processes execute the following code:

```
P1:  begin
     <code sequence A>
     lock(X);
     <critical section CS>
     unlock(X);
     <code sequence B>
     end
```

```
P2:  begin
     <code sequence A>
     lock(X);
     lock(Y);
     <critical section CS>
     unlock(Y);
     unlock(X);
     <code sequence B>
     end
```

```
P3:  begin
     <code sequence A>
     lock(Y);
     lock(X);
     <critical section CS>
     unlock(X);
     unlock(Y);
     <code sequence B>
     end
```

The X and Y locks are initialized to "unlocked", i.e., they are free. <*code sequence A*> takes **2** time units to execute, <*code sequence B*> takes **3** time units to execute, and <*critical section CS*> takes **4** time units to execute. Assume `lock()` and `unlock()` are instantaneous, and that context switching is also instantaneous.

P1 begins executing at time **0**, P2 at time **3**, and P3 at time **10**. There is only one CPU shared by all processes.

a) (**9 points)** Assume that the scheduler uses a priority scheduling policy: at any time, the highest priority process that is ready (runnable and not waiting for a lock) will run. If a process with a higher priority than the currently running process becomes ready, preemption will occur and the higher priority process will start running.

Diagram the execution of the three processes over time. Calculate the job throughput and the average turnaround time.

On your diagram, use:
"A" to signify that the process is executing <*code sequence A*>
"B" to signify that the process is executing <*code sequence B*>
"X" to signify that the process holds lock X and is in the critical section
"Y" to signify that the process holds lock "Y" and is in the critical section
"2" to signify that the process holds both locks and is in the critical section,
and leave blank space if the process is not executing (for any reason).

We've started your diagram for you on the next page.

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
P1	A	A	X			X	X	X																	B	B	B													
P2				A	A				2	2			2	2								B	B	B															
P3											A	A			2	2	2	2	B	B	B																			

Job throughput: 3 jobs in 27 time units = 1/9 jobs/time-unit

Average turnaround time: job 1: 27 time unit turnaround.
job 2: 21 time unit turnaround.
job 3: 11 time unit turnaround.

Average: (27 + 21 + 11) / 3 = 59 / 3 = 19 2/3

b) **(9 points)** "Deadlock" is a phenomenon that occurs when a process waits for a resource that will never become available.

Repeat a) with the same assumptions and priorities, but this time assume that:

- P1 begins executing at time 0, P2 begins executing at time 6, and P3 begins executing at time 3.
- P1 has priority 1, P2 has priority 10, and P3 has priority 5.

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
P1	A	A	X			X			X	X	B	B	B																											
P2							A	A																																
P3				A	A																																		

Job throughput: _____0_____

Average turnaround time: _____infinity!_____

c) **(7 points)** Propose and justify a solution to the problem you uncovered in b). Use no more than 3 sentences.

Change P3 to grab/release locks in the same order as P2

Problem 3: (25 points)

(a) **(9 points)** Find all of the logic bugs in the following piece of code. Modify the code (in place, or above the specific lines you'd like to change) to fix the bugs.

```
~~semaphore mutex = 0;~~    // BUG #1
semaphore mutex = 1;
// returns true if the item was added, false otherwise
bool add_item_to_queue(queue_t queue, item_t item) {
  P(mutex);
  if (is_full(queue)) {
     V(mutex);   // BUG #2 (this was missing)
     return false;
  }
  append_to_queue(queue, item);
  V(mutex);
  return true;
~~  V(mutex);~~  // BUG #3 - this should be before the return
}
```

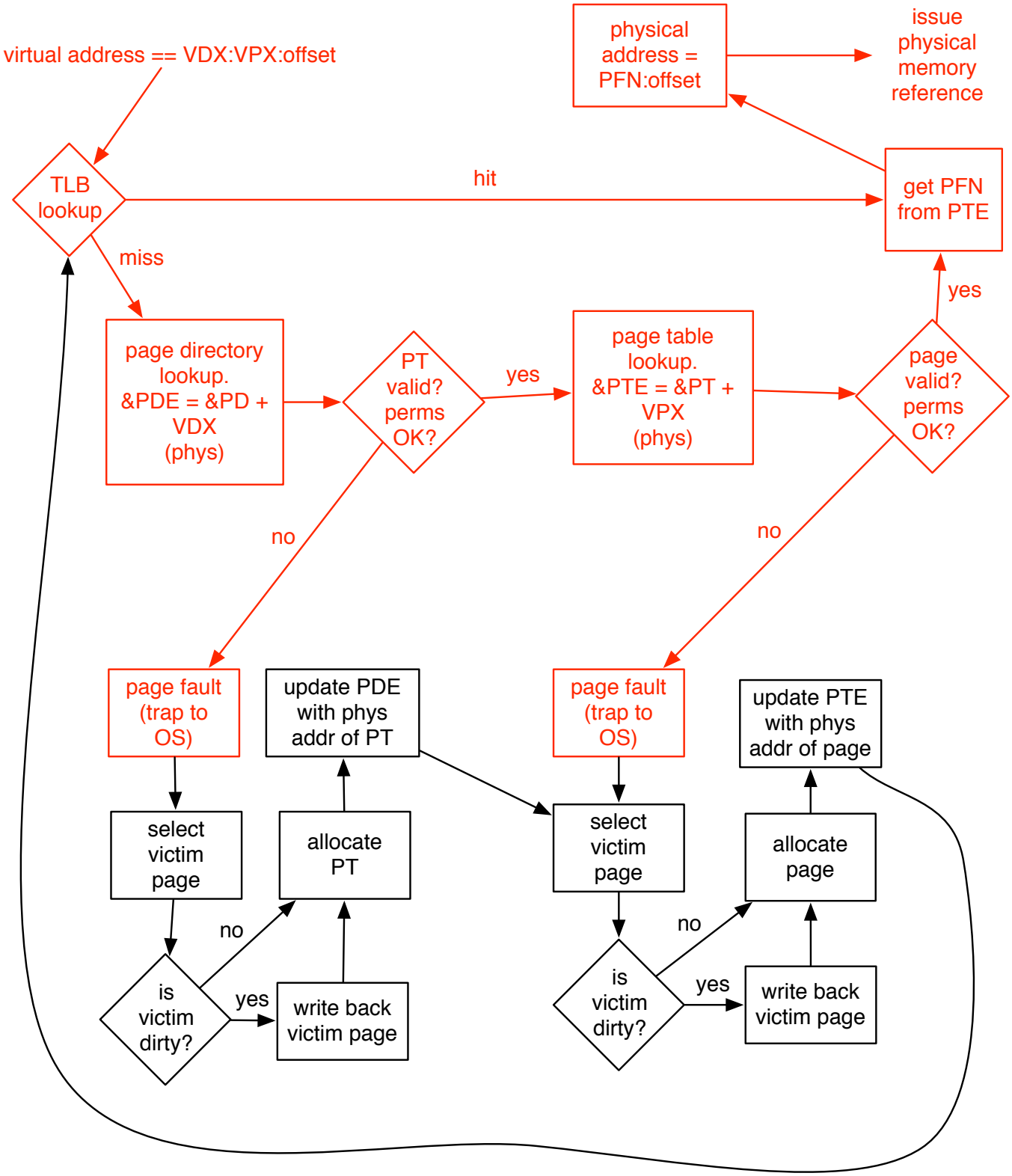
(b) **(16 points)** Find all of the logic bugs in the following piece of code. Modify the code (in place, or above the specific lines you'd like to change) to fix the bugs. If it is helpful, assume the existence of a thread-safe queue class that has add() and remove() functions.

```
~~semaphore thread_available = 0, work_available = 0;~~
semaphore thread_available = NUM_THREADS,
          work_available = 0;

~~functionptr_t dispatch_function;~~
queue dispatch_function_q;

thread_t thread_pool[NUM_THREADS];

void threadpool_start(int my_id);

// PUBLIC FUNCTION; initialize the thread pool
void threadpool_init() {
   for (int i=0; i<NUM_THREADS; i++) {
      thread_pool[i] = thread_create(threadpool_start, i);
   }
}

// PUBLIC FUNCTION; dispatch a worker to run function "f"
void threadpool_dispatch(functionptr_t f) {
  P(thread_available);

  ~~dispatch_function = f;~~
  enqueue(dispatch_function_q, f);

  V(work_available);
}

// PRIVATE FUNCTION: workers wait here until dispatched
void threadpool_start(int my_id) {
   printf("Worker thread %d is alive!\n", my_id);
   while(1) {
     P(work_available);

     x = dequeue(dispatch_function_q);
     x();

     V(thread_available);
   }
}
```

Problem 4: (25 points)

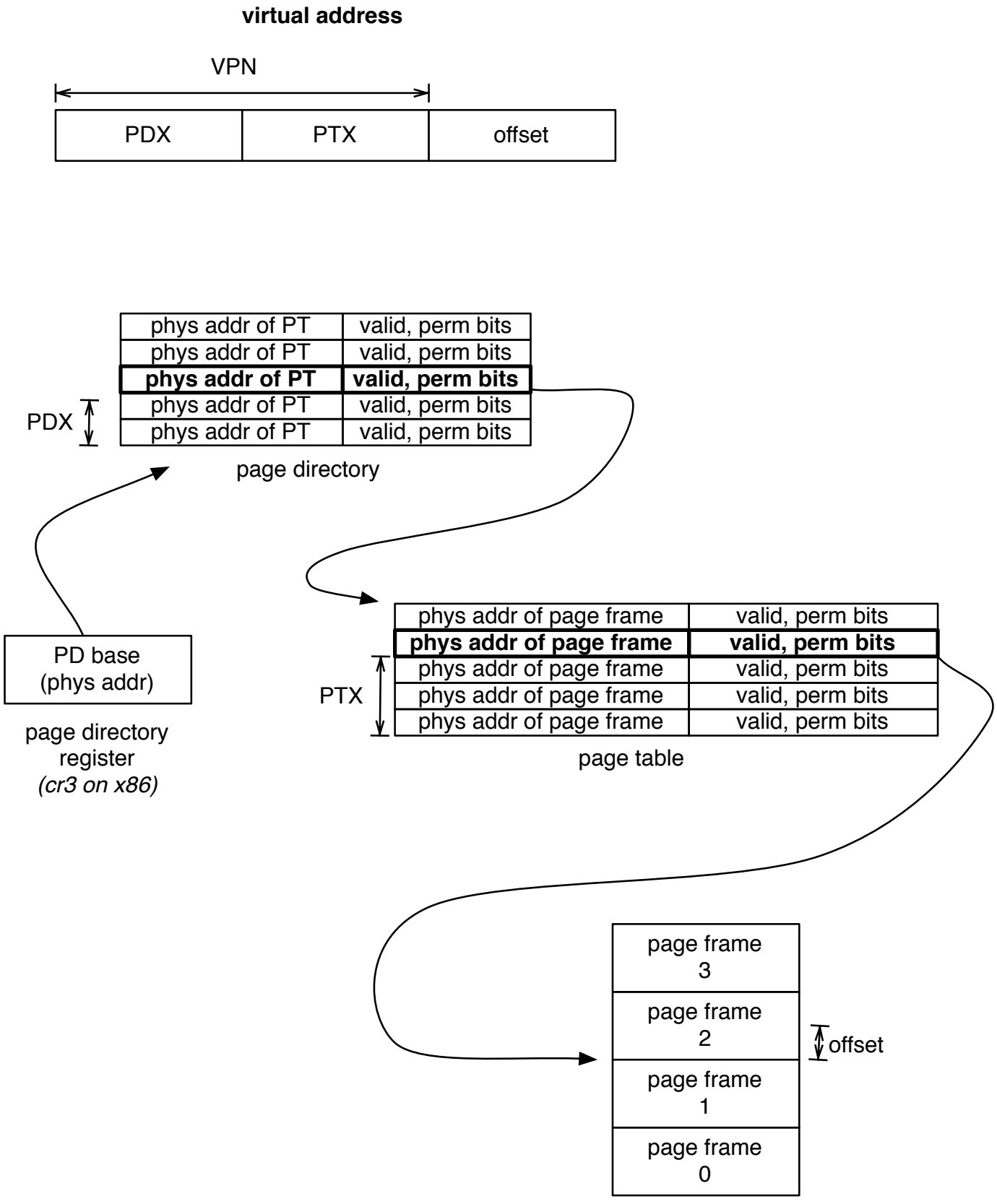
Consider an operating system that uses paging in order to provide virtual memory capabilities; the paging system employs a TLB and a two-level page table. Assume that page tables are "wired" (pinned) in physical memory, and assume that the TLB is hardware-loaded (i.e, the CPU walks page tables to load the TLB on a miss).

Draw a flow chart that describes the logic of handling a memory reference. Your chart must include the possibility of TLB hits and misses, as well as page faults. Be sure to mark which activities are accomplished by hardware, and which are accomplished by the OS's page fault exception handler. Also be sure to clearly identify addresses as either virtual or physical.



RED = HARDWARE

BLACK = OS


virtual address
VPN
PDX
PTX
offset
phys addr of PT
valid, perm bits
phys addr of PT
valid, perm bits
phys addr of PT
valid, perm bits
phys addr of PT
valid, perm bits
phys addr of PT
valid, perm bits
PDX
page directory
PD base
(phys addr)
page directory
register
(cr3 on x86)
phys addr of page frame
valid, perm bits
phys addr of page frame
valid, perm bits
phys addr of page frame
valid, perm bits
phys addr of page frame
valid, perm bits
phys addr of page frame
valid, perm bits
PTX
page table
page frame
3
page frame
2
page frame
1
page frame
0
offset

CSc33200: Operating Systems, CS-CCNY, Fall 2003

Midterm Exam Solutions

November 23, 2003

The highest score is 175 and the average is around 115. Since the scores are significantly lower than previous homework and projects, everyone will receive an increase of 30 points.

1. The Pumpkin Computer uses a segmented addressing scheme in which individual bytes are accessed by combining a 16-bit segment paragraph and a 16-bit relative offset. SR is 16-bit register that points to the beginning of a 16-byte paragraph that is evenly divisible by 16. The segment paragraph is treated as if it were shifted left by four bits. SI is a 16-bit segment index register that contains a relative offset from the segment paragraph specified in SR. What will be the actual memory address accessed if the contents of SR are 1234H and the contents of SI are 4392H?

   Answer:

   Since according to the description, $1234H : 4392H$ is the address of the destination location in main memory based on the segmentation scheme, the actual memory address in the linear address space is:

   $$1234H << 4 + 4392H = 12340H + 4392H = 166D2H$$

2. As a deadlock prevention strategy, the *hold-and-wait* condition may be prevented. If you are allowed to use semaphores, how would you use this strategy to regulate the requests for resources so that deadlock is prevented? Give a skeleton of the program in the same style as the first example regarding the two processes, P and Q, on 10/22's notes.

   Answer:

   To avoid circular wait by preventing the hold-and-wait condition, all resources required by a process may be allocated in the manner of "all at once" or "nothing at all". The scheme may be illustrated as follows:

```
Process P:
  ...
  wait(semaphore)
  Get A
  Get B
  ...
```

```
do_something()
...
Release A
Release B
signal(semaphore)
...
```

Forcing the processes to request for the various resources in a specific order is another way to prevent the hold-and-wait condition.

```
Process P:
  ...
  Get A
  Get B
  ...
  do_something()
  ...
  Release A
  Release B
  ...
```

```
Process Q:
  ...
  Get A
  Get B
  ...
  do_something_else()
  ...
  Release B
  Release A
  ...
```

3. Prove the correctness of Dekker's algorithm in the following aspects:

   (a) Prove that mutual exclusion is enforced.

   Proof 1:

   To show mutual exclusion is enforced in Dekker's algorithm, we prove, based on Figure 5.3 in the textbook, no any other process can get into its critical section after one of the processes is already in its critical section.

   Suppose process P_1 enters its critical section first and remains there. According to the algorithm, $flag[1]$ must be $true$.

   Since P_1 is the only one to get in, another process, say P_0, that wants to get in as well must be executing a statement outside its critical section. Since the $while$ loop is the only control structure that may block its access, we need only to discuss the case when P_0 is executing the $while$ statement. According to $P0()$, when $flag[1]$ is $true$, P_0 won't be permitted to exit the $while$ loop and thus enter the critical section, whatever $turn$ is.

   Hence, at most one process may be in its critical section at a time. Mutual exclusion is enforced indeed in Dekker's algorithm.

   Proof 2:

Suppose two processes, P_0 and P_1, are in their respective critical sections in the purpose of contradiction.

According to Dekker's algorithm in Figure 5.3 in the textbook, we know:

- For P_0, $flag[0]$ is set to $true$ and **then** $flag[1]$ is checked and confirmed to be $false$, which may be denoted by:

```
...
1  flag[0] = true
...
2  flag[1] = false
...
```

- For P_1, $flag[1]$ is set to $true$ and **then** $flag[0]$ is checked and confirmed to be $false$, which may be denoted by:

```
...
3  flag[1] = true
...
4  flag[0] = false
...
```

Suppose P_0 entered the critical section no later than P_1, which means $t_2 \leq t_4$, noting that the confirmation of the falsity of the condition in the outer $while$ loop is the last action a process takes before it enters the critical section.

Then we compare t_2 and t_3. Since once $flag[1]$ is set to $true$ at t_3, it will by no means become $false$, we know $t_2 < t_3$ must hold. We use $<$ here instead of $\leq$ because a location in main memory cannot be accessed simultaneously by multiple processes. Thus we know

$$t_1 < t_2 < t_3 < t_4$$

However once $flag[0]$ is set to $true$ at t_1, it will by no means become $false$ before P_0 finally exits the critical section, noting we suppose that both P_0 and P_1 are in the critical sections at the present time. Thus it is impossible to have checked $flag[0]$ to be $false$ at t_4, which according to our assumption has happened.

Thus a contradiction is drawn. We cannot assume two processes are in the critical sections at any moment, meaning at most one process may access the exclusive resources at one time. So mutual exclusion is enforced in Bekker's algorithm.

(b) Prove that a process requiring access to its critical section will not be delayed indefinitely. That is to show there is no starvation.

Proof:

To show there is no starvation in Dekker's algorithm, we prove, based on Figure 5.3 in the textbook, any process can eventually enter its critical section if it wants.

Suppose process P_1 wants to enter its critical section. Since the $while$ loop is the only control structure that may block its access, we simply suppose it is executing the $while$ statement.

- **If** $flag[0]$ **=** $false$:
  P_1 will surely fail in checking for $while\ (flag[0])$ and thus get into the critical section. One exception is P_1 may not be able to reach $while\ (flag[0])$, due to being trapped at $while\ (turn\ ==\ 0)$. If it is being blocked there, $flag[1]$ has been set to be $false$ to show courtesy. Thus P_1 will in no way present P_0 from accessing the exclusive resources, which will enable P_0 to enter the critical section and finally set $turn$ to 0 and $flag[0]$ to $false$. Thus P_1 may eventually get out of the loop of $while\ (turn\ ==\ 0)$.
  We need not discuss the case when P_0 is not running, because for P_1 to arrive at $while\ (turn\ ==\ 0)$, $flag[0]$ must have once been $true$ and P_0 must have been active, which shows $turn$ and $flag[1]$ have obtained or will have eventually the values giving the green light for P_1 to enter the critical section.
- **Or otherwise** $flag[0]$ **=** $true$:
  Similar to the ending part of the discussion in the first case, $turn$ and $flag[1]$ have been or will be eventually given 1 and $false$ respectively, which guarantees P_1 can exit the $while$ loop at last and get into the critical section.

In either case, P_1 will finally be able to enter its critical section. So due to the equivalence of all the individual processes, any process, if it wants, can eventually visit the exclusive resources. Hence no starvation is in Dekker's algorithm.

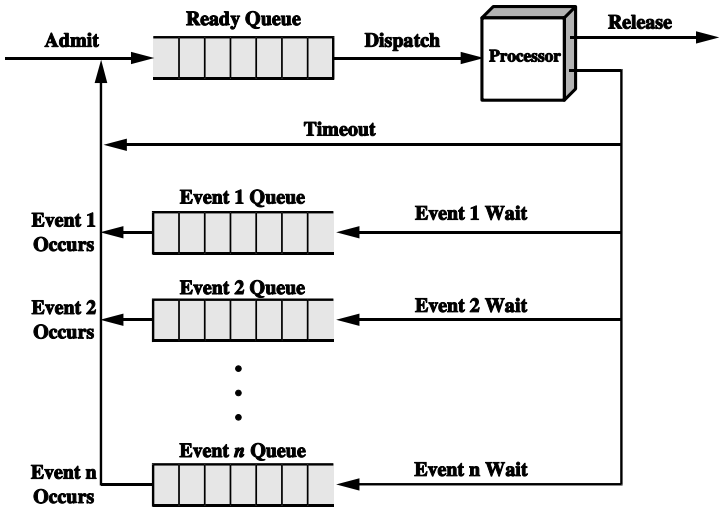
4. The following figure illustrates that a process may be blocked and placed into the corresponding event queue due to waiting for an event of a specific type, but it suggests that a process can only be in one event queue at a time.

   (a) Is it possible that you would want to allow a process to wait on more than one event at the same time? Provide an example.

   Answer:

   Yes, it is possible that a process waits on more than one event at the same time. For example, a process may need to transfer data from one device to another. In this case, it may request both devices at once and wait until both are available for use. Another example is that a network application may wait on multiple sockets until data packets arrive at any of them.

   (b) In that case, how would you modify the queuing structure of the figure to support this new feature? Give the definition of the structures in C/C++ and illustrate it in a picture.



Answer:

Obviously in the event queue model, at least 3 kinds of structures are needed: *process*, *event*, and *process-event pair*, which is defined for event queue nodes.

To enable a process to wait on multiple events at the same time, a chain may be constructed to track all the events that a process is expecting. The head pointer of the chain is stored in `struct process`.

The nodes in event queues thus need additional fields for the event chains they belong to: `struct pe_node *pred, *succ`.

Another issue that needs to be dealt with is that though a process may wait on more than one event at a time, either *all* the events are required for the process to become READY or *any* of them is sufficient to do so. Thus `count` is defined in `struct process`. In the first case, `count` is initialized to be the number of events on which the process is waiting, while in the second case, `count` is set to 1. Whenever an event occurs, `count` is first decreased by 1 and the process-event pair node is removed from both the corresponding event queue and event chain. Then `count` is checked if it becomes 0. If yes, the process is then sent to the READY process queue for dispatching; otherwise continues to wait on the rest of events.

```
struct pe_node {
  // for neighbors in the same event queue
  struct pe_node *next, *prev;

  // for neighbors in the event chain for the same process
  struct pe_node *pred, *succ;

  struct process *p;
  struct event *e;
}
```

```
struct process {
  PCB *pPCB;
  struct pe_node *head;
  int count;
}

struct event {
  char *desc;
  struct pe_node *head;
}
```

5. Do the followings regarding message passing.

   (a) Describe how mutual exclusion and synchronization are supported with message passing.

   Answer:

   (By default, `send` and `receive` primitives that we use for mutual exclusion and synchronization are respectively non-blocking and blocking.)

   To support *mutual exclusion*, a mailbox should be created first and initialized to contain a token message. Each process that is going to access critical resources has to invoke `receive` first to obtain the token before moving on. If permitted, when it exits the critical section, it again sends the token back to the mailbox so that other processes may enter their critical sections later on.

   To support *synchronization* (meaning that a process has to wait until another process is ready for some action), similarly a mail box is needed but is initially empty. The process that is supposed to wait should invoke `receive` and the process that is expected should invoke `send` indicating its availability.

   (b) Give the solution to the producer/consumer problem with an infinite buffer using message passing.

   Answer:

```
producer() {
  while (true) {
    product = produce();
    send(mayconsume, product);
  }
}

consumer() {
  while (true) {
```

```
    receive(mayconsume, product);
    consume(product);
  }
}

main() {
  create_mailbox(mayconsume);
  parbegin(producer, consumer);
}
```

(c) Give a fair solution to the barbershop problem using message passing in a different way from the solution given in the textbook. (*Hint: You may assign a unique number to each barber chair instead of each customer.*)

Answer:

```
void main() {
  create_mailbox(max_capacity);
  for (int i=0; i<20; i++)
    send(max_capacity, null);

  create_mailbox(sofa);
  for (int i=0; i<4; i++)
    send(sofa, null);

  create_mailbox(chair);
  for (int i=0; i<3; i++) {
    send(chair, i);
    create_mailbox(finished[i]);
  }
  create_mailbox(cust_ready);

  create_mailbox(coord);
  for (int i=0; i<3; i++)
    send(coord, null);

  create_mailbox(payment);
  create_mailbox(receipt);
}

void barber() {
  int chair_no;

  receive(cust_ready, chair_no);
  receive(coord, -);
  cut_hair();
  send(coord, null);
  send(finished[chair_no], null);
}
```

```
void customer() {
  int chair_no;

  receive(max_capacity, -);
  enter_shop();
  receive(sofa, -);
  sit_on_sofa();
  receive(chair, chair_no);
  get_up_from_sofa();
  send(sofa, null);
  sit_in_chair(chair_no);
  send(cust_ready, chair_no);
  receive(finished[chair_no], -);
  leave_chair();
  send(chair, chair_no);
  pay();
  send(payment, null);
  receive(receipt, -);
  exit_shop();
  send(max_capacity, null);
}
```

```
void cashier() {
  receive(payment, -);
  receive(coord, -);
  accept_pay();
  send(coord, null);
  send(receipt, null);
}
```

In the above solution, the hair-cut sessions are differentiated by the *chair NOs* instead of *customer IDs* used in the textbook. And with message passing, the interaction between a barber and a customer is always associated with the corresponding chair NO, which is transferred as a message between them.

6. Use semaphores to solve the following problem:

   You have been hired by Greenpeace Organization to help the environment. Because unscrupulous commercial interests have dangerously lowered the whale population, whales are having synchronization problems in finding a mate. The trick is that in order to have children, `three` whales are needed, one male, one female, and one to play matchmaker – literally, to push the other two whales together (I'm not making this up!). Your job is to write the three procedures `Male()`, `Female()`, and `Matchmaker()`. Each whale is represented by a separate process. A male whale calls `Male()`, which waits until there is a waiting female and matchmaker; similarly, a female whale must wait until a male whale and a matchmaker are present. Once all three are present, all three return.

Answer:

```
semaphore male = 0, female = 0;
semaphore male_start = 0, male_end = 0;
semaphore female_start = 0, female_end = 0;

Male() {
  signal(male);
  wait(male_start);
  ...
  wait(male_end);
}

Female() {
  signal(female);
  wait(female_start);
  ...
  wait(female_end);
}
```

```
Matchmaker() {
  wait(male);
  wait(female);
  signal(male_start);
  signal(female_start);
  match();
  signal(male_end);
  signal(female_end);
}
```

Note that this is an unfair solution, in which a matchmaker may send the `male_start` / `female_start` signal to another male/female whale rather than the one that has been allocated to it after `wait(male)` / `wait(female)`. The same problem happens when a `male_end` / `female_end` signal is sent out. These problems may be solved in the same way as the barbershop example in the text by using queues or the solution we give above in Question 5(c).

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

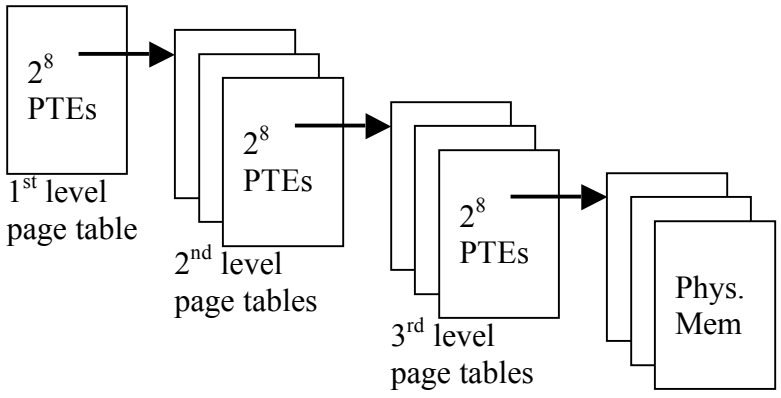
$H = 199/200$

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

    return;
}
```

```
oReady()
{
    P(o_wait);
    V(h_wait);
    V(h_wait);
    makeWater();

    return;
}
```

Alternative equivalent solution:

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

hReady() {
    V(o_wait)
    P(h_wait)

    return;
}
```

```
oReady()
{
    P(mutex)
    P(o_wait);
    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.

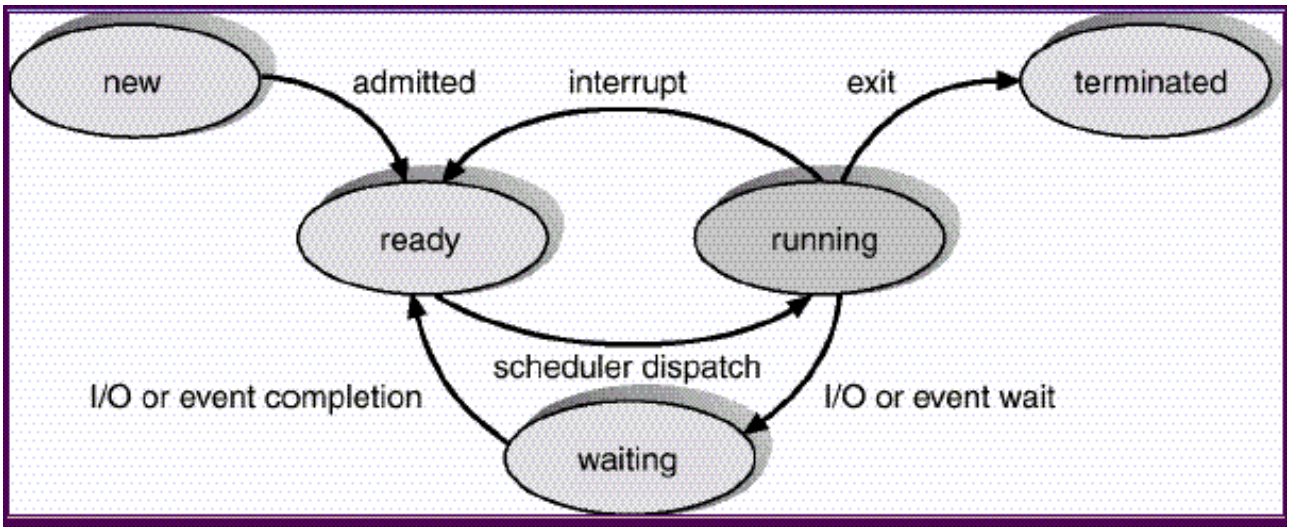
No credit or almost no credit if did not present a viable solution.

Midterm Exam (50 points)
CSCI 4210H01 Operating Systems Name:_____
Semester: Fall 2001
Instructor: Gerard Weatherby

The number of points follows each question.Write answers in the space provided; use the backs of pages if you need more space. **Be sure to indicate where the answer is** if you use the backs of pages. If you use an acronym in your answers, be sure to expand it upon first use. e.g. "The best graduate school is clearly RH (Rensselaer at Hartford)."

1. (2) List two operations which are permitted while the CPU is in monitor/executive mode but not permitted while in user mode.
change interrupt vector
change process (memory) limits
change mode bit
access device
et. al.
Note that saying "privileged instructions" and some else isn't sufficient, by definition everything that can only be done in monitor mode is a privileged instruction.

2. (10) Draw the life cycle of a process. Include the **states** a process may be in and the **transitions** from one state to another, including what causes a process to make a transition.



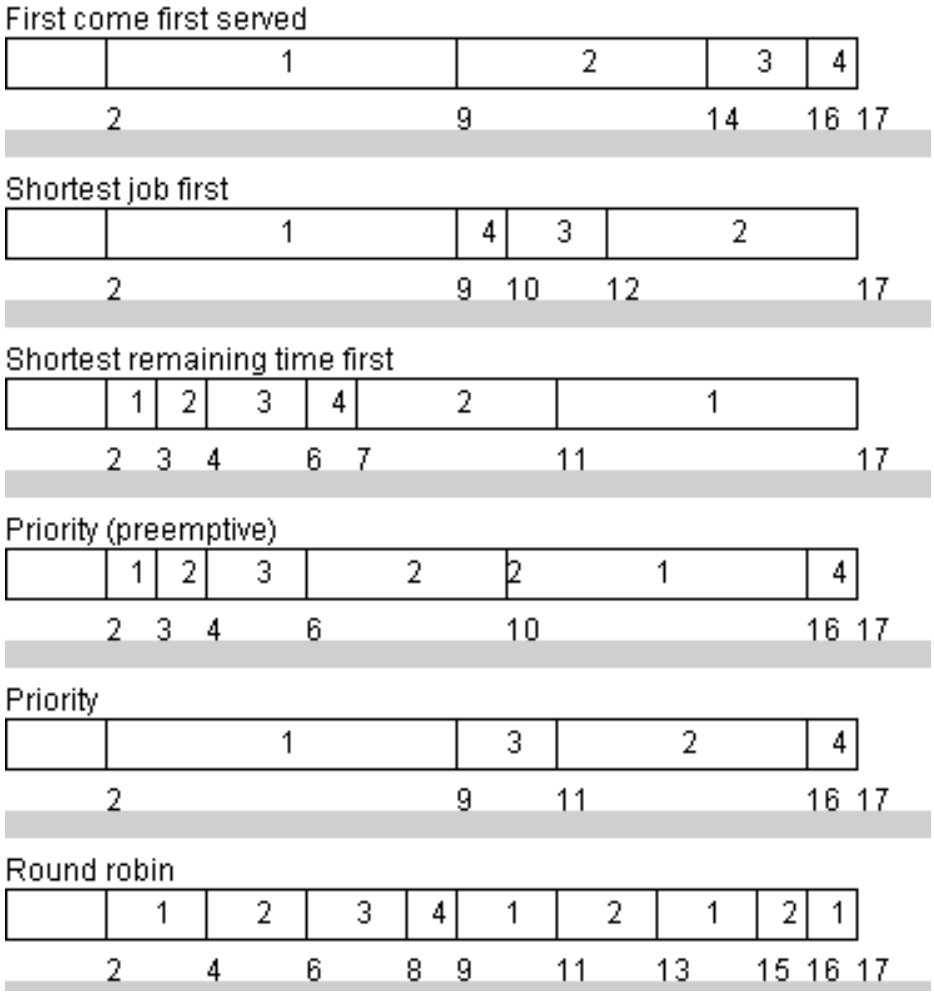
3. (10) The following processes are submitted to an operating system. They are calculation processes which require no input/ output (i.e. they are always ready to utilize the CPU)

a.List and describe 5 different scheduling methods

b.Show in a diagram form (Gantt chart) how the methods you describe would schedule the processes. Assume a time quantum of 2, where applicable. A lower number indicates a greater priority.

Process	Arrival time	Burst time	Priority
1	2	7	9
2	3	5	7
3	4	2	3
4	6	1	10

1 per point description, 1 per Gannt chart
First Come, First Served -- processes served in order of arrival
Shortest Job First - process with shortest burst time scheduled next, non preemptive
Shortest Remaining Time First - processess with shortest burst time remaining run first, preemptive
riority - higher priorites run first (premptive)
Round-robin -- each process gets time quantum (3) in turn
Priority - higher priorites run first (non-premptive)



4. (3) Describe how a PC boots up from power on until the operating system is accepting commands (e.g. awaiting a user to login)

BIOS (basic input output system)conducts power on self test
BIOS loads operating system from specified location at disk drive
operating system installs own routines in interrupt jump vector
operating system starts "startup" processes

5. (7) Which of the following would be stored in a Process Control Block (check those that would be)? Check the underline in front of the item.

___ mode bit
___ track cylinder and sector of first program block on disk
___ processor speed (in MHz)
✓ *register values*
___ local variables of current function
✓ *process counter*
✓ *memory limits*
✓ *list of open files*
___ IP address of machine
___ list of functions in progress (call stack)
___ file permissions of executable(UNIX) or *.exe (Windows) file
___ interrupt vector
___ memory allocated by the new operator
___ stack for each thread
✓ *status word*

1/2 each, rounded up

6. (2) Label the following illustration with the following: track, sector, cylinder, platter, read/write head.

(C) 2000 Western Digital. Permission pending.

7. (2) A process has the following history of bursts. Given the following α's, what is the estimate for the next burst? (higher α weighs more recent history more heavily.) Answers must be accurate within 0.5 time units.

Initial	***History***	***History***	***History***	α	***Estimate***
5	*6*	*9*	*2*	*1.0000*	***2***
6	*6*	*6*	*6*	*0.3425*	***6***
5	*6*	*9*	*2*	*0.0000*	***5***
2	*27*	*18*	*30*	*0.6667*	***26.074 (25.5 - 26.5) accepted***

$$\frac{2}{3} \bullet 30 + \frac{1}{3}\left(\frac{2}{3} \bullet 18 + \frac{1}{3}\left(\frac{2}{3} \bullet 27 + \frac{1}{3} \bullet 2\right)\right) = 26.074$$

8. (11) Given the following code:

```
struct Vehicle {
    enum Env {air,land,sea};
    virtual std::string name( ) const;
    virtual ~Vehicle( );
    virtual Env environment( )=0;
};
struct Boat: public Vehicle {
    virtual Env environment( )
        { return sea; }
};

struct Truck: public Vehicle {
    virtual std::string name( ) const
        { return "truck"; }
};
```

and the following declarations:

```
    Vehicle *pv;
    const Vehicle *cpv;
    Boat *pb;
    Truck *pt;
    using std::cout;
```

Indicate whether the following lines are valid or invalid. If invalid, indicate reason why. Consider only whether the lines will compile, not their runtime execution. Assume any necessary header files are included. Each line is to be considered separately.

Line	***Valid***	***Invalid***	***Reason***
pv = new Vehicle( );		*x*	*Vehicle abstract (has pure virtual function)*
pv = new Boat( );	*x*		
pv = new Truck( );		*x*	*Vehicle abstract (has pure virtual function)*
pv = pb;	*x*		
pb = pv;		*x*	*Not all vehicles are boats (assigning base class to derived class)*
pt = pb;		*x*	*assigning across class hierachy*
*pb = dynamic_cast<Boat *>pb;*	*x*		
cout << cpv.name( );		*x*	*cpv is pointer, requires ->*
cout << cpv->environment( );		*x*	*cpv is const pointer but environment is not const function*

Number wrong	*Points off*
1	*1*
2	*1*
3	*2*
4	*3*
5	*4*
6	*4*
7	*5*
8	*6*
9	*7*
10	*7*
11	*8*

9. (3) What's the output of the following code?

```
#include <iostream.h>
void f(int x) {
    x += 2; }
void g(int &a,int div=2) {
    a /= div; }
void g(double &d) {
    d = 4; }
void h(int *p) {
    *p = 20; }
int main( ) {
    int x = 3;
    int y = 4;
    int z = 5;
    h(&x);
    cout << x << endl;
    g(y);
    cout << y << endl;
    f(z);
    cout << z << endl;}
```

output:
20
2 (4 accepted)
5

Midterm Exam (50 points)
CSCI 4210H01 Operating Systems Name:_____
Semester: Spring 2005
Instructor: Gerard Weatherby

The number of points follows each question. The size of the space provided is **not** an indication of the expected length of the answer.Write answers in the space provided; use the backs of pages if you need more space. **Be sure to indicate where the answer is** if you use the backs of pages. If you use an acronym in your answers, be sure to expand it upon first use. e.g. "The best graduate school is clearly RH (Rensselaer at Hartford)."

1. (3) What is the purpose of a process control block? When it is updated and when it is read by the operating system?
Keeps track of the state of the CPU (register variables) and resources for a given process.
It is updated when a process is preempted (switched from).
It is read when a process is restored (switched to).

2. (1) What's the difference between a process starting another copy of itself and starting another thread?
A thread has access to the other memory in the same process.
Processes have their own distinct memory.

3. _____ (1) DMA is
   a. Direct Manipulation Algorithm
   b. Direct Memory Access
   c. Dynamic Memory Allocation
   d. Data Memory Allocation.

4. (2) Round robin scheduling requires a time quantum (q). What is the effect (i.e. what bad thing happens) if q is too large? What is the effect (i.e. what bad thing happens) if q is too small?
If too large scheduling is same as first come first served.
If too small too much time (overhead) is spent switching processes.

5. (4) Describe the difference between the *wait* and *signal* operations of a Semaphore and a condition variable.
For a semaphore, wait checks the value and blocks if it is zero or less. It decrements the value by one when it proceeds.
A signal unconditionally increments the value by one.
For a condition variable in a monitor, wait unconditionally blocks the calling thread until awoken by another thread.
Signal wakes up one blocked thread, if any. Otherwise it does nothing.

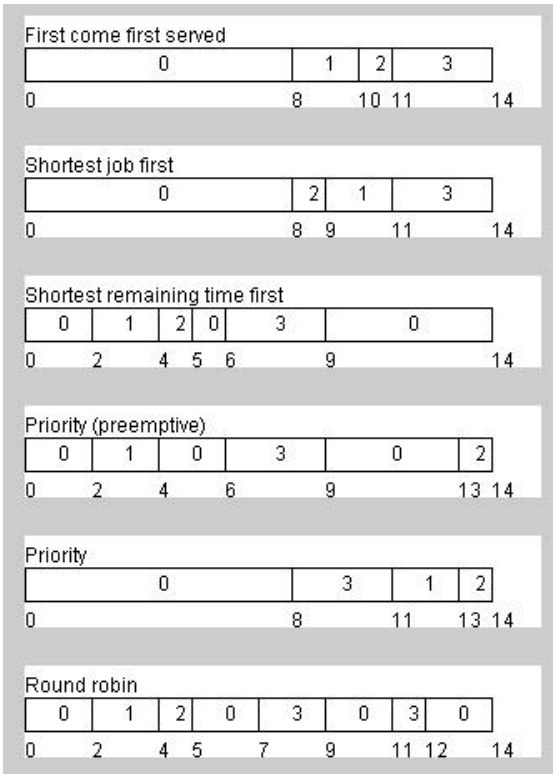
6. (10) The following processes are submitted to an operating system. They are calculation processes which require no input/ output (i.e. they are always ready to utilize the CPU)

a. List and describe 5 different scheduling methods

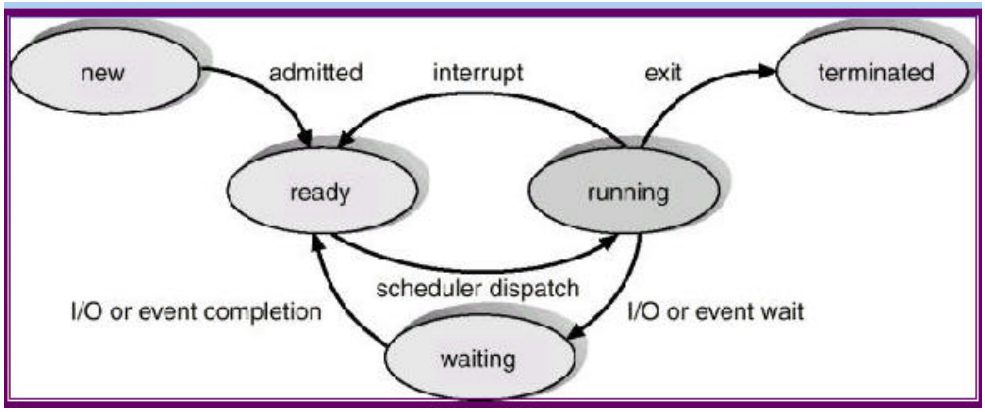
b. Show in a diagram form (Gantt chart) how the methods you describe would schedule the processes. Assume a time quantum of 2, where applicable. A **lower** number indicates a greater priority

Process	Arrival time	Burst time	Priority
0	0	8	5
1	2	2	4
2	4	1	6
3	6	3	3

FCFS - first come first served. Processes dispatched in order of arrival
SJF - shortest job first. Process with shortest burst time runs next. Non-preemptive
SRTF - shortest remaining time first. Process with shortest burst time runs, pre-emptive.
Priority - highest priority runs first Non-preemptive
Priority - highest priority runs first Preemptive



7. (11) Draw a diagram of the possible process states. Include the event(s) which cause a transition from state to another.



8. (1) An interrupt is:

a. A temporary suspension of the PC clock signal to allow device controllers to access memory.
b. A suspension of code executing in a critical section.
c. A signal that causes the control unit to branch to a specific location.
d. The exiting of a loop due to a break statement..

9. (1) What's a privileged instruction?

An instruction which can only be executed when the CPU is in monitor (executive) mode.

10. (6) Briefly describe the difference between *C++ for, do,* and *while* loops.

for generally is used to count a fixed number of iterations; it has initialization, test and increment clauses
a do loop repeats while a condition is true -- the condition is checked after the first iteration
a while loop repeats while a condition is true -- the condition is checked before the first iteration

11. (2) What is a C++ destructor and when is it executed?

set of code used to release resources or notify other objects an object is being destroyed
It is executed whenever an object of the class type is destroyed.

```
class Semaphore {
    public:
        Semaphore(int initial = 0);
        ~Semaphore( );
        void wait( );
        void signal( );
        class Guard {
            public:
                Guard(Semaphore &s);
                ~Guard( );
            };
};
```

```
class Document {
    public:
        static Document & theDocument( ):
        void read( );
        void write( );
};
```

12. (8) Using the definition of Semaphore and Document classes given, show a C++ implementation of the Reader and Writer of the readers/writers problem such that:

- Multiple readers may access the document at the same time
- Only one writer may access the document at a time.
- Writers and readers can **not** access the document at the same time.

```
Semaphore writelock(1);
Semaphore mutex(1);
int readcount = 0;

Writer::write( )
{
    writelock.wait( );
    Document::theDocument( ).write( );
    writelock.signal( );
}


Reader::read( )
{
    mutex.wait( );
    if (++readcount == 1)
        writelock.wait( );
    mutex.signal( );
    Document::theDocument( ).read( );
    mutex.wait( );
    if (--readCount == 0)
        writelock.signal( );
    mutex.signal( );
}
```

University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2012 Anthony D. Joseph and Ion Stoica

Midterm Exam *Solutions*

March 7, 2012
CS162 Operating Systems

Your Name:	
SID AND 162 Login:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book and one 2-sided handwritten note** examination. You have 80 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question. 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 there is something in a question that you believe is open to interpretation, then please ask us about it!

Good Luck!!

QUESTION	POINTS ASSIGNED	POINTS OBTAINED
1	20	
2	20	
3	15	
4	20	
5	15	
6	10	
TOTAL	100	

1. (20 points total) Short answer questions.
   a. (8 points) True/False and Why? **CIRCLE YOUR ANSWER.**
      i) A lightweight process with one thread is equivalent to a heavyweight process.

TRUE FALSE

Why?

***TRUE**. A heavyweight process has only one thread. The correct answer was worth 2 points and the justification was worth an additional 2 points.*

      ii) Demand paging requires the programmer to take specific action to force the operating system to load a particular virtual memory page.

TRUE FALSE

Why?

***FALSE**. The OS automatically loads pages from disk when necessary. The correct answer was worth 2 points and the justification was worth an additional 2 points.*

   b. (8 points) Two-level Page Tables:
      i) Give a two to three sentence description of a two-level page table.

A two-level page table uses two levels of page tables where a pagetable pointer points to the top-level table. Entries in the top-level table point to the lower-level page tables. The lower level tables contain PTEs pointing to the physical locations.

      ii) Briefly (2 sentences) state one advantage AND one disadvantage of two-level page tables.

Advantage: Can map sparse address spaces efficiently
Disadvantage: Requires an additional memory reference to translate virtual to physical addresses.

c. (4 points) List the four requirements for deadlock.

Mutual exclusion, non-preemptable resources, hold and wait, circular chain of waiting.

Each requirement was worth 1 points.

2. (20 points total) Consider the following two functions implementing a producer and consumer by using monitors:

```
void send(item) {
  lock.acquire()
  enqueue(item);
  printf("before signal()\n");
  dataready.signal(&lock);
  printf("after signal()\n");
  lock.release();
}

item = get() {
  lock.acquire();
  while (queue.isEmpty()) {
    printf("before wait()\n");
    dataready.wait(&lock);
    printf("after wait()\n");
  }
  item = dequeue();
  lock.release();
}
```

a. (4 points) Use no more than three sentences to contrast Hoare and Mesa monitors.

With Hoare the signaler gives the CPU and the lock to the waiter; With Mesa the signaler schedules the waiter, and then finishes.

- *-1 if it was not clear that a thread in a Mesa monitor can hold the lock indefinitely after signaling.*
- *-1 if the signaled/woken thread is put on a wait queue instead of the ready queue.*

b. (5 points) Assume two threads T1 and T2, as follows:

```
     T1                                    T2
send(item);                          item = get();
```

What are the possible outputs if the monitor uses the Hoare implementation?

```
before signal
after_signal
```

- *We gave 2 points for the above solution.*

```
before wait
before signal
after wait
after signal
```

- *We gave 3 points for the above solution.*
- *-2 if there were more than two answers, but at least 1 point if "something" was right.*

c. (5 points) Repeat question (b) for a Mesa implementation of the monitor.

```
before signal
after_signal
```

- *We gave 2 points for the above.*

```
before wait
before signal
after signal
 after wait
```

- *We gave 3 points for the above.*
- *-2 if there were more than two answers, but at least 1 point if "something" was right.*

d. (6 points) Now assume a third thread T3, i.e.,

```
   T1                  T2                  T3
send(item);        item = get();       send(item);
```

What are the possible outputs if the monitor uses the Hoare implementation? Please specify from which thread does an output come by specifying the thread id in front of the output line, e.g., `[T1] before signal` or `[T2] after wait`.

```
[T1] before signal          [T3] before signal
[T1] after signal           [T3] after signal
[T3] before signal          [T1] before signal
[T3] after signal           [T1] after signal
```

- *1 point for each of the above ones.*

```
[T2] before wait            [T2] before wait
[T1] before signal          [T3] before signal
[T2] after wait             [T2] after wait
[T1] after signal           [T3] after signal
[T3] before signal          [T1] before signal
[T3] after signal           [T1] after signal
```

- *2 point for each of the above ones.*
- *--1 for each answer beyond four.*

3. (15 points) Design tradeoffs (15 points total):
You've been hired by Orange Computer to help design a new processor and Orange Pro laptop. After choosing the display, case, and other components, you are left with $460 to spend on the following components:

Item	Latency	Minimum Size	Cost
TLB	10 ns	256 entries	$0.10/entry
Main memory	180 ns	2 GB	$10/GB
Magnetic Disk	8 ms (8M ns)	300 GB	$0.10/GB

The page size is fixed at 64 KB. Assume you want to run up to 20 applications simultaneously. Each application has an overall maximum size of 1 GB and a working set size of 256 MB. TLB entries do not have Process Identifiers. Discuss how you would divide the available funds across the various items to optimize performance.

We start with the disk. Since the disk is the slowest component of the system, we take the minimum size, 300 GB or $30, leaving us with $430. Since the TLB does not contain process identifiers, we only need the minimum number of entries to map the working set for a single process – 256 MB / 64 KB = 4,096 entries or $409.60, leaving us with $21.40. With the remaining money, we could buy 2 GB. However, since the maximum number of applications is 20 each with a working set size of 256 MB, we should provide 5 GB of RAM ($50) to avoid paging, so we should spend $380 on 3,800 entries. While this will cause TLB misses, it does not make sense to increase the TLB any more, since that would require that we decrease the memory size below the requirements for the applications; a situation that will cause the system to start paging.

We awarded 3 points per choice, based upon the reasonableness of the choices. We used the following table:

Item / Points	3 points	2 points	1 point	0 points
(a) TLB	3,800 entries	> 3,800	< 3,800	< 256
(b) Memory	5 GB	> 5 GB	< 5 GB	< 2 GB
(c) Disk	300 GB	> 300GB, if using extra money for disk instead of memory or TLB	> 300GB	< 300GB

(d) We awarded another 3 points if the TLB answer was based upon an analysis of a single applications' working set size (i.e., since only mapping the working set matters and the TLB does not include process identifiers).
(e) We awarded an additional three points based upon the overall reasonableness of the answer. For example, a system with a small amount of paging would lose a point, while a system with a significant amount of paging (e.g., only 2 GB of memory), would lose two points.

4. (20 points) Concurrency control: Consider the following pseudocode that aims to implement a solution for the Dining Philosopher problem. Note that a philosopher can use any chopstick.

```
// assume chopstick[i].status = FREE, for 1 <= i <= N
get_chopstick(boolean hold_one_chopstick) {
  lock.acquire();
  for (i = 1; i <= N; i++) {
    if (chopstick[i].status == FREE) {
      chopstick[i].status = BUSY;
      return i;
    }
  }
  lock.release();
  return -1;
}

release_chopstick(i) {
  if (i == -1) return;
  chopstick[i].status = FREE;
}


philosopher() {
  plate = FULL;
  while (plate == FULL) {
    chopstick1 = get_chopstick(FALSE);
    if (chopstick1 != -1) {
      chopstick2 = get_chopstick(TRUE);
      plate = EMPTY;
      release_chopstick(chopstick2);
    }
    release_chopstick(chopstick1);
  }
}

main() {
  for (i = 1; i <= N; i++) {
    thread_fork(philosopher());
  }
}
```

a. (2 points) Name an error in how synchronization primitives are used in `get_chopstick()`

There is a missing `lock.release()` *before* `return i`.

b. (10 points) After fixing the error in part (a), does the program work correctly? If it does not, give a simple example to show how the program fails, and provide a fix. If it does, use no more than three sentences to argue why it works.

Is not guaranteed to work. Every philosopher can get a chopstick, fail to get the second one, release the chopstick they hold, and repeat!

- *We gave 5 points for an example.*

Add code to `get_chopstick()` *to not give the last chopstick to a philosopher if that philosopher doesn't already own a chopstick. For example:*

```
cnt = 0;
for (i = 1; i < N; i++) {
  if (chopstick[i].status == FREE) {
    cnt++;
  }
}
if (cnt == 1 && hold_one_chopstick == FALSE) {
  return -1;
}
```

- *We subtracted 3 points if you did not give the above solution.*
- *We subtracted 1 point if you said it in words, but not give the code.*
- *We subtracted 1 point if your solution was to use mutex around the entire body of* `philosopher()`*, as we considered this solution to do "excessive locking".*

In addition, you need to check for `chopstick2` *in* `philosopher()`*, i.e.,*
`if (chopstick2 != -1) { plate = EMPTY; }`

- *We subtracted 2 point if you did not provide the above fix.*

(We also need to protect the body of release_chopstick(), by lock.acquire() and lock.release(). However, we did not subtract any points for this.)

c. (8 points) Assume `main()` launches N+1 philosopher threads, instead of N. Will the program work correctly given the changes you made for parts (a) and (b)? If it does not, give a simple example to show how the program fails, and provide a fix. If it does, use no more than three sentences to argue why it works.

No change needed, as the modified code in (b) will guarantee that the last chopstick will always be picked by a philosopher that already has another chopstick.

5. (15 points total) Scheduling.
   a. (15 points) Consider the following processes, arrival times, and CPU processing requirements:

Process Name	Arrival Time	Processing Time
1	0	3
2	1	5
3	3	2
4	9	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). For RR and SRTF, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it. The turnaround time is defined as the time a process takes to complete after it arrives.

Time	**FIFO**	**RR**	**SRTF**
0	**1**	**1**	**1**
1	*1*	*2*	*1*
2	*1*	*1*	*1*
3	*2*	*3*	*3*
4	*2*	*2*	*3*
5	*2*	*1*	*2*
6	*2*	*3*	*2*
7	*2*	*2*	*2*
8	*3*	*2*	*2*
9	*3*	*4*	*2*
10	*4*	*2*	*4*
11	*4*	*4*	*4*
Average Turnaround Time	*3+7+7+3/ 4 =* ***5***	*6+10+4+3/ =* ***5.75***	*3+2+9+3/4=* ***4.25***

Each column is worth 5 points: 3 for correctness of the schedule (we deducted 1/2/3 points if you made minor/intermediate/major mistakes), and 2 for the average Turnaround time (1 point was deducted for minor errors).

6. (10 points total) Caching: Assume a computer system employing a cache, where the access time to the main memory is 100 ns, and the access time to the cache is 20ns.

a. (2 points) Assume the cache hit rate is 95%. What is the average access time?

*Average Access Time = Hit*cache_access_time + (1-Hit)*memory_access_time*
*= 0.95*20 ns + 0.05*100 ns = 24 ns*
*Alternatively, we accepted solutions that included the cache time in the memory access time: AAT = 0.95 * 20 ns + 0.05 * (20 ns + 100 ns) = 25 ns.*
We subtracted one point for minor errors.

b. (2 points) Assume the system implements virtual memory using a two-level page table with no TLB, and assume the CPU loads a word X from main memory. Assume the cache hit rate for the page entries as well as for the data in memory is 95%. What is the average time it takes to load X?

*The Average Memory Access Time for X (AMAT) requires three memory accesses, two for each page entry, and one for reading X: 3*24 = 72 ns. The alternate solution from (a) yields 3*25 = 75 ns. We only accepted the alternate solution for (b) if you derived the same value for (a).*

c. (3 points) Assume the same setting as in point (b), but now assume that page translation is cached in the TLB (the TLB hit rate is 98%), and the access time to the TLB is 16 ns. What is the average access time to X?

*AAT_X is TLB_hit *(TLB_access_time + AAT) + (1-TLB_hit) * (3 * AAT):*
0.98(16ns + 24ns) + 0.02*(72ns) = 0.98*40ns + 1.44ns = 40.64ns*
Alternate AAT from (a): 0.98(16ns + 25ns) + 0.02*(75ns) = 41.68ns*

It was acceptable to include the TLB time in the TLB_miss calculation:
*TLB_hit *(TLB _time + AMAT) + (1-TLB_hit) * (3 * AMAT +TLB_time).*
0.98(16ns + 24ns) + 0.02*(72ns + 16ns) = 0.98*40ns + 0.02*88ns = 40.96ns*
Alternate AAT from (a): 0.98(16ns + 25ns) + 0.02*(75ns +16ns) = 42ns*
We subtracted one point for each minor error.

d. (3 points) Assume we increase the cache size. Is it possible that this increase to lead to a decrease in the cache hit rate? Use no more than three sentences to explain your answer.

Yes, using a FIFO replacement scheme could result in Belady's anomaly. Also, using the same hash function while increasing the cache size could cause more collisions and reduce the hit rate. The correct answer was worth one point and the justification was worth two points.

This page intentionally left blank

Do not write answers on this page

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2003

Quiz II Solutions

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to answer this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

1 (xx/15)	2 (xx/25)	3 (xx/20)	4 (xx/25)	5 (xx/15)	Total (xx/100)

Name:

I Control-C and the shell

1. [15 points]: In UNIX when a user is running a program from the shell, the user can terminate the program by typing ctrl-C. How would you change the 6.828 kernel and its shell to support this feature? Be careful, make sure when a user types the name of the program and hits ctrl-C before the program runs that the right thing happens (i.e., don't kill the shell itself or the file system). (Sketch an implementation and define "what the right thing" is.)

There are many possible solutions; here are three:

- Add a system call to the kernel that allows the shell to indicate at any given point what environment, if any, should be killed when the user types ctrl-C. Modify the console code in the kernel appropriately so that the kernel intercepts ctrl-C (instead of passing it on as a control character to whatever environment is reading the console), and kills the indicated environment. With this solution, environments can never prevent themselves from getting killed: thus, CTRL-C acts like a "hard" or "uncatchable" Unix signal.
- Modify the kernel's console code so that the shell can intercept ctrl-C input characters even when other, "normal" input characters may be going to some other (e.g., child) environment that happens to be reading from the console. Then upon receiving this special control character from the kernel in the console input stream, the shell can decide which of its child environments should be killed. With this solution, environments once again cannot prevent themselves from getting killed, but only the *immediate* child environments of the shell can be killed this way: a (killable) child environment can easily create another, "unkillable" environment simply by forking or spawning a child of its own (a grandchild of the shell).
- Add some form of asynchronous event handling support to the kernel's console code so that environments can asynchronously accept and handle ctrl-C input characters themselves. For example, each environment might be able to register a user-mode "ctrl-C handler," along the lines of the current user-mode "page fault handler." Whenever the user hits ctrl-C, the kernel scans through the list of environments and, for each environment that has registered a ctrl-C handler (and perhaps has set a flag indicating that ctrl-C "signals" should be accepted), the kernel sets up the environment's registers and user-mode exception stack to cause the environment's registered ctrl-C handler to be invoked. Each environment can then perform whatever ctrl-C handling it deems appropriate, such as by killing itself immediately, by killing itself after "gracefully" shutting down (such as after closing files and `sync`ing outstanding data to disk), or even by ignoring the ctrl-C signal entirely. With this solution, CTRL-C is effectively a "soft" signal, which is not guaranteed to work on environments that may be either accidentally or maliciously ignoring it. (This is like the Unix behavior for ctrl-C, since Unix processes can intercept the `SIGINT` signal.)

II CPU scheduling

2. [10 points]: `primespipe` spawns many environments until it has generated a prime greater than some specified number. If you modify your shell for the 6.828 kernel to run `primespipe` in the background and then type `ls` at your shell, it can happen that the output of `ls` won't show before `primespipe` completes. Explain why.

(Keep it brief)

The kernel's scheduler uses a round-robin scheduling policy, always choosing the next runnable environment *in environment ID-space* whenever the current environment yields or runs out of its timeslice. Since newly created environments are assigned environment IDs in ascending sequence, `primespipe` creates a pathological unfairness situation. The "root" `primespipe` environment creates the first child environment, fills the pipe to that child with numbers, then yields the processor in order to wait for more space in the pipe. Since the first child environment typically has the environment ID immediately following the parent's, the child is the next to run. The child in turn creates its own grandchild environment, then reads numbers from its parent's pipe buffer and stuffs the ones that "pass" into the grandchild's pipe buffer. Upon running out of numbers, the child yields, causing the grandchild to run next, and so on. Even if the shell managed to start the `ls` environment before the `primespipe` cascade takes over, `ls` requires a few back-and-forth IPCs with the file system, and so it is very unlikely to get much of anything done before the entire `primespipe` cascade has completed. The key point is that this unfairness is created by the interaction of the *round-robin scheduling policy* with the kernel's "first-free" environment ID assignment.

3. [5 points]: Describe a solution to the problem described in the previous question.

(Keep it brief; no pseudocode required)

- A quick-and-dirty solution that would make this unfairness problem go away but would probably leave many others: Simply make the round-robin scheduling policy look for runnable environments in *descending* rather than ascending order of environment ID. Then `ls` and the file server will each typically get one chance to run for each new `primespipe` instance.
- A more general solution: maintain an explicit *queue of runnable environments* from which the scheduler picks the next environment to run instead of just scanning by environment ID. As long as newly-runnable (including newly-created) environments are always added to the *tail* of the queue and the scheduler always picks the environment at the *head* as the next environment to run, environment creation cannot be used as a way to hog the CPU.
- Implement a Unix-style interactive scheduling policy in which processes that have not run for a while or have not used much CPU time receive an elevated priority.
- Implement a hierarchical or group-based scheduling policy where an environment's children receive a share of the CPU time originally allotted to its parent, so that the entire group of `primespipe` environments collectively get only as much CPU time as the `ls` or file server environments do individually.

Name:

Let's assume we modify the 6.828 kernel to build a high-performance Web server. We add a driver for ethernet card in in the Web server environment. Interrupts from the card are directly delivered to the Web server's environment, and the interrupt handler executes on a separate stack in the Web server's environment (in a similar style as the 6.828 kernel sends page faults to environments.) The Web server's interrupt handler runs with interrupts disabled, processes the received interrupt completely (including any TCP/IP and HTTP processing), returns all resources associated with handling the interrupt, and re-enables interrupts. Web pages in the cache are served straight from the interrupt handler. CGI scripts and pages not in the cache are handled by the Web server on the main stack (not in the interrupt handler).

4. [10 points]: Does this implementation suffer from receive livelock? If so, sketch a sequence of events that will result in receive livelock. If not, explain why. (If you need to make any more assumptions about the architecture of the system, be sure to state them explicitly.)

Yes, the implementation does suffer from receive livelock. Suppose HTTP requests arrive at a sufficient rate that by the time the web server's interrupt handler is finished with the interrupt processing for one request, the Ethernet card has already received another request, causing the interrupt to be triggered again immediately as soon as the web server re-enables interrupts. Since cached pages are served straight from the interrupt handler, the web server will be able to continue serving them. Pages that miss in the cache and dynamic CGI pages must be handled by the web server *outside* of the interrupt handler, however. As long as the web server keeps getting a continuous load of interrupts it will *never* get out of its network interrupt handler and get around to processing any of these cache misses or CGI requests. Thus, the web server is livelocked, making no forward progress at all on many of the requests it is supposed to be serving.

III File systems and reliability

Modern Unixes support the `rename(char *from, char *to)` system call, which causes the link named "from" to be renamed as "to". Unix v6 does not have a system call for rename; instead, rename is an application that makes use of the `link` and `unlink` system calls.

5. [5 points]: Give an implementation of rename using the `link` and `unlink` system calls, and briefly justify your implementation.

```
int rename(char *from, char *to) {
     /* 1 */ unlink(to);
     /* 2 */ link(from, to);
     /* 3 */ unlink(from);
}
```

Some editors use `rename` to make a new version of a file visible to user atomically. For example, an editor may copy "x.c" to "#x.c", make all changes to "#x.c", and when the user hits save, the editor calls `sync()` followed by `rename("#x.c", "x.c")`.

6. [5 points]: What are the possible outcomes of running `rename("#x.c", "x.c")` if the computer fails during the rename library call? Assume that both "#x.c" and "x.c" exist in the same directory but in different directory blocks before the call to `rename`.

Because the effects of the calls in our rename implementation are not persistent unless the corresponding block writes go out to disk, it is *not* correct simply to consider the failure happening before or after each call. Instead, we need to think about the possible block writes that can happen before the failure.

The rename implementation above causes three block writes, all possibly delayed (written with `bdwrite`). First, the directory entry for "to" is removed from its block. Second, the directory entry for "to" is added back to its block, but with the i-number for "from". Third, the directory entry for "from" is removed from its block.

The possible outcomes are:

- Nothing done: no changes written to disk yet.
- The first write went to disk: "from" remains as before, but "to" doesn't exist at all. (Looks like rename failed just after line 1 above.)
- The first and second writes went to disk: "from" remains as before, but now "to" points at the same file as "from". (Looks like rename failed just after line 2 above.)
- The first, second, and third writes went to disk: "from" is gone, and "to" points at the new file. (Looks like rename succeeded.)
- The first and third writes went to disk, *but not the second*: both "from" and "to" are gone! (Cannot be explained by examination of the rename implementation!)

Name:

Modern BSD UNIX implements `rename` as system call. The pseudocode for `rename` is as follows (assuming "to" and "from" are in different directory blocks):

```
int rename (char *from, char *to) {
   update dir block for ''to'' to point to ''from'''s inode // write block
   update dir block for ''from'' to free entry // write block
}
```

BSD's Fast File System (FFS) performs the two writes in `rename` synchronously (i.e., using `bwrite`).

7. [5 points]: What are the possible outcomes of running `rename("#x.c", "x.c")` if the computer fails during the rename system call? Assume that both "#x.c" and "x.c" exist in the same directory but in different directory blocks before the call to `rename`.

- Nothing done: state remains as-is before the `rename`.
- Only first ("to") update done: "from" and "to" both refer to the same file.
- Everything done: `rename` completed successfully.

Because of the atomic change of the "to" entry in this implementation, we never get into a state where "to" does not point to anything. Because the first write must go out before the second, we cannot lose the new copy of the file. Many common Unix applications today depend on this level of `rename` atomicity in the file system in order to ensure reasonable robustness against failures.

Assume the same scenario but now using FFS with soft updates.

8. [5 points]: How does using FFS with soft updates change this scenario?

In terms of robustness, soft updates do not change the file system's semantics at all: exactly the same situations are possible as above for FFS with synchronous writes, because the writes must go out in the same order. The difference is in *performance*: with soft updates, the file system does not have to perform all meta-data writes synchronously, and therefore can achieve much greater performance benefits from caching without compromising robustness.

Just to be precise, it's worthy of note that soft updates *can* potentially change the file system's failure semantics in ways that are theoretically *observable* to applications; just not in ways that create *file system metadata inconsistencies*. For example, suppose an application creates a file A somewhere in the file system, then creates a file B *somewhere else* in the file system that happens to involve completely unrelated directory and inode blocks. If a failure occurs during or shortly after these files are created, with soft updates the application could (on system restart) observe that file B was successfully created but A wasn't, because the soft updates algorithm imposes no ordering dependency between them. With synchronous writes, however the application would never observe that B as having been successfully created unless A had also been successfully created.

Name:

IV Virtual machines

A virtual machine monitor can run a guest OS without requiring changes to the guest OS. To do so, virtual machine monitors must virtualize the instruction set of a processor. Consider implementing a monitor for the x86 architecture that runs directly on the physical hardware. This monitor must virtualize the x86 processor. Unfortunately, virtualizing the x86 instruction set is a challenge.

9. [5 points]: Using the instruction "mov %cs, %ax", give a code fragment that shows how a guest operating system could tell the difference between whether it is running in a virtual machine or directly on the processor, if the virtual machine monitor is not sufficiently careful about how it virtualizes the x86 architecture.

(List a sequence of x86 assembly instructions)

```
movw    %cs, %ax
andw    $3, %ax
jnz     running_on_vm
```

The bottom two bits of the CS register on the x86 represent the processor's current privilege level, 0 meaning kernel mode and 3 meaning user mode. The guest operating system can therefore tell whether it is running in user or kernel mode simply by testing those two bits of CS. Since the "mov %cs, %ax" instruction is *not* privileged, the virtual machine monitor has no easy way to trap that instruction and emulate the *correct* behavior—i.e., to return a CS register value with 0 in the low two bits, indicating that the guest OS is running in "virtualized" kernel mode, even though the guest OS code is *actually* running in user mode as far as the physical hardware is concerned.

10. [5 points]: Describe a solution for how to virtualize the instruction "mov %cs, %ax" correctly. What changes do you need to make to the monitor?

(Give a brief description; no pseudocode required)

The virtual machine monitor first scans through all code in the guest OS before allowing it to run, replacing each non-virtualizable instruction such as the one above with a one-byte "`int $3`" instruction, and remembering the original instruction replaced at each point in the code using a table maintained elsewhere by the monitor. When the guest OS hits one of these "`int $3`" instructions, the virtual machine monitor intercepts the interrupt, scans its table of replaced instructions for that EIP address, emulates the correct behavior for the replaced instruction, and increments EIP past the space reserved for it.

Name:

The monitor might emulate a load of CR3 as follows:

```
// addr is a physical address
void
emulate_lcr3(thiscpu, addr)
{
  Pte *fakepdir;

  thiscpu->cr3 = addr;
  fakepdir = lookup(addr, oldcr3cache);
  if (!fakepdir) {
    fakedir = page_alloc();
    store(oldcr3cache, addr, fakedir);
    // CODE MISSING:
    // May wish to scan through supplied page directory to see if
    // we have to fix up anything in particular.
    // Exact settings will depend on how we want to handle
    // problem cases.
  }
  asm("movl fakepdir,%cr3");
  // Must make sure our page fault handler is in sync with what we do here.
}
```

11. [15 points]: Describe a solution for handling a guest OS that executes instructions stored in its data segment (e.g., `user_icode` in the 6.828 kernel).

The virtual machine monitor must use some protection mechanism to ensure that:

- Any attempt by the guest OS to "jump into" and run newly loaded or modified code will trap into the monitor, giving the monitor a chance to scan the code and replace non-virtualizable instructions as described for question 10.
- *After* a page of physical memory containing code has been scanned and modified, any attempts by the guest OS to *read or write* that page of memory (treating it once again as "data" instead of jumping into it as "code") will once again trap into the monitor, allowing the monitor to undo its modifications or simulate the correct read or write to the code page as if the page contained the guest OS's original, unmodified code.

There are different ways of achieving the above protection characteristics on the x86, but none of them are simple due to such practical complications as the lack of support for "execute-only" page mappings in the x86 architecture. Clever use of segment registers, privilege levels, and/or more intelligent guest OS code analysis and rewriting can help.

V Feedback

Since 6.828 is a new subject, we would appreciate receiving some feedback on how we are doing so that we can make corrections next year. (Any answer, except no answer, will receive full credit!)

12. [1 points]: Did you learn anything in 6.828, on a scale of 0 (nothing) to 10 (more than any other class)?

13. [1 points]: What was the best aspect of 6.828?

14. [1 points]: What was the worst aspect of 6.828?

15. [2 points]: If there is one thing that you would like to see changed in 6.828, what would it be?

16. [2 points]: How should the lab be changed?

17. [1 points]: How useful was the v6 case study, 0 (bad) to 10 (good)?

18. [1 points]: How useful were the papers, 0 (bad) to 10 (good)?

19. [2 points]: Which paper(s) should we definitely delete?

20. [2 points]: Which paper(s) should we definitely keep?

Name:

21. [1 points]: Rank TAs on a scale of 0 (bad) to 10 (good)?

22. [1 points]: Rank the professor on a scale of 0 (bad) to 10 (good)?

End of Quiz II

Name:

6.828 Operating System Engineering

Fall 2012

COMP 300E Operating Systems
Fall Semester 2011
Final Examination SAMPLE QUESTIONS

Disclaimer: these sample questions should not be used as predictors for the questions in the final exam. These questions are biased towards virtual memory and page replacement, but the final exam will consist of a balanced selection of questions from each chapter. You should also carefully review the sample questions provided on the lecture notes page, as well as the sample and actual midterm exam papers.

1. Single-Choice (Each question has a single correct answer) questions and brief Q&As.

* Which of the following is NOT a valid source of cache misses?
A. Compulsory
B. Capacity
C. Conflict
D. Trashing

ANS: D

* Which of the following statement is true for a Fully Associative Cache?
A. No conflict misses since a cache block can be placed anywhere.
B. More expensive to implement because to search for an entry we have to search the entire cache.
C. Generally lower miss rate than a fully-associate cache.
D. All of the above

ANS: D

* Which of the following statement is true for write-through cache and write-back cache?
A. A write-through cache will write the value back to memory when it changes. (For example if the value 'x' is stored in the cache and I increment it by one, then the incremented value is written to the cache and to physical memory as well.)
B. A write-back cache will only write the value back to memory when the cache block is evicted from the cache.
C. All of the above

ANS: C

* Compare fully-associative cache and direct-mapped cache: _____ cache has lower miss rate; _____cache has smaller hit time?
A. fully-associative, direct-mapped
B. direct-mapped, fully associative

ANS: A
(A fully-associative cache has lower miss rate, since it eliminates conflict misses. A direct-mapped cache has smaller hit time, since cache lookup is a simple table index operation, while a fully-associative cache needs to search through all cache blocks to find a match.)

* True or False: TLBs are typically organized as a directly-mapped cache to maximize performance.
ANS: False
(TLB is usually fully-associative, but can also be set-associative, to minimize miss rate (as compared to fully-associative). Since TLBs are typically very small, the hit time can still be very fast despite the associativity.)

* For a single-level page table system, with the page table stored in memory without a TLB. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?
A. 600 nanoseconds
B. 200 naboseconds
C. 400 nanoseconds
D. can't say
Ans: C

* For a single-level page table system, with the page table stored in memory. If the hit ratio to a TLB is 80%, and it takes 15 nanoseconds to search the TLB, and 150 nanoseconds to access the main memory, then what is the effective memory access time in nanoseconds?
A. 185
B. 195
C. 205
D. 175
ANS: B. 0.8*(150+15)+0.2*(300+15)=195 $(m + s)\, h + (2m + s)(1 - h)$

* In a 64 bit machine, with 256 MB RAM, and a 4KB page size, how many entries will there be in the inverted page table?
A. 2^16
B. 2^50
C. 2^14
D. None of the above
Ans: A Total physical frames is 256MB/4KB=64K=2^16, which is number of PTEs in IPT.

* Consider the segment table: What are the physical address for the following logical addresses (Segment ID, Segment Offset) :
a. 0,430
b. 1,10
c. 1,11
d. 2,500

Segment	*Base*	*Length*
0	**219**	**600**
1	**2300**	**14**
2	**90**	**100**
3	**1327**	**580**
4	**1952**	**96**

ANS:
A. 219+430 = 649.
B. 2300+10=2310.
C. 2300+11=2311
D. Illegal address since size of segment 2 is 100 and the offset in logical address is 500.

* In which of the following operations, the OS scheduler is invoked?
A. Process requests for I/O.
B. Process finishes execution.
C. Process finishes its time slot.
D. All of the above A through C
E. None of the above A through C
ANS: D

* What happens if the time slice allocated in a Round Robin Scheduling is very large? And what happens if the time slice is very small?

ANS: If time slice is very large, it results in FCFS scheduling. If time slice is too small, the processor through put is reduced, since more time is spent on context switching.

* What are the two principles of locality that make implementing a caching system worthwhile?

ANS: Spatial Locality – (If I access a block of data, I am likely to access blocks of data next to it in the future. A simple example is sequentially reading through an array. In this scenario, a caching system with a large block size takes advantage of spatial locality because when I read the first element, I read in seven additional elements so that the next seven reads are from the cache and therefore fast.)
Temporal Locality – (If I have accessed a block of data before, I am likely to access this block of data again. A simple example is if I have a piece of code that constantly references/updates a global counter.)

* True or False: A direct mapped cache can sometimes have a higher hit rate than a fully associative cache with an LRU replacement policy (on the same reference pattern).

ANS: True. A direct mapped cache will do better than LRU on a pattern like ABCDEABCDE… if the cache size is one entry smaller than the total number of items in the pattern (e.g., four cache entries). LRU will miss on every access, while a direct mapped cache will only miss on the two entries that map to the same cache entry.

* True or False: Virtual memory address translation is useful even if the total size of virtual memory (summed over all programs) is guaranteed to be smaller than physical memory.

ANS: True. It provides protection between different processes, it doesn't require that each program's addresses be contiguous.

* What type of file access pattern exploits spatial locality?

ANS: Spatial locality is accessing a location that is close to or next to recently accessed location. Sequential access of a file exploits spatial locality.

* Which component of disk access time is the disk scheduling algorithm trying to minimize?

ANS: The disk scheduling algorithm is attempting to minimize overall time wasted to moving the disk arm (i.e. it optimizes seek time).

* Name at least two ways in which the *buffer cache* is used to improve performance for file systems.

ANS: 1) Reads can be from cache instead of disk
2) Allow delayed writes to disk, thereby permitting better disk scheduling an/or temporary files to be created and destroyed without even being written to disk.
3) Used to cache kernel resources such as disk blocks and name translations

* Suppose a new process in a system arrives at an average of six processes per minute and each such process requires an average of 8 seconds of service time. Estimate the fraction of time the CPU is busy in a system with a single processor.

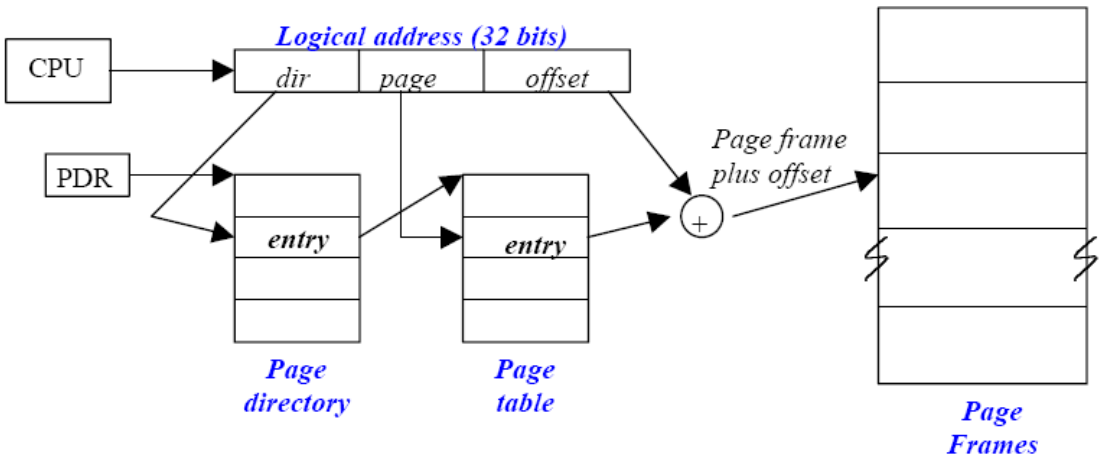
ANS: Given that there are on an average 6 processes per minute. So the arrival rate = 6 process/min. i.e. every 10 seconds a new process arrives on an average.
Or we can say that every process stays for 10 seconds with the CPU
Service time = 8 sec.
Hence the fraction of time CPU is busy = service time / staying time
= 8 / 10
=0.8
So the CPU is busy for 80% of the time.

* If the system does not have enough memory to contain all the processes' working sets and is thrashing, and the page replacement algorithm is CLOCK, then does the clock hand move quickly or slowly? Why?

ANS: Quickly. Since all pages are hot, so the R bits of all pages are continuously set to 1, the clock hand may come around multiple cycles to find a victim page. And there are a lot of page faults, so the clock hand needs to find victim page to replace very often.

* Suppose that we have two-level page tables. Each page is 4KB pages and each Page Table Entry (PTE) is 4 bytes. Suppose we want each page table to occupy exactly one memory page. What is the format of a 32-bit virtual address?

ANS: 4 KB = 2^12 bytes, which means page offset is 12 bits to address a specific byte within the page. This leaves us with 20 bits we need to allocate. Since each page is 4 KB and each PTE is 4 bytes, this means that we can fit 4KB/4 bytes=1024 (2^10) PTEs on a page. Since we want each page table to occupy exactly one memory page, each page table should contain 2^10 PTEs. This means that we can use 10 bits to represent each level in the page table.
Thus, the breakdown is as follows:
10 bits to reference the correct page table entry in the first level.
10 bits to reference the correct page table entry in the second level.
12 bits to reference the correct byte on the physical page.
(Note I will not ask you to draw such a figure in the exam.)



* Multi-Level Page Tables
(1): Suppose we have a memory system with 32-bit virtual addresses and 4 KB pages. If the page table is full, i.e., there are no null pointers in the page table hierarchy, show that a 20-level page table consumes approximately twice the space of a single level page table.

ANS: 4 kilobytes = 2^12 bytes, which means page offset is 12 bits to address a specific byte within the page. This leaves us with 20 bits we need to allocate. We have 20 bits to work with and a 20-level page table, hence there is one-bit in the virtual address for each level of the 20-level page table. Each level of page table consists of 2^1=2 entries (where a '0' bit

references the first entry while a '1' bit references the second entry). The total number of page tables in this implementation is therefore:
2^0 + 2^1 + ... + 2^19 = 2^20 - 1
Since each table has two entries, this means that we have a total of 2^21 - 2 entries.
A single-level page table on the other hand, has 2^20 entries, meaning that there are a total of 2^20 entries in such an implementation. Therefore the 20-level page table consumes approximately twice the space when full.

(2) Show that the above is not necessarily true for a sparse page table, where the process memory size is small, and many pointers in the page table hierarchy are null.

ANS: Imagine if we only have 1 page of physical memory allocated to a process. This means that we have only 20 tables, since we just need one entry per level of indirection. Each table has 2 entries, so we have a total of 40 entries for our page table implementation.
In the single-level case, in the sparse memory scenario, we still have 2^20 entries allocated, meaning that the 20-level page table is theoretically feasible for sparse memory usage (although 20 memory accesses to determine a translation is still expensive and therefore not practical).

* Page replacement
For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:
A B C D E F C A A F F G A B G D F F
Indicate in the following table which pages are mapped to which physical pages for each of the following policies. *If there is no page fault at that time-step, leave the column blank*. Here MIN refers to OPT (optimal policy)

Access→		**A**	B	C	D	E	F	C	A	A	F	F	G	A	B	G	D	F	F
FIFO	1	A				E									B				
	2		B				F										D		
	3			C					A									F	
	4				D								G						
MIN	1	*A*															*D*		
	2		*B*														*D*		
	3			*C*									*G*				*D*		
	4				*D*	*E*	*F*												
LRU	1	*A*				*E*							*G*						
	2		*B*				*F*										*D*		
	3			*C*											*B*				
	4				*D*			*A*										*F*	

For MIN, we accepted the final D in any of the first three pages.
Grading: 2 pts. each for MIN and LRU; -1 for each error.

* For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:
A B C D E A A E C F F G A C G D C F

Indicate in the following table which pages are mapped to which physical pages for each of the following policies. Assume that a blank box matches the element to the left. We have given the FIFO policy as an example.

Access→		**A**	B	C	D	E	A	A	E	C	F	F	G	A	C	G	D	C	F
FIFO	1	A				E									C				
	2		B				A										D		
	3			C							F								
	4				D								G						
MIN	1	*A*																	*F*
	2		*B*			*E*					*F*		*G*						*F*
	3			*C*															*F*
	4				*D*														*F*
LRU	1	*A*				*E*								*A*					*F*
	2		*B*				*A*						*G*						
	3			*C*															
	4				*D*						*F*						*D*		

Any one of these

* Consider a virtual memory architecture with the following parameters:

Virtual addresses are 48 bits. Hence the architecture has virtual address space of 64 Terabytes (TB), i.e., it allows a maximum of 64TB physical memory (RAM). (1TB=10^12 =2^40bytes)

The page size is 32KB.

The first- and second-level page tables are stored in physical memory.

All page tables can start only on a page boundary.

Each second-level page table fits exactly in a *single* page frame.

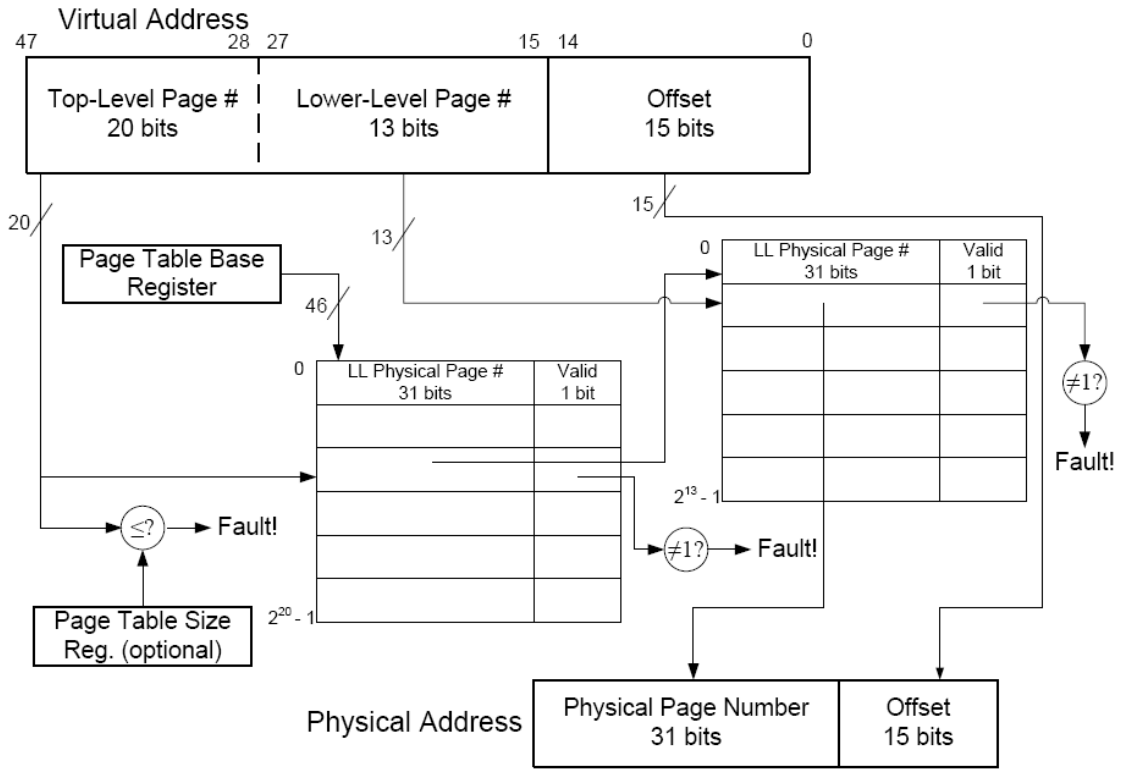
Each Page Table Entry (PTE) is 32 bits, consisting of 31-bit PPN and 1 valid bit, for both 1st and 2nd-level page tables. Assume there is a single valid bit for each PTE and no other extra permission, or dirty bits.

Draw and label a figure showing how a virtual address gets mapped into a real address. You should list how the various fields of each address are interpreted, including the size in bits of each field, the maximum possible number of entries each table holds, and the maximum possible size in bytes for each table (in bytes).

ANS: The page size is 32KB→Page offset is 15 bits.
A 32 KB page can thus hold 8K 4-byte PTEs, and each second-level page table fits exactly in a single page frame, hence each 2nd level page table has 8K entries. So we should allocate 13 bits (2^13=8K) to the 2nd level page table, and 20 bits (48-15-13) to the 1st level page table.

The first-level page table has 2^20 (1 million) entries, each of which is 32 bits (31-bit PPN plus valid bit, or 4 bytes), so it has a maximum size of 4 MB.

(Note I will not ask you to draw such a figure in the exam.)


Virtual Address
47
28
27
15
14
0
Top-Level Page #
20 bits
Lower-Level Page #
13 bits
Offset
15 bits
20
13
15
Page Table Base
Register
46
0
LL Physical Page #
31 bits
Valid
1 bit
0
LL Physical Page #
31 bits
Valid
1 bit
≠1?
Fault!
2^13 - 1
≤?
Fault!
≠1?
Fault!
Page Table Size
Reg. (optional)
2^20 - 1
Physical Address
Physical Page Number
31 bits
Offset
15 bits

CS 354 Operating Systems

Study Question Solutions

1 Processes and Threads

1. Longer quanta reduce scheduling overhead, since scheduling decisions are made less frequently. Longer quanta can increase response time for interactive processes.

2.

Proc. Number	Finish Time	Response Time
1	6	6
2	14	13
3	12	9

Average response time: $\frac{6+13+9}{3} = 9.33$

3.

Proc. Number	Finish Time	Response Time
1	10	10
2	14	13
3	13	10

Average response time: $\frac{10+13+10}{3} = 11$

This solution assumes that (1) at time 3, process 3 gets put onto the ready queue before process 1, whose quantum is expiring at that time, and (2) that when a process exits during its quantum, the first ready process runs during the remainder of that quantum. Different assumptions give different answers.

4.

Proc. Number	Finish Time	Response Time
1	4	4
2	14	13
3	8	5

Average response time: $\frac{4+13+5}{3} = 7.33$

5. Response time is the time between a process' creation and its exit, i.e., it is the amount of time that a process exists in the system. For a round-robin scheduler, the solution depends on the order of the processes in the ready queue. Assuming that process k is at the front of the ready queue and that other processes appear in decreasing order of required computing time, we can write the following expression for $\bar{R}$, the average response time:

$$\begin{aligned} \bar{R} &= \frac{k + (k + (k-1)) + (k + (k-1) + (k-2)) + \ldots + \sum_{i=1}^{k} i}{k} \\ \bar{R} &= \frac{\sum_{i=1}^{k} i^2}{k} \end{aligned}$$

For a SJF scheduler, we have:

$$\begin{aligned} \bar{R} &= \frac{1 + (1+2) + (1+2+3) + \ldots + \sum_{i=1}^{k} i}{k} \\ \bar{R} &= \frac{\sum_{i=1}^{k} i(k-i+1)}{k} \end{aligned}$$

6. • block waiting for a device (e.g., the console)

- block waiting for another process (e.g., Join() in Nachos)
- terminate

7. To create a new process, the operating system must create a new thread of control, a new address space, and new instances of any per-process data structures (such as open file tables and page tables). It must initialize the address space and the per-process tables. It must assign a unique identifier to the process, and it must place the process on the ready queue.

   To create a new thread within a process, the operating system must create a new thread of control and must associate it with an existing address space and data structures. It must assign a unique identifier to the new thread, and it must place the thread on the ready queue.

8. If the threads are not supported by the operating system, T_{21}, T_{22}, or T_{31} will run during the next quantum. (This assumes that none of the processes are blocked and that they are scheduled round-robin by the operating system.)

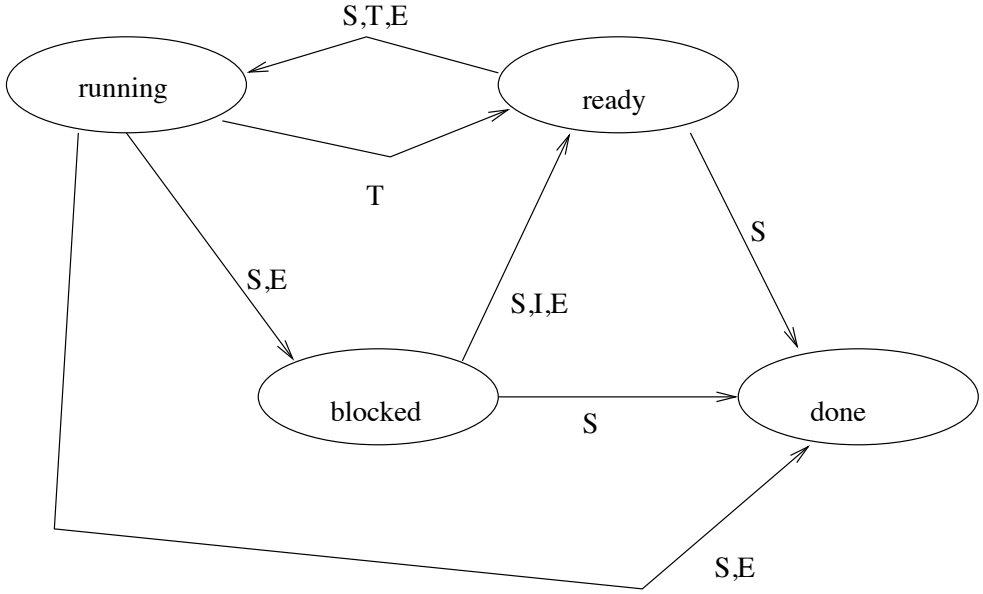
   If the threads are supported by the operating system, then any thread except T_{11} will run during the next quantum, assuming not all threads are blocked and scheduling is round-robin.

9. If the threads are not supported by the operating system, either T_{12} or T_{13} will run after T_{11} finishes.

   If the threads are supported by the operating system, any of the other threads may run.

10. A non-preemptive scheduler assigns the processor to a process until that process is finished. A preemptive scheduler may force a runnable process to yield the processor before it is finished so that another runnable process can be allowed to run.

11. If you consider the Nachos Yield() system call, an arc labeled "S" should be included from "running" to "ready".

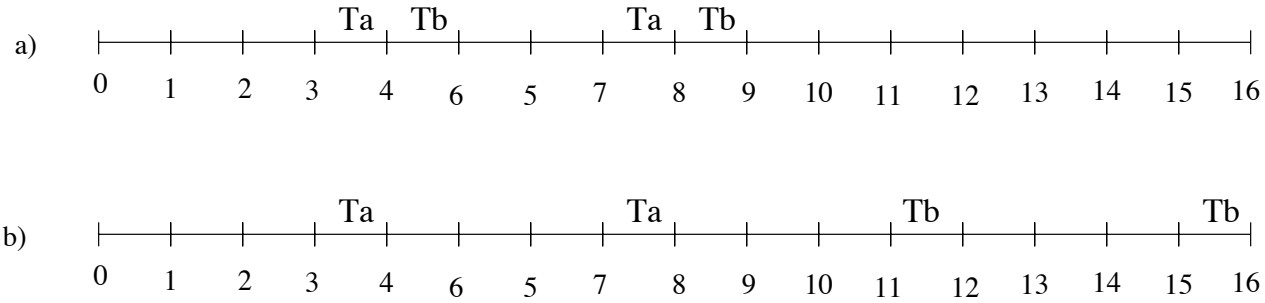


12. **a.** Process 1 will receive one time unit, process 2 will receive 2 units, and process 3 will receive three.

    b. Each process will receive two time units.

    c. Scheduler (b) is a fair share scheduler. Scheduler (a) is not. Scheduler (a) could be modified to become a two-level scheduler. The top level would select a process to run, using a fair scheduling discipline like round-robin. The bottom level would select one thread from the process that was selected by the top level scheduler.

13.



14. When $Q \geq T + S$, the basic cycle is for the process to run for T and undergo a process switch for S. Thus, 1. and 2. have an efficiency of $\frac{T}{T+S}$. When the quantum is shorter than $S + T$, each run of T will require $T/(Q - S)$ process switches, wasting time $ST/(Q - S)$. The efficiency here is then

$$\frac{T}{T + ST/(Q - S)}$$

For 4., by setting $Q = S$ in the formula above, we see that the efficiency is zero - no useful work is done.

15. For round robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of 10 minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time D finishes. Then each of the remaining three jobs get 1/3 of the CPU for 6 minutes, until B finishes, and so on. The finishing times for the five jobs are 10, 18, 24, 28, and 30, for an average of 22 minutes. For priority scheduling, B is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 20.0 minutes. If the jobs run in the order A through E, they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes. Finally, shortest job first yields finishing times of 2, 6, 12, 20, and 30, for an average of 14 minutes.

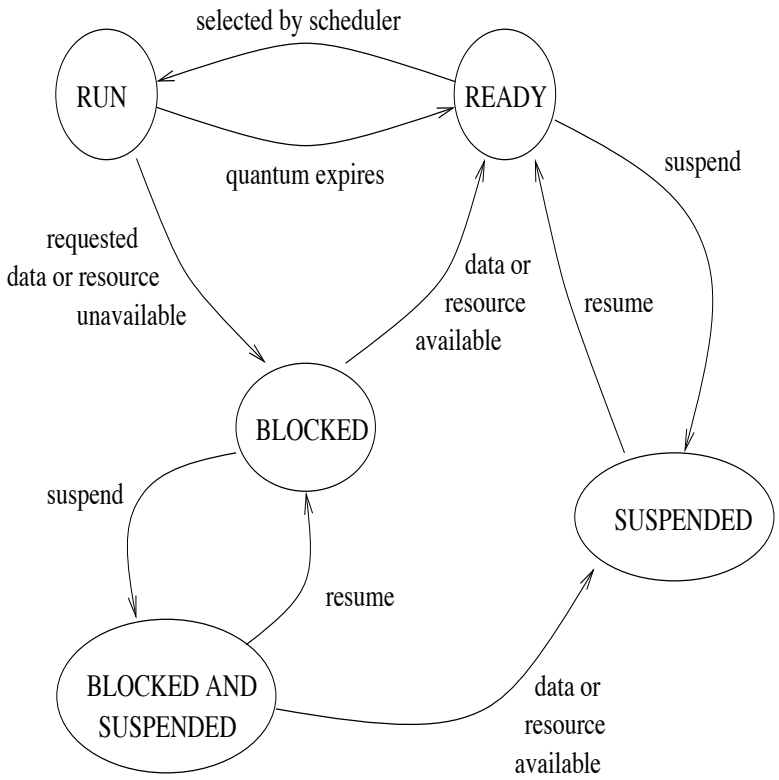
16. Once the high priority process blocks (after running for time t_c), the low priority process will run for a quantum. If the high priority process is still blocked after that quantum, the low priority process will receive another. When the high priority process unblocks, the low priority process will finish out the current quantum, at which point the scheduler will give the processor to the high priority process. Thus, for every t_c time the high priority process runs, the low priority process will run for

$$\lceil \frac{t_b}{q} \rceil q$$

units of time. The fraction of CPU time used by the low priority process is:

$$\frac{\lceil \frac{t_b}{q} \rceil q}{\lceil \frac{t_b}{q} \rceil q + t_c}$$

17.



18. **a.** Suppose that $2^{p-1} \leq c < 2^p$. Initially, the process will climb to priority $p-1$ since

$$c \geq 2^{p-1} > \sum_{i=0}^{p-2} 2^i$$

It will then drop to $p-2$ after blocking during its $p-1$ quantum. If $2^{p-2} + 2^{p-1} \leq c < 2^p$, the process will move to p_1 and then p when it next runs, and then alternate between $p-1$ and p forever. If $2^{p-1} \leq c < 2^{p-1} + 2^{p-2}$ then the process will alternate between $p-2$ and $p-1$ forever.

b. Assuming both types of processes have been in the system for a long time, the Bs will have a very low priority and will only run when A is blocked. So, they get fraction

$$\frac{b}{c+b}$$

c. If there are enough A process, the Bs will never get to run.

19. **a.** If a process uses its entire size q quantum, the scheduler will allow that process to run until either it finishes or it blocks, since it will have higher priority than all others. If the process blocks, it will rejoin the round-robin q-quantum queue. In other words, this is essentially a round-robin scheduler that may give some processes a longer "turn" than other processes. Round-robin schedulers do not cause processes to starve.

b. Suppose that a process (P_1) uses its entire size q quantum, and then a steady stream of new processes arrives in the system such that there is always at least one such process ready to run. These processes will have higher priority than P_1, and the scheduler will always choose to run them rather than P_1. P_1 will starve.

20. **a.** Because the thread implementation is user-level, the Consumer cannot run while the Producer reads from the file. Thus, the total time for the two threads will be the sum of the times for the two individual threads, or $10(t_r + t_c)$.

b. With an OS thread implementation, the Consumer can run while the Producer is blocked. There are two cases to consider:

$t_c \leq t_r$**:** In this case, the Consumer will finish consuming buffer[i] before the Producer finishes producing buffer[$i+1$]. The total time will be the time required to produce all ten buffers plus the time for the Consumer to consume the final buffer, i.e., it will be $10t_r + t_c$.

$t_c > t_r$**:** In this case, the Consumer is the bottleneck. The total time will be the time required to consume all of the buffers, plus the time the Consumer must wait while the first buffer is being produced, i.e. the total time will be $t_r + 10t_c$.

2 Virtual Memory

1. There can be at most $2^{21}/2^{10}$ frames in main memory, so a page table entry will require 11 bits for a frame number, plus protection and reference bits, etc. A page table may have up to $2^{24}/2^{10} = 2^{14}$ entries. A virtual address has 24 bits. Virtual address 1524 lies on page 1 which resides in frame 8 with an offset of 1524 modulo 1024, or 500. So, 1524 maps to physical address $8 * 1024 + 500 = 8192 + 500 = 8692$. Physical address 10020 is in frame 9, with an offset of 804. Virtual page 4 maps to frame 9, so virtual address $4 * 1024 + 804 = 4900$ will map to 10020.

2. The diagram should include a virtual address that is 22 bits wide. The most significant 3 bits of the virtual address should be used to select one of eight entries from a segment table. A base register, 21 bits wide, should point to the physical address of the first entry in the segment table. Each segment table entry should include the physical address (21 bits) of the first entry of a page table for that segment. The entry should also include the length of the page table. Seven bits are required since each page table may have up to $2^{19}/2^{12} = 2^7$ entries. Seven bits of the virtual address (the seven bits to the right of the 3 segment selector table bits) are used to select one page table entry from the page table. The PTE should include a 9 bit frame number plus protection and other bits. The remaining 12 (least significant) bits of the virtual address are used to select one of the 2^{12} offsets within the frame.

3. For LRU:

Frame	Reference																
	1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
1		2	2	2	2	2	2	3	3	3	3	4	4	4	6	6	6
2				3	3	3	4	4	4	4	2	2	2	5	5	5	1
Fault?	Y	Y	N	Y	N	N	Y	Y	N	N	Y	Y	N	Y	Y	Y	Y

The total number of faults is 11.

For clock:

Frame	Reference																
	1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
0	1	1	1	1	1	1	4	4	4	4	4	4	4	5	5	5	5
1		2	2	2	2	2	2	2	1	1	1	1	1	1	6	6	6
2				3	3	3	3	3	3	3	2	2	2	2	2	2	1
Fault?	Y	Y	N	Y	N	N	Y	N	Y	N	Y	N	N	Y	Y	N	Y

The total number of faults is 9.

4. Reading virtual address v will require four memory accesses in case of a TLB miss, one for the segment table, and one for each level of page table, and one for the actual physical memory location corresponding to v. In case of a TLB hit, only one memory access (for the physical location itself) will be required. So, if h is the hit ratio of the TLB, we need $4T(1-h) + T(h) = 2T$, i.e., $h = 0.67$.

5. $2^{(a+b+c)}$, since there are $2^{(a+b+c+d)}$ total addresses in the address space, and there are 2^d on each page.

6. Swapping means moving entire address spaces between the disk and the memory. Paging means moving individual pages, so that part of an address space may be on disk while the other part is in main memory.

7. Segments are variable-sized, and are visible to user processes. Pages are fixed-size and are not visible.

8. The operating system must locate the page table for the process that is to start running. It must set the base register in the MMU so that it points to the page table in memory, and it must set the length register if present. Finally, it must clear any now-invalid cached address translations from the TLB.

9. One method is a length register, which can be used to limit the length of a single page table. Another method is to use multiple levels of page tables. A lower-level page table that is not needed is simply not allocated, and its corresponding entry in a higher level table is marked as invalid.

10. An associative memory is one that is accessed by content rather than by address. Often an "entry" in an associative memory will have two parts, a tag and a value. To access the associative memory, one supplies a tag. If that tag matches the tag of an entry in the associative memory, that entry is returned. In a hardware implementation (e.g., a TLB), the tags of all entries are checked in parallel.

11. The frame to page mapping is useful during page replacement. The operating system must know which page resides in each frame so that it can check the page table entry for that page and so that it can mark that page table entry as invalid if the page is replaced.

12. Coalescing holes means combining one or more adjacent holes to form one larger hole.

13. Advantages of larger page sizes:

    - smaller page tables
    - faster sequential access to on-disk portions of the address space, since a few large blocks can be retrieved more quickly than many small ones

    Disadvantages:

    - increased wasted space in the last page of the address space (internal fragmentation).

14. page size: 2^{10}, max segment size = 2^{16}

15.

    a. Frames 0 and 2 are selected for replacement and are added to the free list. The new page tables should look as follows:

    Process 1 Page Table

	Frame	R	M	V
0	6	0	1	1
1	4	0	1	1
2	5	1	0	1
3	0	0	0	0
4	1	0	0	1
5	9	1	0	1

    Process 2 Page Table

	Frame	R	M	V
0	0	0	0	1
1	7	1	0	1
2	8	0	0	1
3	2	0	1	1
4	0	0	0	0
5	3	0	1	1

    b. Page 4 of process 2 must be moved from disk to memory. If you choose to replace frame 0, no pages are moved to the disk from memory, and the new page table for process 2 is as follows:

	Frame	R	M	V
0	0	0	0	0
1	7	1	0	1
2	8	0	0	1
3	2	0	1	1
4	0	1	0	1
5	3	0	1	1

    If you choose to replace frame 2, page 3 of process 2 must be copied from disk to memory, and the new page table for process 2 is as follows:

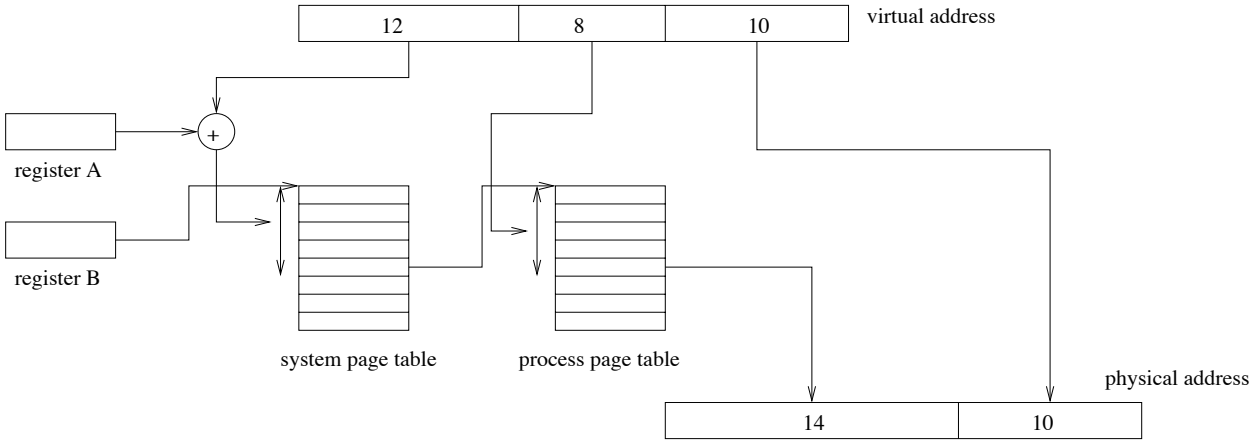
	Frame	R	M	V
0	0	0	0	1
1	7	1	0	1
2	8	0	0	1
3	0	0	0	0
4	2	1	0	1
5	3	0	1	1

16. **a.** 100,650 → 1000,50 → 4000,700 → 4900,100

    b. 100,500 → 700,50 → 800,25 → 1000,200 → 3800,900 → 4900,100

17. **a.** Register A requires only 20 bits, since the page tables are page-aligned in the system's virtual address space. Register B requires 24 bits. The max. page table size is 2^{22} bytes.

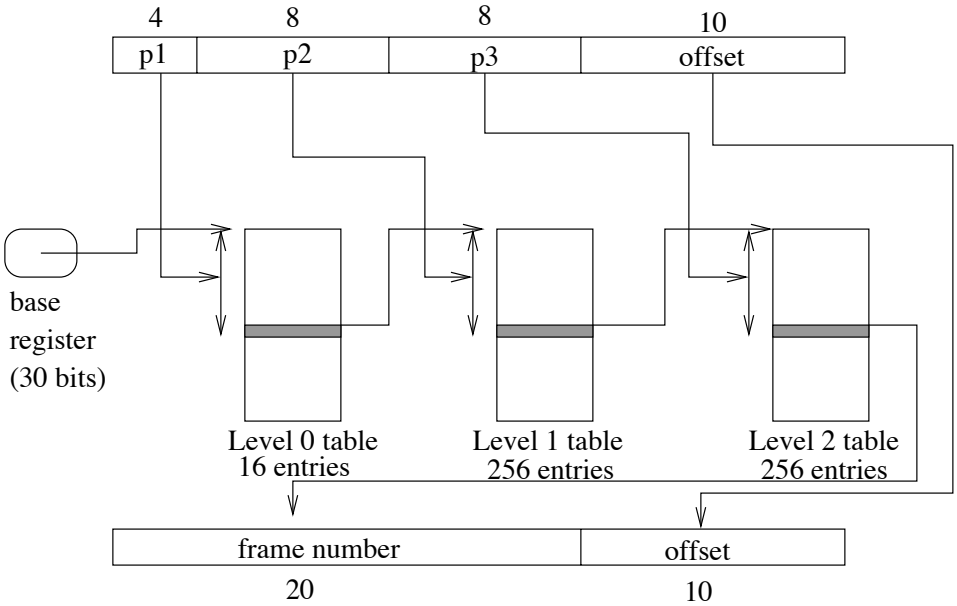
b. This is a tricky question! The value in register A is combined with 12 bits from the virtual address to produce an offset into the system's page table. If register A contains a complete virtual address, it should be divided by the page size before being combined with the bits from the virtual address. (This division operation is not shown in the diagram.) If register A holds a page number, no division is needed.



18.
- The TLB is used as a cache for the page table, which resides in primary memory.
- Primary memory is used as a cache for pages, which reside in secondary memory.

19.
- The code segment contains the executable program.
- The data segment contains initialized and uninitialized global variables and dynamically-allocated variables (the heap).
- The stack segment contains the program stack.

20. The lowest level of page tables requires up to $2^{30}/2^{10} = 2^{20}$ page table entries. At most 2^8 page table entries can fit on one frame. So, at least three levels of page tables will be required. This diagram assumes maximum page table sizes of 2^4 at the top level and 2^8 at the two lower levels.



21. **a.** (a) Any virtual address in the 300's. This will cause page 3 to be paged in and page 1 to be paged out.
(b) Any virtual address in the 100's. This will cause page 1 to be paged in and page 4 to be paged out.

(c) Any virtual address in the 400's.

b. Any sequence of three addresses in which all are less than 500 and none are in the 300's.

22. **a.** physical address 130

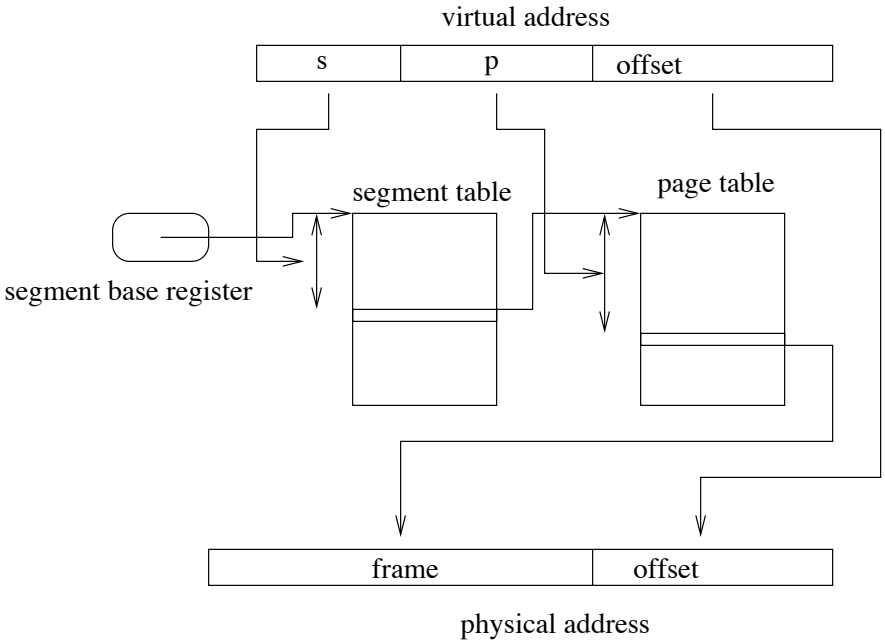
    b. does not map

    c. virtual address 250 of process 1

23. **a.** 2998,2999,0000,0001

    b. 1998,1999,3000,3001

24.



25. The effective instruction time is 100h + 500(1-h), where h is the hit rate. If we equate this formula with 200 and solve for h we find that h must be at least 0.75.

26. The aging algorithm is not a stack algorithm unless some mechanism is provided for breaking "ties" among pages with the same age. The mechanism should order tied pages the same way regardless of the size of the memory. For example, ordering based on page number will work.

27. LRU requires that recency information be updated every time a page is referenced. Since one or more references may occur each time an instruction is executed, a software implementation would, at best, be extremely inefficient.

28. Under a global policy, a page fault by process P_1 may cause replacement of a page from P_2's address space. Under a local policy, a fault by P_1 can only cause replacement of P_1's pages - the total number of frames allocated to each process remains constant.

29. The software can maintain a R bit in each PTE. It can use V and R as follows:

V	R	Meaning
0	0	page is in memory, but not referenced
0	1	page is not in memory
1	0	not used
1	1	page is in memory and referenced

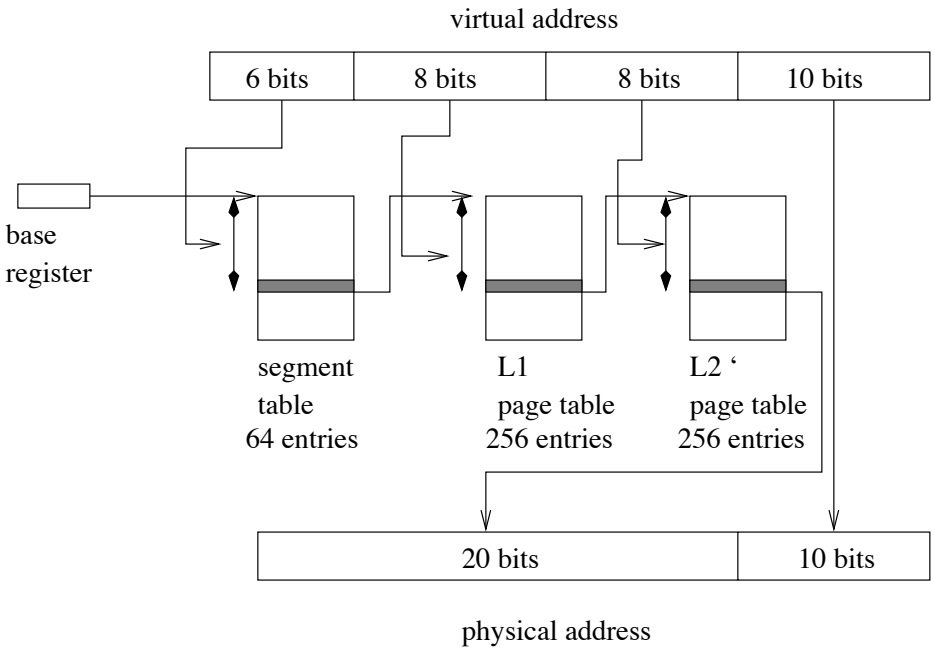
- When the OS wishes to clear the R bit, it should clear the V bit as well. (This is a 11 to 00 transition.) This will cause a page fault to occur the next time the page is referenced.
- When a page fault occurs (V is 0), the OS checks the R bit to determine whether the page is actually in memory or not. If R is 0, the page is in memory, and the OS sets both R and V to one. This indicates that the page is referenced, and it prevents further page faults. (We only need to get a fault on the first reference to the page that occurs after the R bit has been cleared). If R is 1, the page is not in memory. This is a regular page fault. The OS must bring the page into memory. It sets V and R to one for the new page (since it is about to be referenced). The page table entry for the replaced page gets V set to one and R to zero, to indicate that it is no longer memory-resident.

30.

a. Two levels are required, because there are 2^{16} pages per segment and 2^8 page table entries can fit in a single page table.

b. 2^{26} bytes.

c.



31. The graph should have page fault rate on the dependent (vertical) axis and page frames assigned on the independent (horizontal) axis. The page fault rate should be low and (roughly) flat if the number of page frames assigned is greater than W, and should increase as the number of frames drops below W. The graph should be non-increasing with as additional frames are assigned because the replacement algorithm is a stack algorithm.

32. The MMU hardware causes an exception to occur. (This transfers control to the operating system's exception handler.)

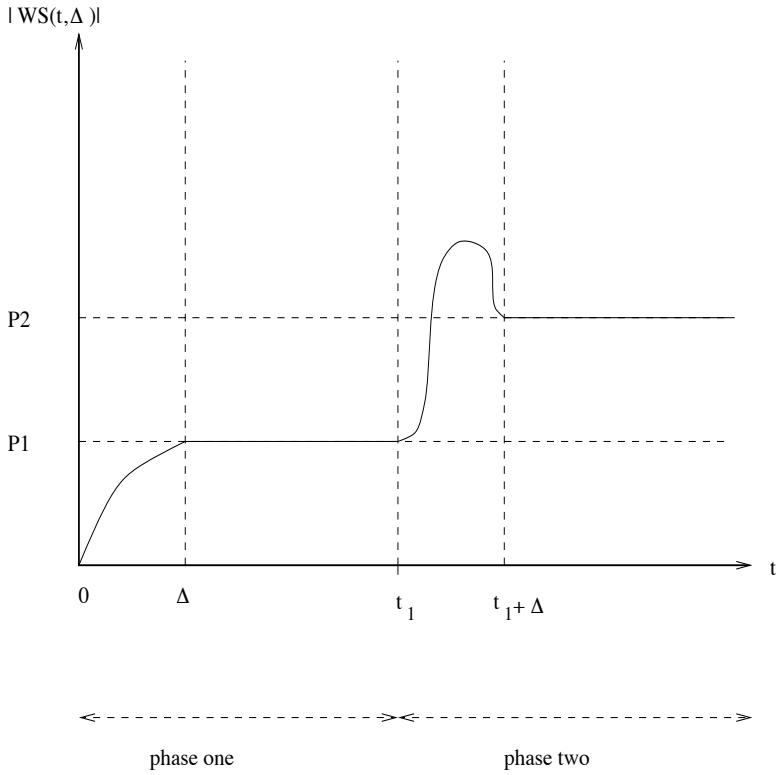
33. This solution skips lots of details. The general idea is to use the P bit to force an exception to occur when a shared page is about to be updated.

    When the a child process is created, its page table should be created as a duplicate of the parent's table, i.e., both parent and child point to the same in-memory copy of each page. The P bit should be set for all entries of both tables (parent and child) so that an attempt to update any page will result in an exception.

    When a protection fault occurs, the offending page should be duplicated. One process's page table should be made to point to the duplicate and other other to the original. The P bit can be cleared for both processes so that further updates do not cause exceptions.

This solution assumes that the only thing the protection mechanism is used for is to implement copy-on-update. If the protection mechanism is used for other things, then the OS needs a way to distinguish pages that are protected for copy-on-update reasons from pages that are protected for other reasons. (In one case, a protection fault should cause the page to be duplicated, in the other it should not.) One way for the OS to handle this is to maintain a copy-on-update bit in each page table entry.

34.



35.

ref str	1	2	1	3	2	1	1	4	3
stack	1	2	1	3	2	1	1	4	3
		1	2	1	3	2	2	1	4
				2	1	3	3	2	1
								3	2
dist str	∞	∞	2	∞	3	3	1	∞	4

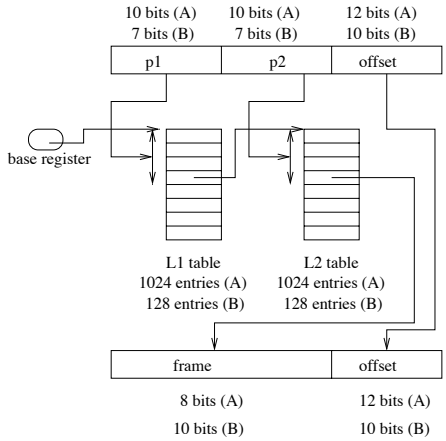
36. The optimal page replacement policy (OPT) can be used to determine the required lower bound. There will be at least five faults.

ref str	1	2	3	2	3	1	3	1	2	1
frames	1	1	3	3	3	3	3	3	2	2
		2	2	2	2	1	1	1	1	1
fault?	x	x	x			x			x	

37. **a.** This is a write into segment 0. It will fail (protection fault) because segment 0 is read-only.

b. This is a read from segment 1, offset 1. This corresponds to physical address $1000 + 1 = 1001$.

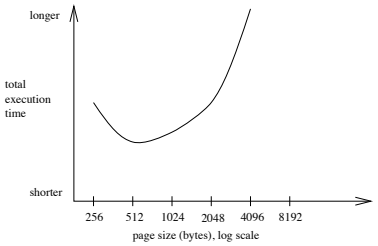
c. This is a write to segment 3, offset $4000 - 3072 = 928$. Segment 3 is only 300 bytes long, so this will result in a segmentation fault (segment length violation).



38. **a.**

    b. A total of 3000 level two PTEs will be required for the code and data. These will occupy $\lceil \frac{3000}{1024} \rceil = 3$ level two page tables. 100 PTEs for the stack segment will occupy another level two page table. In addition, there is a single level 1 page table, for a total of 5 page tables (occupying one frame each).

39. **a.** Suppose that part or all of a process' working set is not in memory and must be paged in. If pages are very small, more page faults will be required to bring in the missing parts of the working set. This is what drives the curve up when the pages get too small. Large pages reduce the number of page faults required to bring in the working set. However, as pages become larger it becomes more likely that they will contain non-working-set data as well as working-set data. This unnecessary data will occupy memory space that could have been used for the working set. This drives the curve up when the page size gets too large.



    b. This curve assumes that faults for larger pages take longer to handle than faults for smaller pages. This will tend to drive the minimum of the curve towards the left (where each page fault is cheaper). The curve sketched assumes this effect is strong enough to make a 512 byte page size better than an 1024 byte page size, though the latter results in fewer faults. If it is assumed that all faults take the same amount of time to handle, the minimum of the curve should remain at 1024.

40. The minimum is zero memory accesses, since on a TLB hit the entire translation is found in the TLB. The maximum is 3 access in case of a TLB miss, since 3 page tables will need to be queried to complete the translation.

41. Clock is not a stack algorithm. Clock can degenerate to FIFO - FIFO is subject to Belady's anomaly and so is not a stack algorithm.

42. **a.** The OS will need to read in the filename parameter to the Open system call. In the worst case, this may span two pages of the address space.

    b. As a result of the Read call, the OS will need to update "buf", which is 1500 bytes long. In the worst case, this could span three 1024-byte pages.

    c. The buffer resides at virtual addresses 2100 through 3599, i.e, on pages 2 and 3 of the address space. Virtual 2100 is offset 52 on page 2, which is in frame 1. Virtual 3599 is offset 527 on page 3, which in in frame 4. So, the buffer occupies physical 1076 to 2047, and physical 4096 to 4623.

3 Disks and Other Devices

1.
   - bytes/cylinder = TSB
   - to reduce rotational latency, increase the rotational velocity
   - to reduce transfer time, increase the rotational velocity, or increase the quantity SB, which represents the number of bytes per track

2. Since 500 sectors fit on each cylinder, disk block 596 resides on cylinder one (the second cylinder).
   - seek distance is $20 - 1 = 19$ cylinders
   - seek time is $3 + 0.025 * 19 = 3.5$ms
   - transfer time is $16.7/50 = .3$ms
   - expected rotational latency is $16.7/2 = 8.3$ms
   - expected service time is $3.5 + .3 + 8.3 = 12.1$ms
   - worst case service time is as above, but with the expected rotational latency replaced by the worst case latency of 16.7 ms, so service time is $3.5 + .3 + 16.7 = 20.5$ms
   - best case service time is as above, but with the expected rotational latency replaced by the best case latency of 0ms, so service time is $3.5 + .3 + 0 = 3.8$ms.

3.
   - SSTF: 74,20,19,400,899
   - SCAN: 400,899,74,20,19 (assuming the initial direction is up)
   - CSCAN: 400,899,19,20,74 (assuming the initial direction is up)

4. Block devices store and retrieve data in units of fixed-size blocks. Character devices supply (or consume) streams of characters, i.e., bytes.

5. **a.**
   - seek time = 0
   - transfer time = $\frac{1}{S}\frac{1}{\omega}$
   - rotational latency = $\frac{1}{\omega} - d$

   service time = $\frac{1}{S}\frac{1}{\omega} + \frac{1}{\omega} - d$

   b. the smallest integer k such that $\frac{k}{S}\frac{1}{\omega} \geq d$, i.e., $k = \lceil dS\omega \rceil$

6. **a.**
   - seek time = $5 + 0.05 * 10 = 5.05$
   - rotational latency = $\frac{1}{2}\frac{1}{\omega}$
   - transfer = $\frac{1}{S}\frac{1}{\omega}$

   service time = seek time + rotational latency + transfer time

   b. This rotational latency given below assumes that $d + t_{seek} <= 1/\omega$. If this is not the case, then the subtracted term should be $d + t_{seek}$ modulo $1/\omega$
   - seek time = $5 + 0.05 * 1 = 5.05$
   - rotational latency = $\frac{1}{\omega} - (d + t_{seek})$
   - transfer = $\frac{1}{S}\frac{1}{\omega}$

   service time = seek time + rotational latency + transfer time

7. DMA is direct memory access, the direct transfer of data between a disk controller and memory without passing through the processor. An interrupt is generated to notify the processor of completion of a DMA operation.

8. **a.** For the first request, $t_{seek} = 100(0.1) + 5 = 15$, $t_{rot} = (1/2)10 = 5$, and $t_{trans} = (1/10)10 = 1$, giving $t_{service} = 21$. For the second request, $t_{seek} = 0$ and t_{rot} and t_{trans} are as for the first request. This assumes that the delay between requests is unknown so that the disk is equally likely to be at any rotational position when the second request arrives. The sum of the service times is 27 milliseconds.

   b. We have $t_{seek} = 100(0.1) + 5 = 15$, $t_{rot} = (1/2)10 = 5$ and $t_{trans} = 2/10(10) = 2$, giving $t_{service} = 22$.

9. The seek plus rotational latency is 40 msec. For 2K pages, the transfer time is 1.25 msec, for a total of 41.25 msec. Loading 32 of these pages will take 1.32 seconds. For 4K pages, the transfer time doubles to 2.5 msec, so the total time per page is 42.5 msec. Loading 16 of these pages takes 0.68 seconds.

10. (a) $10 + 12 + 2 + 18 + 38 + 34 + 32 = 146$ tracks = 730 msec
    (b) $0 + 2 + 12 + 4 + 4 + 36 + 2 = 60$ tracks = 300 msec
    (c) $0 + 2 + 16 + 2 + 30 + 4 + 4 = 58$ tracks = 290 msec

11. Interleaving means skipping sectors when laying data out on a disk so that DMA can occur while the skipped sectors spin beneath the heads. Its purpose is to reduce rotational latency.

12. Video RAM is memory in which addresses correspond to locations on a display screen. It serves as an interface between the CPU and the display.

13. A device driver is software: that part of the OS that is responsible for interacting with a particular device. A device controller is a piece of hardware that controls the device and implements an interface between the device and the rest of the system.

14. Increasing the number of sectors per track will decrease the transfer time since more sectors will pass under the read/write head in a given amount of time. Seek times and rotational latencies are unaffected, so the total service time will decrease

    Increasing the number of tracks per surface will not affect the transfer time or the rotational latency. Assuming the surface itself remains the same size, it is possible to make an argument that it may reduce seek times slightly (since track i and track j will be closer together on the disk with more tracks per surface), but this depends on the sequence of data requests and on how the data is distributed across the disk tracks.

15. An asynchronous device does not require that the CPU block while the device handles a request.

16. **a.** For the first request, the seek distance is 100 cylinders and the seek time is 15 milliseconds. Max rotational latency is 10 milliseconds, expected is 5 milliseconds, and the min is 0 milliseconds (this solution will assume expected time). Transfer time is 5/25 of 10 or 2 milliseconds, for a total expected service time of 22 milliseconds. The second request requires no seek time, but has the same rotational latency (expected) and transfer time as the first, for a total service time of 7 milliseconds. In sum, the two requests require 29 milliseconds.

    b. The first request requires a 15ms seek. Rotational latency is zero, and 10ms a required to transfer the whole track into the buffer, for a total time of 25ms. The second request requires zero time to service since the required data is in the buffer. Total service time for the two requests is 25ms.

17. A serial interface is used to exchange data one bit at a time through a single address or port. A memory-mapped interface uses one address per location (character or pixel) on the terminal screen.

18. **a.** Read Order: $40, 37, 27, 90$
    Distance: $2 + 3 + 10 + 63 = 78$

    b. Read Order: $90, 40, 37, 27$
    Distance: $48 + 50 + 3 + 10 = 111$

    c. For SCAN, the worst case distance is twice the number of disk cylinders (200 for version A, 400 for version B), since for a request at one edge, the heads may need to travel all the way across the disk and back. For SSTF, the worst case is unbounded. Requests may starve if new requests arrive continuously near the currently location of the heads.

19. **a.** Disk head scheduling can reduce seek times by changing the order in which requests are serviced. Track buffering completely eliminates seek times when a request hits the buffer.

    b. Interleaving can reduce rotational latencies by leaving space between logically consecutive sectors, in anticipation of delays between requests. Track buffering can completely eliminate rotational latencies, as above.

    c. Track buffering can eliminate transfer times. (Data must still be delivered from the disk controller to memory, but it need not be read from the platters.

4 File Systems

1. The per-process table is used to ensure that a process cannot access file table entries created for other processes.
2. File systems implement objects (files) of arbitrary size which can grow and shrink over time. Files can have names and other attributes. The file system also controls shared access to a storage device.
3. $10(2^{13}) + ((2^{13})/4)2^{13} + ((2^{13})/4)((2^{13})/4)2^{13}$ bytes
4. Open() and close() are provided so that the costs of locating a file's data and checking access permissions can be paid once, rather than each time a file is accessed. When a UNIX open() call is performed, the file's i-node is cached and new entries are created in the open-file table and in the per-process descriptor table of the process that issued the open().
5. An i-node's reference count keeps track of the number of hard links to that i-node.
6. Random access to file blocks is faster with an indexed layout scheme. Chained layout does not require a separate index.
7. One technique is to use a bit map, another is to maintain a list of the block numbers of free blocks.
8. Fragmentation refers to space that is wasted when files are laid-out on a disk.
9. Directory entries contain a pathname component and an i-number. The i-number is assumed to refer to a file in the same file system as the directory.
10.
```
int translate(string pathname) {
int inum = 0;
int i;
    FOR i FROM 1 TO num_components(pathname) {
        IF is_regular(inum) THEN return(INVALID);
        inum = dsearch(inum,get_component(i,pathname);
        IF inum < 0 THEN return(INVALID);
    }
    return(inum);
}
```
11.
```
int symlinkcount = 0;
int translate(string pathname) {
int inum = 0;
int i;
    FOR i FROM 1 TO num_components(pathname) {
        IF is_regular(inum) THEN return(INVALID);
        inum = dsearch(inum,get_component(i,pathname);
        IF inum < 0 THEN return(INVALID);
        IF is_symlink(inum) DO
            symlinkcount = symlinkcount + 1;
            if (symlinkcount > 8) return(INVALID);
            inum = translate(get_symlink(inum));
            if (inum == INVALID) return(INVALID);
        ENDWHILE
    }
    return(inum);
}
```
12. reference count, ID of file owner, access permissions, direct and indirect pointers to file data blocks, file size, file type

13. Two disk operations are performed, both because of the read() request. The first operation retrieves the second file data block from the disk, and the second operation retrieves the third file data block.

14. lseek(3,0);
    read(3,buffer,1024);
    /* modify buffer here */
    lseek(3,0);
    write(3,buffer,1024);

15.

symlink(/b/y, /a/d/x)	NO - /a/d/x already exists
link(/b/y, /a/d/x)	NO - /a/d/x already exists
symlink(/c, /b/z)	OK
link(/c, /b/z)	NO - /c does not exist
symlink(/a/c/f, /b/f)	OK
link(/a/c/f, /b/f)	OK
symlink(/a/c, /b/g)	OK
link(/a/c, /b/g)	NO - cannot hard link to a directory

16. Hierarchical name spaces allow files to be organized into groups, e.g., by user. This also helps to avoid name conflicts. If the name space is structured as a DAG, users have more flexibility. For example, two users working as a team can place their work in a single shared subdirectory.

17. /, /a, /a/b

18. /, /a, /a/b, /, /d, /d/e

19. 3 blocks: two data blocks plus one pointer (indirect) block

20. 203 blocks: 200 data blocks, plus the single indirect pointer block, plus two double indirect pointer blocks

21. The max number of modified blocks is three: one data block plus two double indirect pointer blocks. The max cost will be paid when $L_G = 110,000$. This is the largest file that can be represented using only the direct and single indirect pointers. The next byte added to the file will cause two double indirect blocks to be allocated, along with a new data block.

22. rename(string oldpath, string newpath) {
    int i;
    int old, oldp, newp;
        old = 0;
        for i from 1 to num_components(oldpath) do
            oldp = old;
            old = dsearch(old,get_component(i,oldpath))
            if (old < 0) return(ERROR)
        endfor;
        newp = 0;
        for i from 1 to num_components(newpath)-1 do
            if (newp == old) return(ERROR)
            newp = dsearch(newp,get_component(i,newpath))
            if (newp < 0) return(ERROR)
        endfor;
        newp = dinsert(newp,get_component(num_components(oldpath),oldpath),old);
        if (newp < 0) return(ERROR);
        old = ddelete(oldp,get_component(num_components(oldpath),oldpath));
        return(SUCCESS);
        }

23.
- advantages:
  - fewer pointers required in index nodes
  - improved performance on sequential file access, since one large block can be retrieved more quickly than several smaller blocks
- disadvantage:
  - poor space utilization because of increased internal fragmentation

24.
- advantage:
  - fewer disk operations are required to access a file's attributes
- disadvantage:
  - hard links (multiple pathnames for the same file) are difficult to implement

25. B - (F MOD B)

26.
```
link(string oldname, string newname) {
int oldi,newpi,i;
    /* find the i-number of oldname */
    oldi = 0;
    for i from 1 to num_components(oldname) {
        oldi = dsearch(oldi,component(i,oldname));
        if ( oldi < 0 ) return(INVALID);
    }
    /* make sure that oldname is not a directory */
    if ( NOT is_regular(oldi) ) return(INVALID);
    /* find the i-number of the parent of newname */
    /* first make sure that newname is not the root, since the root has no parent */
    if ( num_component(newname) < 1 ) return(INVALID);
    newpi = 0;
    for i from 1 to num_components(newname) - 1 {
        newpi = dsearch(newpi,component(i,newname));
        if ( newpi < 0 ) return(INVALID);
    }
    /* insert a new entry in the parent of newname */
    if ( dinsert(newpi, component(num_component(newname),newname), oldi) < 0)
        return(INVALID);
    else
    return(OK);
}
```

27. $2^{13} + 2^{18} + 2^{26}$

28.
- if $0 \leq N < 8$, then one operation is required
- if $8 \leq N < 256 + 8$, then two operations are required
- if $256 + 8 \leq N < 2^3 + 2^8 + 2^{16}$, then three operations are required

29.
- if $0 \leq N < 8$, then one operation is required
- for $i \geq 1$, if $8 + 255(i-1) \leq N < 8 + 255i$, then $i + 1$ operations are required

30.

	single indirect pointer	access permissions	file size	file type	reference count
`link("/a/b","/c")`	-	-	-	R	W
`symlink("/a/b","/c")`	-	-	-	-	-
`open("/a/b")`	-	R	-	- or R	-
`write(d,buf,amount)`	W	-	W	- or R	-
`read(d,buf,amount)`	R	-	R	-	-

31. The global index must hold 1024/4 = 256 pointers, each of which occupies 2 bytes, for a total of 512 bytes. Four sectors are required to hold these 512 bytes.

32. string
showpath(int i) {
    string path,s;
    int i,p;
    path = NIL;
    while (i ≠ 0) {
        p = dsearchByName(i,"..");
        s = dsearchByNumber(p,i);
        path = concat(s,"/",path);
        i = p;
    }
    return(path);
}

33. **seek:** requires no block operations

    write: bytes 10100 through 11099 of the file must be updated. These bytes partially occupy blocks 10 and 11. These blocks must be read in, modified, and written back to the disk. Reading them in requires 3 block reads, since an indirect block must be read, in addition to the two data blocks. Writing them out requires 2 block writes.

    read: bytes 11100 through 12899 must be read. These bytes fall on blocks 11 and 12. Block 11 is already in the cache, so it need not be written. One block read is required for block 12.

34. To create the new file, the root directory must be updated to add an entry for the new file, the free list must be updated to reflect the space used for the new file's header, and the new file's header itself must be created.

    - Directory:
      - read the header for the root directory file (1 read)
      - create new data block (in memory) containing directory entry and write it to the disk (1 write)
      - update the header to reflect new file length, and possibly new timestamp (1 write)
    - Free List:
      - read in data block(s) containing free list (1 read, or more)
      - update free list to reflect new file header and new directory data block (1 write, or more)
      - update file header (optional), which is already in memory (1 write)
    - New File:
      - create new file header in memory and write it to the disk (1 write)

35.

    - Link Count: 4
    - Size: 2
    - Read Timestamp: 1,2,3

- Write Timestamp: 2

36. In general, the blocks may not be contiguous, so three disk read operations will be needed. The final block is only partially used - the answer may assume that it is read in completely or partially. In the first case the answer should be $3(50 + 4) = 162$. In the second case it should be $2(50 + 4) + (50 + 2) = 160$.

37. Since the disk is not full, it should be possible for the file to be laid out as a single extent. This can be read in a single disk read operation requiring $50 + 10 = 60$ milliseconds.

38.

    a. The Seek requires no disk read or write operations. The Write partially updates blocks 5 and 6 of the file. Blocks 0 through 5 are located through the direct pointers in the i-node, while block 6 is located through the single indirect block. A total of three disk reads are required for reading in the two blocks plus the indirect block. Eventually, the updated versions of blocks 5 and 6 need to be written back out to the disk, but this need not happen right away because of the copy-back policy. Similarly, timestamps in the i-node may be updated but this need not be written immediately to the disk. The above assumes that the Write does not cause the file to grow. If the file can grow, additional activity is required to find free space for a new block, to update the free list or free map, and to update the indirect block to point to the newly allocated space.

    b. This is the same as in part (a) except that no indirect block is involved, so only two data blocks (5 and 6) need to be read in. The global index is normally kept in memory and is not updated unless the file grows, in which case it may be written back to the disk.

39. In a log-structured file system, the location of an i-node changes each time it is updated. The i-node map is needed to record the currently location of the i-node. In a non-log-structured system, i-node locations are static and can be calculated from the i-number.

40. In a write-through cache, updates to cached data are reflected immediately in the backing (secondary) store. In a copy-back cache, they are not.

41. For very large files, almost all data blocks are accessed through the double indirect pointer, so consider only those blocks. For a block size of B and a pointer size of p, the total size of the double-indirect data blocks is

$$\left(\frac{B}{P}\frac{B}{P}\right)B = \frac{B^3}{P^2}$$

    So, if B is doubled, the maximum file size will increase by approximately a factor of 2^3, or eight.

42. **a.** The worst case is a file that is just large enough to require one byte of the first double-indirect data block. Such a file would have size

$$10B + \frac{B}{4}B + 1$$

    where the first time accounts for the direct data blocks, the second term for the single indirect data blocks and the third term (1 byte) for the data in the double indirect block. The total internal fragmentation for such a file is

$$(B - 4) + (B - 4) + (B - 1) = 3B - 9$$

    where the two terms $(B - 4)$ account for space wasted in the two almost-empty double-indirect pointer blocks and the $(B - 1)$ term account for space wasted in the last data block of the file, which holds only one byte of file data.

    b. The file system with the global index will require exactly B disk operations to read the data blocks. The other file system will also require B disk operations to read the data blocks. In addition, another disk operation will be needed to read the single-indirect pointer block, and four additional disk operation will be required to read double-indirect pointer blocks. Thus, a total of five extra disk operations are required.

43. **a.** In this case the file system must do many (64) block writes. The total time required, t_{total} is the sum of the service times for each block write.

t_{seek}	10
t_{rot}	5
t_{xfer}	$\frac{8}{64}10 = 1.25$
$t_{service}$	16.25
t_{total}	$64t_{service} = 1040$

b. In this case, only one (large) disk operation is required to write the log segment to the disk. The time for this is

t_{seek}	10
t_{rot}	5
t_{xfer}	$\frac{512}{64}10 = 80$
$t_{service}$	95

44. Note that this solution does not include any error checking.

```
int L = num_components(pathname);
int n = 0;
int i;
for i from 1 to L − 1 do {
    n = dsearch(n,get_component(i,pathname);
}
f = ddelete(n,get_component(L,pathname));
remove_link(f);
```

45. Two reasons are:

- a block may be updated more than one time in the cache before being written to disk, if copy-back is used. This would result in multiple I/O's if write-through was used, but only one if copy-back is used.
- a block may be deleted (from its file) after update and before being copied-back to the disk. Copy-back does no I/O, but write-through needed an I/O for the updated block.

46. **a.** A total of six disk reads will be performed: the first three blocks of the file will be read by calls to Read, and the last three will be read by calls to Write. The latter three reads are needed because each Write call only partially updates a file block. If it is assumed that the i-node has very few direct pointers, additional disk read(s) may be needed for indirect blocks.

b. A total of six writes will be performed - each of the last three blocks of the file will be written to the disk twice. Each disk write is caused by one of the six calls to Write made by the program.

c. If the block size is 500, a total of six disk reads are also needed. However, all six disk reads are caused calls to Read. No disk reads are induced by the Write calls, since each Write completely updates a single disk block.

d. If copy-back is used, a total of three disk Writes will be needed. Each of the last three file blocks is written to disk once. One disk write is caused by a Write system call, which forces a previously-updated block from the cache. The remaining two disk writes are caused by the call to Close.

Solutions:
Final Examination
TDDB 63/12: Concurrent Programming and Operating Systems

Wednesday, 20 October, 1999, Time: 9:00

Instructor: Nancy Reed

Aids

No aids (books, notes, calculators etc.), except English dictionaries, are allowed.

Writing

Please write or print clearly. We will not give credit for unreadable answers. You may write answers in English or Swedish (or a combination of the two).

Results

The results will be posted by 1 November 1999.

Assessment

0 – 19	Fail
20 – 27	3 (Pass)
28 – 34	4
35 – 40	5

or (C programme)

0 – 19	Fail
20 – 31	G (Pass)
32 – 40	VG

In case of questions

Nancy Reed can be reached at 28 26 68 during the examination.

Grading Notation

A check mark ✓ indicates a correct answer or part of an answer. An ✗ indicates an incorrect answer or incorrect part of an answer.

1. Operating System Structures (4 p)

a) What are system calls and how are they used? Justify your answer with an example. (2 p)

Ans: System calls provide the interface between a process and the operating system. System calls are typically written in assembly language and used by assembly-language programmers. Some high-level languages allow system calls to be made directly, and some have been designed to replace assembly language for systems programming. C and PERL implementations include direct system call access. System calls can be roughly grouped into five major categories: process control, file manipulation, device manipulation, information maintenance, and communications.

An example of the use of system calls is when a program needs to read or write to a file. Only the operating system has the permission to access the file, so the program must request the information by using system calls. System calls access privileged information in a controlled manner by the operating system on behalf of programs. Reference Section 3.3, pages 57-66 in the text. Reference NACHOS lab #2 on implementing system calls.

b) What three major factors are used when configuring a computer system to determine the number of and size of different types of storage? Justify your answer by comparing at least 4 commonly used types of storage. (2 p)

Ans: When deciding on the types and amounts of storage in a computer, the three major factors are ***volatility, cost per unit of storage,*** *and* ***speed****. Fig. 2.6 shows the hierarchy of storage devices based on speed and cost. Faster storage costs more. In addition to speed and cost,* ***volatility*** *matters – is the data maintained when the power goes off? The fastest kinds of storage, registers, CPU cache and main memory, are all volatile – the data disappears when the power is turned off.*

A typical computer system will have ***registers*** *– fastest, most expensive, volatile;* ***RAM*** *(main memory) - fast, volatile, expensive;* ***hard disk*** *- slower, cheaper, non-volatile,* ***CD-ROM*** *– slower, cheaper, non-volatile, but portable. Floppy disks, magnetic tape storage, main memory caches, and optical disks are also relatively common. We would like to have all memory as fast as registers or RAM, however, those storage media are volatile so they lose all data when the power is turned off. We need some form of non-volatile storage to keep our programs on when the computer is not in use or if a power failure occurs. This is where hard disks and other non-volatile storage media are useful, and they are also less expensive per amount of storage. Reference Sec. 2.4 on pp. 35-37, including figure 2.6.*

Notes: Security and portability. Magnetic disks, magnetic tapes, and floppy disks are non-volatile, but not "secure" storage media since they are erased when exposed to magnets or physically damaged. CD ROMs are much more secure in that respect, although they are still subject to physical damage. You can also talk about the portability of data (floppy disks, tapes, CDROMs, CD-W, CD-RW, and zip drives and the like), however security and portability are not as important as volatility, speed and cost/unit of storage.

2. Processes and threads (4 p)

a) What happens when a context switch occurs? Diagram your explanation showing what the CPU is doing at each point in time and what each process is doing at each time. (2 p)

Ans: Your diagram should look like figure 4.3 in the text. A context switch is when one process is suspended and another process starts executing (running) on the CPU. The process initially executing

(call it process A), is interrupted or executes a system call. The operating system saves the information about process A's execution in its process control block (PCB), loads the PCB of the process scheduled to execute next (identified by the scheduler), then turns execution over to the next process (call it process B). Process A is idle (waiting) when the OS is copying PCBs and also when process B is executing. Process B is initially idle while process A executes and the PCBs are copied. Then process B executes until an interrupt occurs or a system call is made, and the operating system performs a PCB save for B and a PCB load for the next process. Reference Sections 4.1 and 4.2.3.

b) Describe the concept of a thread. What differences are there between user level threads and kernel level threads? How do these differences affect performance (give an example). (2 p)

Ans: A thread, also called a lightweight process, consists of a program counter, a set of registers, and stack space. It shares with all peer threads its code section, data section, and operating-system resources such as open files and signals. On systems with kernel-level threads, switching among threads is somewhat more time-consuming because the kernel must do the switch via an interrupt. In Solaris 2, kernel-level threads have only a small data structure and a stack, and switching between them does not require changing memory access information and is relatively fast, although not as fast as user-level threads. Context switching between threads is much more efficient than between heavyweight processes because less information must be saved and restored in a switch. A user-level thread is programmed by the programmer and may take advantage of thread libraries provided or interrupt the operating system to switch threads. Reference Section 4.5

Example: If the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns. If a kernel thread blocks, the processor(s) are free to run other kernel threads. Reference your NACHOS labs……

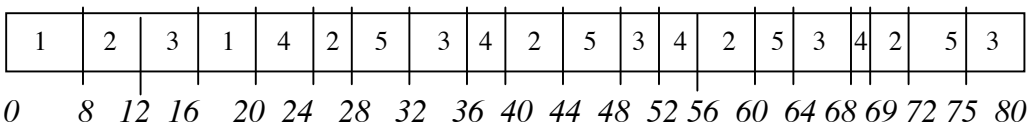
3. CPU Scheduling (6 p)

Consider the following set of processes with arrival times and CPU execution times given in milliseconds. A process with a *larger* priority number has a *higher* priority. State any assumptions that you make.

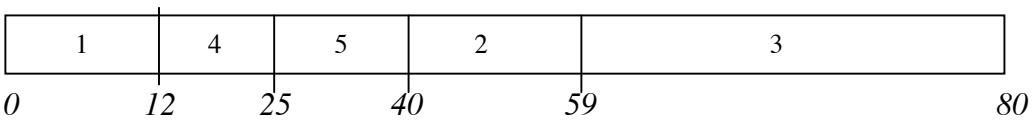
Process	Arrival time	Execution time	Priority
P1	0	12	3
P2	5	19	3
P3	8	21	5
P4	11	13	2
P5	15	15	3

a) Draw and label Gantt charts illustrating the execution of these processes using:

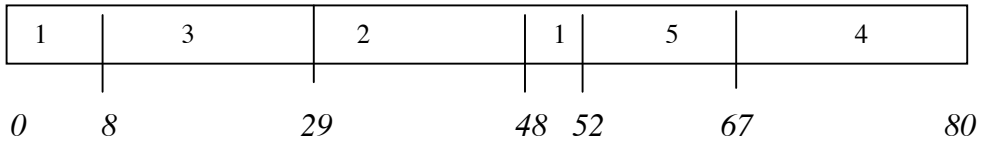
i) RR scheduling with a time quantum of 4 msec. (1 p)



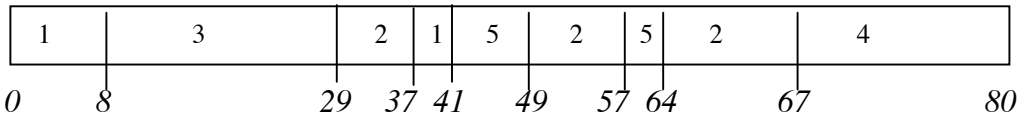
ii) nonpreemptive SJF scheduling. (1 p)



iii) immediately preemptive priority (use FCFS when priorities are equal). (1 p)



iv) (non-immediately preemptive) priority scheduling with a time quantum of 8 msec. (1 p)



b) Calculate the average waiting times for schedules i and ii above, ignoring priority values. Which method gives a better waiting time for this data? (1 p)

i) AWT = (sum of (Finish – arrival – length) for all processes)/ # of processes

= ((20 – 0 – 12) + (72 – 5 – 19) + (80 – 8 – 21) + (69 – 11 –13) + (75 – 15 -15))/ 5 = 197/5 = 39.4 msec.

ii) AWT = ((12 – 0 – 12) + (59 – 5 – 19) + (80 – 8 – 21) + (25 – 11 –13) + (40 – 15 -15))/5 = 97/5 = 19.4 msec.

The second schedule has a smaller average waiting time (as you would expect).

c) For schedules iii and iv above, use each job's priority as its weight and calculate the weighted average waiting time. Which method gives better weighted average performance for this data? (1 p) Grading note: you do not need to simplify the formulas (i.e. do the math).

*Ans: weighted AWT = (sum of ((Finish – arrival – length)*priority) for all processes)/(# of processes)*(sum of priorities)*

*iii) weighted AWT = ((52 – 0 – 12)*3 + (48 – 5 – 19)*3 + (29 – 8 – 21)*5 + (80 – 11 –13)*2 + (67 – 15 -15)*3) / (5*16) = 415/80 = 5.1875 msec.*

*iv) weighted AWT ((41 – 0 – 12)*3 + (67 – 5 – 19)*3 + (29 – 8 – 21)*5 + (80 – 11 –13)*2 + (64 – 15 -15)*3) / (5*16) = 430 /80 = 5.375 msec*

The first schedule has a slightly smaller weighted average waiting time, also expected.

4. Synchronization (5 p)

a) Describe the four constraints Dijkstra placed on *solutions to the critical section problem* and why they are necessary. (2 p)

Ans: There are three major constraints plus one minor constraint. A solution to the critical section problem must satisfy the following three constraints:

- *Mutual Exclusion -- If process P1 is executing in its critical section, then no other processes can be executing in their critical sections. Necessary to ensure mutual exclusion (processes don't clobber each other's data).*
- *Progress -- If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely. Necessary to ensure that processes eventually can enter their critical sections when they need to. Processes in their remainder section have already entered the critical section, so they are not allowed to influence others that are still waiting*
- *Bounded Waiting – There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. This constraint is necessary to prevent starvation.*

Atomic instructions must be available to implement the above – Atomic hardware instructions are available such as test&set, load, and store.

b) The wine-making club problem. (3 p) Consider a wine-making club with 8 members and a warehouse of supplies. For a member to make wine, they need to use: 2 carboys (jugs), 1 yeast lock, sweetened fruit juice, and wine-making yeast. The initial mixing process requires a mixing station, the fermentation process requires 4 weeks to produce wine once all three ingredients have been properly mixed together. The second carboy is needed only at the end of the process to decant the wine off the dead yeast. The warehouse contains 2 mixing stations, 6 carboys (10 liter size), 7 yeast locks, 15 containers of (5 liter size) of sweetened fruit juice, and 20 packages of wine yeast (for 10 liters of wine each). Once a member has finished his/her wine, they all taste the batch before that member starts a new batch.

Write a program that simulates the members of the club making wine. Each member should be represented by a process. Use semaphores for synchronization. Use pseudocode, C++, or Pascal syntax in your solution. Your solution must be free from deadlock, but need not be free from starvation.

Ans: In order to prevent deadlock, one carboy must be saved for the decanting process, but the same one can be used by all members. We can assume that the members order additional re-usable supplies as necessary. Alternatively that can be the first thing each member does before starting to make wine.

Counting semaphores are used for all the re-usable resources

```
semaphore carboy = 5, decant_carboy = 1, yeast_locks = 7,
mixing_station = 2;

semaphore mutex =1;
```

Integers are used for all non re-usable resources

```
int juice 15, yeast = 20;
```

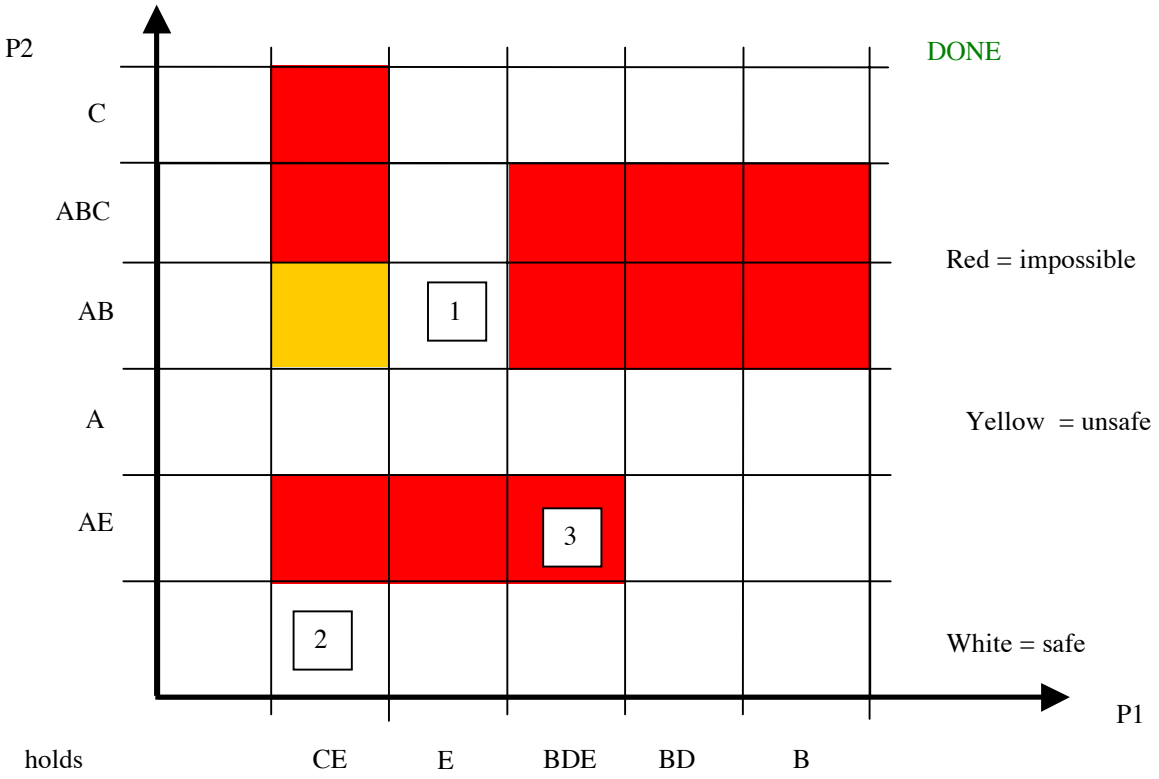
```
void member (int id)     /* each member's process}
{ while (1)  {      /* order juice and yeast here, if necessary*/
     wait(carboy);     /*carboy is the most scarce resource */
     if (juice > 1) and (yeast > 0)
          wait(mutex);
             juice = juice - 2;
             yeast = yeast - 1;
          signal(mutex);
          wait(yeast_lock);
               wait(mixing_station);
               /* mix the juice and yeast in the carboy, add lock */
               signal(mixing_station);
          /* drink any wine available while waiting for 4 weeks */
               /* the wine is done */
               wait(decanting_carboy);
                         /*decant the wine and wash a carboy */
               signal(decanting_carboy);
          signal(yeast_lock);
     /* drink your own wine and offer it to the other members */
     signal(carboy);
     /* drink other's wine */
     }
}

int main(void) {
     parallel {* Executes the statements in parallel */
     member(1); member(2); ... member(8);
     }
}
```

5. Deadlocks (6 p)

a) A resource allocation grid shows illegal, safe, and unsafe states for a sequence of requests (REQ) and releases (REL) of resources for 2 processes. Draw a resource allocation grid like the one below and identify the illegal (forbidden), safe, and unsafe states (assume the Banker's algorithm is used) for 2 processes, P1 and P2, with the following list of requests. All resources are unshareable. State any assumptions you make. (3 p)

P1: REQ{c,e}, REL{c}, REQ{b,d}, REL{e},REL{d}, REL{b}

P2: REQ{a,e} REL{e}, REQ{b}, REQ{c}, REL{a,b},REL{c}



b) Explain how and why you determined the states of squares 1, 2, and 3. (3 p)

*Ans: Assumption: resources in MAX **cannot be requested again** after they have been used and released. The MAX claim for P1 is {A,B,C,E}. The MAX claim for P2 is {B,C,D,E}.*

State 1 is safe because P2 holds B and the remaining MAX2 at that point is {C}. P1 holds E and the remaining MAX1 is {B,D}. There can be no deadlock because there can be no circular wait between P1 and P2. We can give P2 all of its claim, then it will finish, and then P1 can finish.

State 2 is safe. All states in the bottom row and in the left column are safe because one process has not received any resources. If either row 1 or column 1 states are followed, the processes execute in sequence, not interleaved, so no deadlock can occur.

State 3 is impossible because both P1 and P2 would be holding E, which is an unsharable resource.

6. Virtual Memory (6 p)

a) Show the sequence of frames put into main memory with following reference string. Which gives the lowest fault rate on this string? Why? (4 pt)

	5	**4**	**8**	**2**	**7**	**4**	**5**	**9**	**1**	**8**	**7**	**4**	**5**	**2**	**7**

i. Using 3 available frames and the OPT algorithm.

Ans:	**5**	5	5	5	5		*	**9**	**1**	**8**			**5**	5	
		4	4	4	4	*		4	4	4		*	4	**2**	
Total:10			**8**	**2**	**7**			7	7	7	*		7	7	*

ii. Using 3 available frames and the FIFO algorithm.

Ans:	**5**	5	5	**2**	2	2	**5**	5	5	**8**	8	8	**5**	5	5
		4	4	4	**7**	7	7	**9**	9	9	**7**	7	7	**2**	2
Total: 15			**8**	8	8	**4**	4	4	**1**	1	1	**4**	4	4	**7**

iii. Using 4 available frames and LRU.

Ans:	**5**	5	5	5	**7**		7	7	**1**	1	1	1	**5**	5	
		4	4	4	4	*	4	4	4	**8**	8	8	8	**2**	
			8	8	8		**5**	5	5	5	**7**	7	7	7	*
Total: 13				**2**	2		2	**9**	9	9	9	**4**	4	4	

iv. Using 5 available frames and FIFO.

Ans:	**5**	5	5	5	5		*	**9**	9			9	9	9	**7**
		4	4	4	4	*		4	**1**			1	1	1	1
			8	8	8			8	8	*		**4**	4	4	4
				2	2			2	2			2	**5**	5	5
Total: 11					**7**			7	7		*	7	7	**2**	2

OPT in part i) is the best, although FIFO comes close because it has more frames (5 instead of 3) to work with (11 vs. 10 page faults). Grading note: "OPT is optimal" is not an adequate explanation because it depends on the number of frames available – OPT is provably optimal only when compared to other algorithms with the same number of page frames, and FIFO above has more frames.

b) What is meant by the term *thrashing* in a virtual memory system? Describe at least two ways to detect and stop thrashing if it occurs. (2 pt)

Ans: Thrashing occurs when many processes are running concurrently on a system and at least some of those processes generate a very large number of page faults, although they don't make much progress. The processes need more page frames than they are currently allocated or the scheduling

policy is allowing processes to "steal" frames from other processes, causing a circular shift of page frames between processes.

Detection: The CPU utilisation is very low and the page swap rate is very high when thrashing occurs, so the degree of CPU utilization vs. paging can be monitored. The working-set model can be used to estimate the number of frames a process needs to run efficiently, to detect thrashing.

Recovery: If thrashing is detected, it can be stopped by reducing the amount of multiprogramming in the system, i.e. by suspending some of the processes so the others get more page frames and then those processes can make progress and finish. Then the suspended processes will be given time.

7. File Systems (6 p)

a) What is a file? How does a computer identify the type of contents that are in a file? Describe at least two methods used on typical Unix systems and one method used on DOS systems. (3 p)

Ans: A file is a collection of related information that can be stored as bits by a computer. In Unix, there are also special files like directories, pipes and device drivers. The contents of a file can be of many different types – executable programs (binaries), plain text, word processing documents, large structured databases, etc. Unix identifies the type of contents of a file using magic numbers (bit sequences) for particular executable programs as well as using the command name for shell script files, i.e. #!/bin/sh. DOS uses the extension part of the name of a file to indicate what type of file it is. For example, files that end in .EXE or .COM are binary executables, those ending in .BAT are batch files (similar to Unix shell scripts), those ending in .TXT are plain text files.

b) Consider a file currently consisting of 50 blocks of information. Assume that the file control block (and index block, in the case of indexed allocation) is already in memory. Calculate the number of disk I/O operations required for **contiguous**, **linked**, and **indexed** allocation strategies to make the following changes to the file. In the contiguous case, you may assume there is no space to grow in the beginning, but there is room to grow at the end. Assume that the new information to be added to the file is stored in memory. (3 p)

	Contiguous, I/O	Linked , I/O	Indexed, I/O
i) add beginning block	***50 R + 51 W = 101***	***0 R + 1 W = 1***	***0 R + 1W= 1***
ii) add ending block	***0 R + 1 W = 1***	***1 R +2 W = 3***	***1 W + 0 R = 1***
iii) remove middle block	***25 R + 25 W = 50***	***25 R + 1 W = 26***	***0 R + 0 W = 0***

i) Add a block at the beginning of the file.

Ans: Contiguous: 1 block Write for new information plus 50 reads & 50 writes to shift the current information in the file down one block.
Linked: 1 block to write the new information, then update pointers (already in RAM, so no reads/writes).
Indexed: Write the new information(1 block), then update the index pointers in RAM.

ii) Add a block at the end of the file.

Ans: Contiguous: Write the new information, 1 block, update file header (in RAM).

Linked: Read in the last block, write last block with new pointer, write new block at the end. Note, linked file schemes typically have a pointer to the first and the last block in the file, which makes accessing the last block very easy, no reads necessary. Indexed: Write the new block and update links in RAM.

iii) Remove a block from the middle of the file.

Ans: Contiguous: To delete position 25, read all blocks after # 25 and write them back one place closer to the front. Update file size in RAM.
Linked: Read and follow links to position 25, then write block 24 to link to former block 26.
Indexed: No reads or writes necessary, only modification of links in RAM.

Reference Exercise 1 in Lecture 10 and Section 11.2 in your text.

8. Protection and security (3 p)

a) What is authentication? Describe two fundamentally different schemes that can be used for authentication. (2 p)

Ans: Authentication is the process of determining if a user is who he/she claims to be. Generally authentication is based on of three types of items:

- *Something in the user's possession (key or card)*
- *Something the user knows (password or user identifier)*
- *Something the user is - an attribute (fingerprint, retinal scan, signature, movements)*

b) What is a computer virus and how are they spread? What is different about macro viruses compared to other viruses?(1 p)

Ans: Computer viruses are a threat to computer security. They infect other, usually legitimate executable files with their code and can do a great deal of damage –for example sometimes deleting entire file systems (erasing a hard disk) or writing information into the EPROMs of the system so that it will no longer boot or operate. They are spread to other files when a file containing their code is executed. This usually happens if you loan or borrow diskettes from someone or if you download files from a bulletin board or network (some other system). Since executable files run on only one type of computer and operating system, the viruses are also limited to the hardware and operating system environment they are written for.

Macro viruses are different in that they infect what we consider to be "data files" – Microsoft office documents and spreadsheets rather than "executable files". These viruses are written in the macro language used by MS Word, Excel, etc. The other major difference with macro viruses is that they are not limited to one type of CPU or operating system as are most other viruses. The same macro virus can infect both a PC and a MAC if they are both using MS office programs!

COMP 300E Operating Systems
Fall Semester 2011
Final Examination SAMPLE QUESTIONS

Disclaimer: these sample questions should not be used as predictors for the questions in the final exam. These questions are biased towards virtual memory and page replacement, but the final exam will consist of a balanced selection of questions from each chapter. You should also carefully review the sample questions provided on the lecture notes page, as well as the sample and actual midterm exam papers.

1. Single-Choice (Each question has a single correct answer) questions and brief Q&As.

* Which of the following is NOT a valid source of cache misses?
A. Compulsory
B. Capacity
C. Conflict
D. Trashing

ANS: D

* Which of the following statement is true for a Fully Associative Cache?
A. No conflict misses since a cache block can be placed anywhere.
B. More expensive to implement because to search for an entry we have to search the entire cache.
C. Generally lower miss rate than a fully-associate cache.
D. All of the above

ANS: D

* Which of the following statement is true for write-through cache and write-back cache?
A. A write-through cache will write the value back to memory when it changes. (For example if the value 'x' is stored in the cache and I increment it by one, then the incremented value is written to the cache and to physical memory as well.)
B. A write-back cache will only write the value back to memory when the cache block is evicted from the cache.
C. All of the above

ANS: C

* Compare fully-associative cache and direct-mapped cache: _____ cache has lower miss rate; _____cache has smaller hit time?
A. fully-associative, direct-mapped
B. direct-mapped, fully associative

ANS: A
(A fully-associative cache has lower miss rate, since it eliminates conflict misses. A direct-mapped cache has smaller hit time, since cache lookup is a simple table index operation, while a fully-associative cache needs to search through all cache blocks to find a match.)

* True or False: TLBs are typically organized as a directly-mapped cache to maximize performance.
ANS: False
(TLB is usually fully-associative, but can also be set-associative, to minimize miss rate (as compared to fully-associative). Since TLBs are typically very small, the hit time can still be very fast despite the associativity.)

* For a single-level page table system, with the page table stored in memory without a TLB. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?
A. 600 nanoseconds
B. 200 naboseconds
C. 400 nanoseconds
D. can't say
Ans: C

* For a single-level page table system, with the page table stored in memory. If the hit ratio to a TLB is 80%, and it takes 15 nanoseconds to search the TLB, and 150 nanoseconds to access the main memory, then what is the effective memory access time in nanoseconds?
A. 185
B. 195
C. 205
D. 175
ANS: B. 0.8*(150+15)+0.2*(300+15)=195 $(m + s)\, h + (2m + s)(1 - h)$

* In a 64 bit machine, with 256 MB RAM, and a 4KB page size, how many entries will there be in the inverted page table?
A. 2^16
B. 2^50
C. 2^14
D. None of the above
Ans: A Total physical frames is 256MB/4KB=64K=2^16, which is number of PTEs in IPT.

* Consider the segment table: What are the physical address for the following logical addresses (Segment ID, Segment Offset) :
a. 0,430
b. 1,10
c. 1,11
d. 2,500

Segment	*Base*	*Length*
0	**219**	**600**
1	**2300**	**14**
2	**90**	**100**
3	**1327**	**580**
4	**1952**	**96**

ANS:
A. 219+430 = 649.
B. 2300+10=2310.
C. 2300+11=2311
D. Illegal address since size of segment 2 is 100 and the offset in logical address is 500.

* In which of the following operations, the OS scheduler is invoked?
A. Process requests for I/O.
B. Process finishes execution.
C. Process finishes its time slot.
D. All of the above A through C
E. None of the above A through C
ANS: D

* What happens if the time slice allocated in a Round Robin Scheduling is very large? And what happens if the time slice is very small?

ANS: If time slice is very large, it results in FCFS scheduling. If time slice is too small, the processor through put is reduced, since more time is spent on context switching.

* What are the two principles of locality that make implementing a caching system worthwhile?

ANS: Spatial Locality – (If I access a block of data, I am likely to access blocks of data next to it in the future. A simple example is sequentially reading through an array. In this scenario, a caching system with a large block size takes advantage of spatial locality because when I read the first element, I read in seven additional elements so that the next seven reads are from the cache and therefore fast.)
Temporal Locality – (If I have accessed a block of data before, I am likely to access this block of data again. A simple example is if I have a piece of code that constantly references/updates a global counter.)

* True or False: A direct mapped cache can sometimes have a higher hit rate than a fully associative cache with an LRU replacement policy (on the same reference pattern).

ANS: True. A direct mapped cache will do better than LRU on a pattern like ABCDEABCDE… if the cache size is one entry smaller than the total number of items in the pattern (e.g., four cache entries). LRU will miss on every access, while a direct mapped cache will only miss on the two entries that map to the same cache entry.

* True or False: Virtual memory address translation is useful even if the total size of virtual memory (summed over all programs) is guaranteed to be smaller than physical memory.

ANS: True. It provides protection between different processes, it doesn't require that each program's addresses be contiguous.

* What type of file access pattern exploits spatial locality?

ANS: Spatial locality is accessing a location that is close to or next to recently accessed location. Sequential access of a file exploits spatial locality.

* Which component of disk access time is the disk scheduling algorithm trying to minimize?

ANS: The disk scheduling algorithm is attempting to minimize overall time wasted to moving the disk arm (i.e. it optimizes seek time).

* Name at least two ways in which the *buffer cache* is used to improve performance for file systems.

ANS: 1) Reads can be from cache instead of disk
2) Allow delayed writes to disk, thereby permitting better disk scheduling an/or temporary files to be created and destroyed without even being written to disk.
3) Used to cache kernel resources such as disk blocks and name translations

* Suppose a new process in a system arrives at an average of six processes per minute and each such process requires an average of 8 seconds of service time. Estimate the fraction of time the CPU is busy in a system with a single processor.

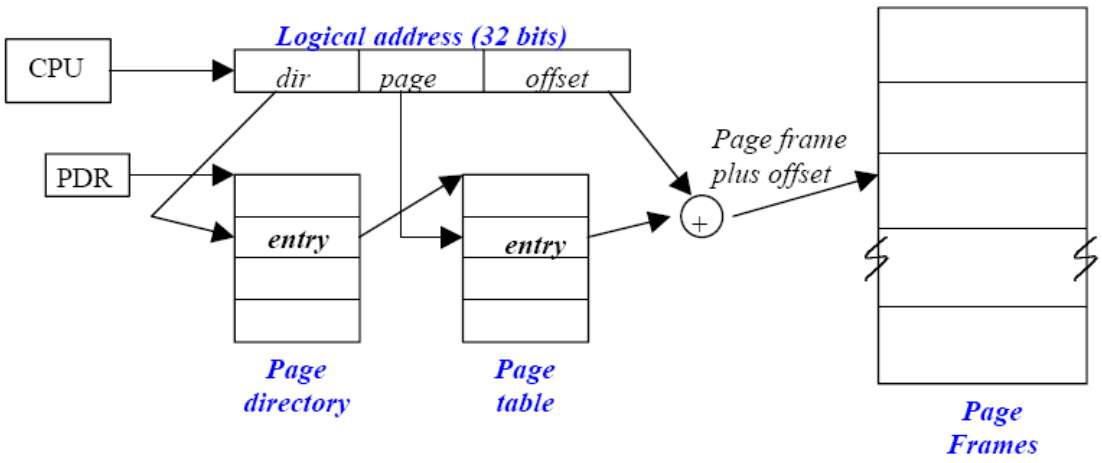
ANS: Given that there are on an average 6 processes per minute. So the arrival rate = 6 process/min. i.e. every 10 seconds a new process arrives on an average.
Or we can say that every process stays for 10 seconds with the CPU
Service time = 8 sec.
Hence the fraction of time CPU is busy = service time / staying time
= 8 / 10
=0.8
So the CPU is busy for 80% of the time.

* If the system does not have enough memory to contain all the processes' working sets and is thrashing, and the page replacement algorithm is CLOCK, then does the clock hand move quickly or slowly? Why?

ANS: Quickly. Since all pages are hot, so the R bits of all pages are continuously set to 1, the clock hand may come around multiple cycles to find a victim page. And there are a lot of page faults, so the clock hand needs to find victim page to replace very often.

* Suppose that we have two-level page tables. Each page is 4KB pages and each Page Table Entry (PTE) is 4 bytes. Suppose we want each page table to occupy exactly one memory page. What is the format of a 32-bit virtual address?

ANS: 4 KB = 2^12 bytes, which means page offset is 12 bits to address a specific byte within the page. This leaves us with 20 bits we need to allocate. Since each page is 4 KB and each PTE is 4 bytes, this means that we can fit 4KB/4 bytes=1024 (2^10) PTEs on a page. Since we want each page table to occupy exactly one memory page, each page table should contain 2^10 PTEs. This means that we can use 10 bits to represent each level in the page table.
Thus, the breakdown is as follows:
10 bits to reference the correct page table entry in the first level.
10 bits to reference the correct page table entry in the second level.
12 bits to reference the correct byte on the physical page.
(Note I will not ask you to draw such a figure in the exam.)



* Multi-Level Page Tables
(1): Suppose we have a memory system with 32-bit virtual addresses and 4 KB pages. If the page table is full, i.e., there are no null pointers in the page table hierarchy, show that a 20-level page table consumes approximately twice the space of a single level page table.

ANS: 4 kilobytes = 2^12 bytes, which means page offset is 12 bits to address a specific byte within the page. This leaves us with 20 bits we need to allocate. We have 20 bits to work with and a 20-level page table, hence there is one-bit in the virtual address for each level of the 20-level page table. Each level of page table consists of 2^1=2 entries (where a '0' bit

references the first entry while a '1' bit references the second entry). The total number of page tables in this implementation is therefore:
2^0 + 2^1 + ... + 2^19 = 2^20 - 1
Since each table has two entries, this means that we have a total of 2^21 - 2 entries.
A single-level page table on the other hand, has 2^20 entries, meaning that there are a total of 2^20 entries in such an implementation. Therefore the 20-level page table consumes approximately twice the space when full.

(2) Show that the above is not necessarily true for a sparse page table, where the process memory size is small, and many pointers in the page table hierarchy are null.

ANS: Imagine if we only have 1 page of physical memory allocated to a process. This means that we have only 20 tables, since we just need one entry per level of indirection. Each table has 2 entries, so we have a total of 40 entries for our page table implementation.
In the single-level case, in the sparse memory scenario, we still have 2^20 entries allocated, meaning that the 20-level page table is theoretically feasible for sparse memory usage (although 20 memory accesses to determine a translation is still expensive and therefore not practical).

* Page replacement
For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:
A B C D E F C A A F F G A B G D F F
Indicate in the following table which pages are mapped to which physical pages for each of the following policies. *If there is no page fault at that time-step, leave the column blank*. Here MIN refers to OPT (optimal policy)

Access→		**A**	B	C	D	E	F	C	A	A	F	F	G	A	B	G	D	F	F
FIFO	1	A				E									B				
	2		B				F										D		
	3			C					A									F	
	4				D								G						
MIN	1	*A*															*D*		
	2		*B*														*D*		
	3			*C*									*G*				*D*		
	4				*D*	*E*	*F*												
LRU	1	*A*				*E*							*G*						
	2		*B*				*F*										*D*		
	3			*C*											*B*				
	4				*D*			*A*										*F*	

For MIN, we accepted the final D in any of the first three pages.
Grading: 2 pts. each for MIN and LRU; -1 for each error.

* For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:
A B C D E A A E C F F G A C G D C F

Indicate in the following table which pages are mapped to which physical pages for each of the following policies. Assume that a blank box matches the element to the left. We have given the FIFO policy as an example.

Access→		**A**	B	C	D	E	A	A	E	C	F	F	G	A	C	G	D	C	F
FIFO	1	A				E									C				
	2		B				A										D		
	3			C							F								
	4				D								G						
MIN	1	*A*																	*F*
	2		*B*			*E*					*F*		*G*						*F*
	3			*C*															*F*
	4				*D*														*F*
LRU	1	*A*				*E*								*A*					*F*
	2		*B*				*A*						*G*						
	3			*C*															
	4				*D*						*F*						*D*		

Any one of these

* Consider a virtual memory architecture with the following parameters:

Virtual addresses are 48 bits. Hence the architecture has virtual address space of 64 Terabytes (TB), i.e., it allows a maximum of 64TB physical memory (RAM). (1TB=10^12 =2^40bytes)

The page size is 32KB.

The first- and second-level page tables are stored in physical memory.

All page tables can start only on a page boundary.

Each second-level page table fits exactly in a *single* page frame.

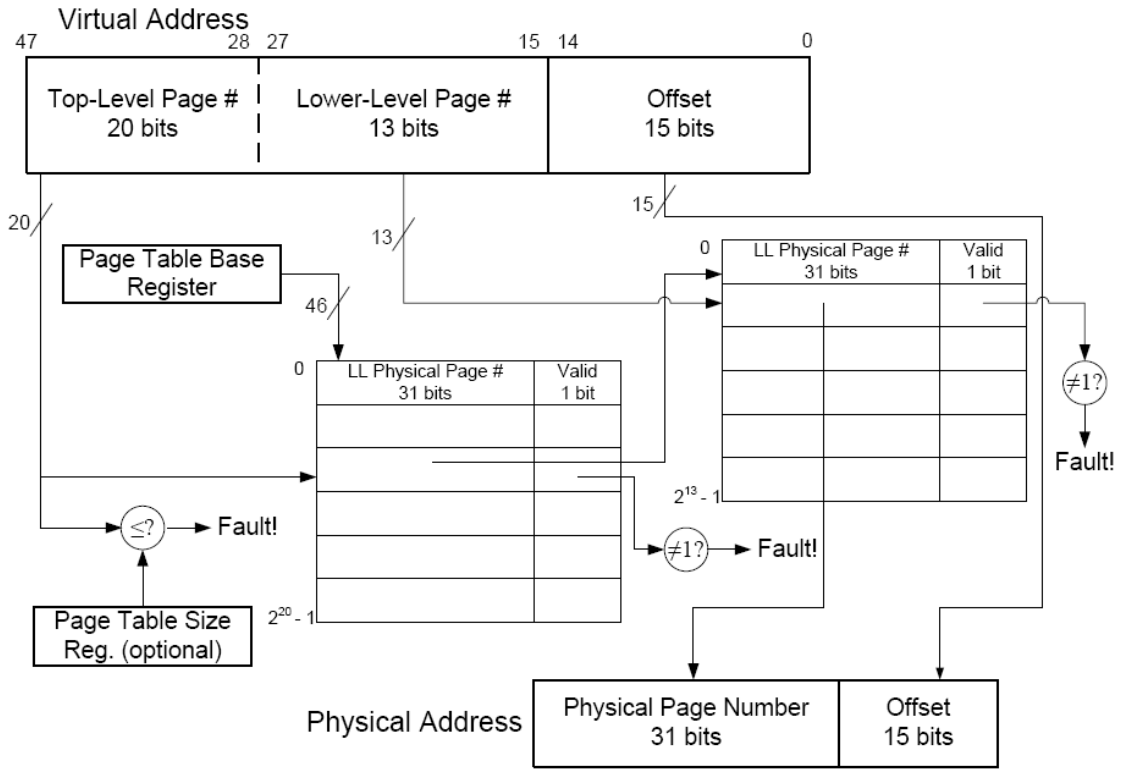
Each Page Table Entry (PTE) is 32 bits, consisting of 31-bit PPN and 1 valid bit, for both 1st and 2nd-level page tables. Assume there is a single valid bit for each PTE and no other extra permission, or dirty bits.

Draw and label a figure showing how a virtual address gets mapped into a real address. You should list how the various fields of each address are interpreted, including the size in bits of each field, the maximum possible number of entries each table holds, and the maximum possible size in bytes for each table (in bytes).

ANS: The page size is 32KB→Page offset is 15 bits.
A 32 KB page can thus hold 8K 4-byte PTEs, and each second-level page table fits exactly in a single page frame, hence each 2nd level page table has 8K entries. So we should allocate 13 bits (2^13=8K) to the 2nd level page table, and 20 bits (48-15-13) to the 1st level page table.

The first-level page table has 2^20 (1 million) entries, each of which is 32 bits (31-bit PPN plus valid bit, or 4 bytes), so it has a maximum size of 4 MB.

(Note I will not ask you to draw such a figure in the exam.)


Virtual Address
47
28
27
15
14
0
Top-Level Page #
20 bits
Lower-Level Page #
13 bits
Offset
15 bits
20
13
15
Page Table Base
Register
46
0
LL Physical Page #
31 bits
Valid
1 bit
0
LL Physical Page #
31 bits
Valid
1 bit
≠1?
Fault!
2^13 - 1
≤?
Fault!
≠1?
Fault!
Page Table Size
Reg. (optional)
2^20 - 1
Physical Address
Physical Page Number
31 bits
Offset
15 bits

University of Rochester **Computer Systems (CSC2/456)**

Final Exam

Amy Murphy

6 May 2003

Read before beginning: Please write clearly. Illegible answers cannot be graded. Be sure to identify all of your answers in your blue book (e.g., 1a, 2b, etc). All questions are worth 10 points, numbers in parenthesis indicate relative point values.

While I have tried to make the questions as unambiguous as possible, if you need to make any assumptions in order to continue, state these as clearly as possible as part of your answer. In the name of fairness, I would like to avoid answering questions during the exam.

CSC256: Students enrolled in 256 must choose **6 of the 8** questions to answer. For up to 10 extra credit points, you may answer **one** of the other 2 questions, indicating clearly on the front of the bluebook which question is to be counted as extra credit. If you do not specify, I will assume the first 6 in your bluebook are for *normal* credit and the next one (if any) is *extra*. If you answer all 8 questions, only 7 of them will be counted toward your grade. Grades will be assigned 0-60 (with 10 possible extra credit).

CSC456: Students enrolled in 456 must choose **7 of the 8** questions to answer. For extra credit, you may answer the 8th question, but you must clearly identify which question is to be counted as *extra*. If you do not indicate any, I will assume the 8th answer (if any) is *extra*. Grades will be assigned 0-70 (with 10 possible extra credit)

1. **Short answer.** Your responses for each part should be at most 3-4 lines of text:

   (a) (3) What is the difference between a user-level instruction and a privileged instruction?

   A privileged instruction can only be invoked when the hardware is running in privileged (a.k.a. kernel) mode. Examples of privileged instructions include those that move data in and out of protected device registers.

   (b) (3) What is the difference between a network operating system and a distributed operating system?

   A distributed operating system presents a single system image to many hosts in an environment, and remote operations are hidden from the users. A network operating system, while it still allows access to remote hosts, leaves the user acutely aware of the remote-ness of any operation issued.

   (c) (4) Name two positive motivations for building distributed systems and two problems they introduce.

   Positives: speed, support inherent distribution, reliability, incremental growth, resource sharing. Negatives: overhead of communication and synchronization, consistency management, software complexity, increased security problems

2. **Scheduling.** I have just invented a new scheduling algorithm that I claim gives the highest priority to processes that have just entered the system, but is fair to all processes. The algorithm works like this: There are two queues, one for *new* processes and one for *old* processes. When a process enters the system, it is put at the end of the *new* queue. After 2 milliseconds on the *new* queue, whether a process has been scheduled or not, it is moved to the end of the *old* queue. When it is time to schedule a process, the system schedules the process at the head of one of the queues, alternating between the two queues. Each process runs to completion before the next process is scheduled. Assume that processes enter the system at random times and that most processes take much longer than 2 milliseconds to execute.

   (a) (4) Does this algorithm give the highest priority to new processes? Explain your answer.
   This algorithm does not give the highest priority to new processes. There are several reasons. First, even if executing processes took much less than 2 milliseconds to execute, the scheduler would alternate between "new" processes and "old" processes, giving equal priority to both. Second, given that executing processes take much longer than 2 milliseconds to execute, most "new" processes will not get scheduled during their 2 milliseconds on the "new" queue and will then drop to the bottom of the "old" queue without ever having been given any priority. Now they have to wait for processes older than them to execute, and processes newer than them to execute.

   (b) (3) Is this algorithm starvation free? Explain your answer.
   Yes. Because the scheduler alternates between the two queues, every job that is not executed from the "new" queue eventually ends up in the "old" queue and from there, eventually reaches the head of the queue and is executed.

   (c) (3) Discuss whether this algorithm is fair to all processes. By "fair" we mean every process has a wait time approximately equal to the average wait time, assuming all processes have close to the same execution time.
   No, it is not fair to all processes. Some lucky "new" processes will get to execute when they reach the top of the "new" queue, while some will drop to the bottom of the "old" queue and have to wait much longer to execute. These unlucky processes will have to wait for all of the processes older than them to complete, and will have to wait for many processes younger than them to complete as well.

3. **Synchronization.** Suppose a program has three threads Thread_1, Thread_2, and Thread_3, and a shared counter, `count`, as shown below:

```
int count = 10;
Semaphore Lock = 1; // initial value is 1
```

```
Thread1(...)
{
// do something
Lock.Wait();
count++;
Lock.Signal();
}
```

```
Thread2(...)
{
// do something
Lock.Wait();
count--;
Lock.Signal();
}
```

```
Thread3(...)
{
// do something
Lock.Wait();
printf(``$d'', count);
Lock.Signal();
}
```

   (a) (4) What are the possible outputs of this program? Hint: there is more than one answer, and you must provide them all.
   9, 10, 11

   (b) (3) Define a race condition.
   A race condition is when it is possible to observe the order in which events in different processes occur, and that order is not constrained by synchronization.

(c) (3) Does this process suffer from a race condition? Justify your answer.

A race condition does exist here because the output can be different for different orderings of the program.

Note: if your definition of a race condition addresses only the final result of the data, then the answer to this part of the answer could be "no" because the final result of the variable count is always 10.

4. **Caching.**

(a) (3) Define spatial locality and temporal locality.

Spatial locality *means the access to a data item makes near-future accesses to spatially nearby data items more likely.* Temporal locality *means the access to a data item makes another access to the same data item in the near future more likely.*

(b) (3) Disk manufacturers added a full track buffer to hard drives to improve program performance. Which of spatial or temporal locality most motivated this? Explain your answer.

Spatial locality. Access to one data block makes near-future accesses to other blocks on the same track more likely. Additionally temporal locality is already exploited by the operating system's file buffer cache

(c) (4) Hardware caches in the memory hierarchy are other performance enhancing techniques. Explain how/why spatial locality and temporal locality contribute to this performance enhancement.

The concept of caching recently accessed data items is motivated by temporal locality. The fact that a cache line (32 or 64 bytes) is usually larger than what an instruction needs is motivated by spatial locality.

5. **Paging.** Consider a reference string 1,2,3,4,2,5,6,2,3,2,1,6,7; and a system with only 4 frames, pure demand paging, and all frames initially empty.

(a) (2) How many page faults would occur with a FIFO replacement scheme? What are the identities of pages in the frames when the reference string has completed?

10 page faults, [2 3 1 7]

(b) (2) How many page faults would occur with a perfect LRU replacement scheme? What are the identities of pages in the frames when the reference string has completed?

9 page faults, [2 1 6 7]

(c) (3) Describe one possible implementation scheme for LRU.

A linked list: always evict the head page, new page join at the tail, the hit page is relocated to the tail.

Associate a timer with every page. Update it when the page is accessed. To evict a page, find the timer with the oldest value.

(d) (3) Would increasing the number of frames always decrease the number of page faults for a particular reference string for FIFO? for LRU? Why or why not? You need not provide "proofs", just an explanation.

FIFO: no. Belady's anomaly states that it is possible to have more resources, and worse access behavior.

LRU: yes. Belady's anomaly does not apply to stack based algorithms.

6. **Memory Management.** LRU can be thought of as an attempt to predict future memory access patterns based on previous access patterns (i.e. if a page has not been accessed for a while, it is not likely to be be referenced again soon). Another idea that some researchers have explored is to record the memory reference pattern from the last time the program was run and use it to predict what it will access next time.

   (a) (3) For what kinds of programs will this be most effective?

   Programs that repeat the same access pattern every time they run, i.e. deterministic programs in which the sequence of memory operations does not depend on the values of program inputs. This is most commonly true for scientific, mathematical applications.

   (b) (3) In what kinds of programs might this lead to worse performance?

   Programs that exhibit different access patterns on different inputs, or even on the same input (e.g., some non-deterministic programs).

   (c) (4) Why would this be difficult to implement in an operating system? Give at least two reasons.

   We need some mechanism to collect this information. We need to store this information, and retrieve it when a program is run again. We need to change the page replacement scheme to work with this collected information.

7. **File systems.**

   (a) (4) A file system must keep track of free blocks on disk. Name two schemes for doing this, and one advantage of each.

   Linked list of free pages supports very fast deletion: we can link a whole file onto the free list in constant time.

   Bit map facilitates spatial locality optimizations in disk layout. It is also the most effecient in terms of memory inside the OS required to find the free pages.

   The list of pages with free page references in each is also space-efficient, particularly for disks that are mostly full (the bit map takes space proportional to the size of the entire disk). Also, free page insertion and deletion operations exhibit high locality under this scheme.

   (b) (3) Consider a very large file: the maximum size file supported by a particular system. Describe the process to read the last byte of that file if it is stored in a FAT file system. Assume you already have the directory entry cached.

   From the directory entry, you will find the index of the first block. Looking up this entry in the FAT will give you the next block index. Keep following this chain until the block you want is reached.

   (c) (3) Describe the process to read the last byte of that file if it is stored in a UNIX inode-like system where the inode contains 10 direct block pointers, one single indirect pointer, one double indirect pointer, and one triple indirect pointer. Assume you already have the inode cached.

   From the inode, look up the triple indirect pointer. Read the indicated block off the disk, and look at the last pointer in this block. This points to another indirect block. Read that block, and the last pointer in this block. This points to another indirect block. Read the last pointer, and it will get you to the last block of the file.

8. **Unix File Systems.**

(a) (3) One of the changes FFS (Fast File System) made over the "classic" Unix file system was to place the inodes at multiple locations on the disk, as opposed to placing all inodes at the front. Why did this (generally) increase file system performance?

Co-locating the data blocks and their corresponding inodes in the same track reduces the need for disk seeks.

(b) (3) FFS tries to achieve the advantages of both of large and small block sizes. Briefly describe the technique by which this is achieved, and state one of its disadvantages.

FFS introduced the ability to create up to 16 block fragments per block. This allowed data that was not the size of a full block to actually share a block with information from other files. The main disadvantage is increased overhead to maintain fragment information, and complexity of implementation of the block allocation scheme: e.g., balancing the amount of data copying with the number of fragments appearing in a file. It is worth noting that the importance of this technique has, arguably, declined with increases in disk capacity. Though block sizes have increased, disk capacity has increased much more, so the internal fragmentation due to under-full blocks has declined as a percentage of total capacity.

(c) (4) What was the motivation for LFS (Log-based File System), and why does it achieve better write performance than FFS when there are many random writes of a large file?

The motivation is that when disk caches get large enough, most reads can be served from memory while only writes go to the disk. Treating random writes as log appends essentially converts random disk accesses to sequential accesses, which reduces disk seek and rotational delays.

There are seven questions on this exam, each worth 20 points. All parts of a question are of equal weight, except as noted. The question on which you score the minimum number of points will be discarded; the question on which you score the next-to-minimum number of points will be treated as extra credit. The remaining five questions will be summed to generate your score out of 100.

Write on blank sheets of paper, not on the exam. Write your name clearly at the top of each page. Fasten the pages with a staple. Read the entire exam before you begin writing. If you make any assumptions not stated in the question, write them down as part of your answer. Ask questions if anything is unclear.

1. (20 points) Paged virtual memory

Consider a memory management system that translates 16 bit virtual addresses to 24 bit physical addresses. These addresses are byte addresses; the memory consists of 16 bit words. The system uses a two-level page table, with a 4-bit first level page number, a 4-bit second level page number, and an 8-bit offset within the page.

a. What is the maximum number of physical memory frames that can be addressed by this memory management system?

2^4 first level page table entries x 2^4 second level page table entries = 2^8 frames

b. Explain the circumstances under which the pages tables in this system will take up less memory than a single-level page table with an 8-bit page number and an 8-bit offset. It may be useful to give an example, but this is not required for full credit.

If most second page tables are empty, they need not be represented by this system, and thus take up no memory.

The space taken is 16(1 + n) where n is the number of non-empty second level page tables. This system takes less memory when 16(1 + n) < 64, ie when n < 3.

2. (20 points) Processor scheduling to minimize page faults

Consider a processor scheduling system for preemptive multitasking that is based on a multi-level feedback queue.

- There are three queues, #1 being the highest priority and #3 being the lowest priority.
    - The scheduler will run processes in queue #1 first
    - If queue #1 is empty it will run processes in queue #2, and
    - Only if both queues #1 and #2 are empty will it run processes in queue #3.
- Processes are moved between queues as follows:
    - New processes are placed in queue #1. Any process that completes its time slice without causing a page fault is placed at the end of queue #1.
    - Any process that causes a page fault will be moved to the next highest numbered queue (processes in queue #3 remain there).

a. Assuming that the virtual memory system has a fixed allocation of F frames to each processes, how might a programmer take advantage of the above information to obtain favorable scheduling for their program?

A program whose memory references are arranged to fall into F frames will run in queue #1 without interruption. By arranging their program to focus memory references into at most F frames, a programmer can ensure that their program will run uninterrupted.

b. In what way can this scheduling scheme cause some processes to exhibit very poor performance? Explain using a simple example.

A program running without interruption can cause starvation of all other processes. An example would be a program in a tight infinite loop.

3. (20 points) Page replacement algorithms

a. Apply true LRU replacement to the following reference string of page number references, assuming that 3 page frames are allocated to the process. Show which three frames are in memory after each page reference. Show which page each frame is allocated to (if any) after each reference.

Reference string: 3 2 4 5 4 1 2 4 3 4

Pages in memory:

3	3	3	5	5	5	2	2	2	2
	2	2	2	2	1	1	1	3	3
		4	4	4	4	4	4	4	4

b. Apply the Second Chance Clock Algorithm to the same reference string. When a frame is newly allocated, its use bit is initially set to 0. Show which page each frame is allocated to (if any) and the value of each page's use bit after each reference.

The resident pages are the same as shown above for true LRU in this example. The contents of the used bit and the location of the clock hand is as follows:

Reference string: 3 2 4 5 4 1 2 4 3 4

Pages in memory:

0 ←	0	0	0 ←	0 ←	0	0 ←	0 ←	0	0
0	0 ←	0	0	0	0 ←	0	0	0 ←	0 ←
0	0	0 ←	0	1	1	0	1	1	1

4. (20 points) File Systems

a. Explain why the indexed method of disk space allocation in files performs better than the linked method in implementing non-sequential access.

The indexed method concentrates all the links between blocks into a small number of index blocks which can be cached in memory, whereas the linked method embeds the links in the data blocks. For non-sequential access, every access may require that a sequence of links be followed starting at the first block of the file. Thus, the indexed method reduces the number of block accesses required to follow this sequence of links.

b. What is one benefit of the linked method over the indexed method in implementing large files?

The linked method separates the links that define one file from those that define other files. Thus, with the linked method, a very large linked file does not take up space in common link blocks, making them larger than will fit into memory, as can happen with the indexed method.

c. Explain how the Unix inode combines the strengths of both methods in its use of both direct blocks (similar to indexed) and indirect blocks (similar to linked).

The inode is a block which maintains an index of links to the initial data blocks in a file. For larger files, it then stores pointers to indirect blocks which hold links to more of the file's data blocks. Very larger files are implemented using double and triple indirect blocks to store pointers to the data blocks implementing the rest of the file. The initial block of a small file can be cached, providing good non-sequential access. However, if the file is large, then indirect blocks can be brought into memory as needed. However, each file's indirect blocks are cached separately, so the need to read the links implementing a very large file need not affect the performance of access to other, smaller files.

5. (20 points) Security

a. How can a buffer overflow condition be used to hijack a process, and cause it to implement code that is supplied by an attacker?

A buffer overflow condition allows input that is larger than the supplied buffer to write into the data memory adjacent to the buffer, filling it with whatever contents the attacker supplies. If the buffer is in stack memory, then the attacker can write malicious code onto the stack, and overwrite the function return address (stored on the stack) with the location of that code. Upon returning from the function, the program will branch to the malicious code.

b. What is a Trojan Horse attack? Give one example of how such an attack can be made in a Unix/Linux system.

A Trojan Horse attack runs a piece of code that presents the user with a trusted interface (such as a login screen) that legitimately requires privileged information (such as a user password). Upon receiving this information, the attack may perform the expected operation on behalf of the user (in order to avoid detection) but it also communicates the privileged information to back to the attacker.

c. Shared secret cryptography such as DES/AES and public key cryptography have different issues with the secure initial distribution of keys. Explain the problem in each case.

Shared secret cryptography requires both sender and receiver to have the same key, which must be communicated to them. However, without a preexisting secure channel, the key may be intercepted, breaking the security of the scheme.

Public key cryptography allows a user's public key to be distributed freely without the need for secrecy as long as their private key remains a secret. However, it is impossible to know that particular public key belongs to a particular user unless it can be authenticated. One way to authenticate the key is to send it in a preexisting secure channel.

6. (20 points) End-to-End arguments

A secure disk driver encrypts every block before writing it out to the disk, and decrypts it when it is read from the disk, using a key that is compiled into the kernel. A user level secure I/O library encrypts data before before calling the kernel `write()` primitive and decrypts it after calling `read()`, using a key that the user provides each time the program is run.

a. Give an "End-to-End" argument that the user level library is more secure.

The user level library allows the data to remain encrypted until after it has been delivered to the reader's process, avoiding schemes whereby the operating system is attacked and unencrypted data is read from kernel memory. Also, the decryption key is known only to the client, and not to the individual who configures the kernel.

b. Also give an argument in favor of the secure driver without contradicting your argument in part (a) above. The second argument can be on the basis of performance, security, or any other important system property.

An example argument: A driver that performs encryption and decryption in the will decrypt data transferred from disk due to file read-ahead, before it has been explicitly read by the user. Because read-ahead increases resource utilization, it can most efficiently be performed in the kernel, which can schedule resources on behalf of all processes.

7. (20 points) Storage Systems

a. (5 pts) Explain why RAID 0 (striping) increases read performance but decreases fault tolerance.

With RAID 0, a single read must access data on multiple disks. This increases performance because the aggregate bandwidth of these disks is greater than the bandwidth of a single disk. However, if any one disk fails, then no data can be read, so fault tolerance is decreased.

b. (5 pts) Explain how RAID 1 (mirroring) increases both read performance and fault tolerance (surviving any single fault) at the cost of doubling disk usage.

With RAID 1, all data is written to two disks (doubling disk usage), so a single read can access data on both of them, increasing performance as in RAID 0. However, if one disk fails, then the single remaining disk can be used, which increase fault tolerance (although performance suffers after a failure).

c. (10 pts) RAID 4 stripes data blocks across N disks (for some value of N), and for every N data blocks calculates a parity block which it stores on a dedicated parity disk. How is the parity block calculated, and if there is a fault that results in the loss of a data block, how can the data be recovered?

Let the N data blocks be denoted $D_0, D_1, \ldots D_{N-1}$.

The parity block is calculated using a bit-wise XOR operation:

$$P = D_0 \oplus D_1 \oplus \ldots \oplus D_{N-1}$$

If one block (data or parity) is lost, let the remaining N-2 blocks be denoted $B_0, B_1, \ldots B_{N-2}$. *The lost block L can then be recovered by calculating*

$$L = B_0 \oplus B_1 \oplus \ldots \oplus B_{N-2}$$

CS 540 - Operating Systems - Final Exam - Name: _____
Date: Monday, May 12, 2003

Part 1: (80 + 8 (bonus) points - 4 points for each problem)

(C) 1. In an operating system a utility which reads commands from a terminal is called:
(A) Terminal Handler (B) Kernel (C) Shell (D) None of the above

(B) 2. Which scheduler is responsible for controlling the degree of multiprogramming?
(A) Admission scheduler (B) Memory scheduler (C) CPU scheduler (D) None of the above

(A) 3. Which statement about user-level threads and kernel threads is correct?
(A) Both User-level and kernel threads can write into each other's memory space.
(B) Both user-level and kernel threads use OS services via system calls.
(C) Kernel thread scheduling is faster than user-level thread scheduling.
(D) None of the above

(B) 4. Which is not able to solve the race condition?
(A) Test and Set Lock (B) Shared memory (C) Semaphore (D) Monitor

(D) 5. CPU burst distribution is generally characterized as
(A) Constant (B) Linear (C) Polynomial (D) Exponential or hyper-exponential

(A) 6. CPU Scheduling algorithms are used for:
(A) Picking one of the ready processes in main memory to run next
(B) Putting to sleep and waking up processes in an efficient manner
(C) Allocating memory to the processes in a fair and efficient way
(D) None of the above

(D) 7. Which is not a CPU scheduling criterion?
(A) CPU utilization (B) Throughput (C) Waiting time (D) Burst time

(C) 8. Which is a preemptive scheduling?
(A) SJF (B) FCFS (C) RR (D) None of the above

(C) 9. Which is not the necessary condition of a deadlock?
(A) Mutual exclusion (B) Hold and wait (C) Preemption (D) None of the above

(A) 10. Which of the memory allocation schemes are subject to internal fragmentation?
(A) Multiple Contiguous Fixed Partitions (B) Segmentation
(C) Multiple Contiguous Variable Partitions (D) None of the above

(A) 11. A computer provides the user with virtual address space of 2^{24} words. Pages of size 4096 (2^{12}) words If the hexadecimal virtual address is 123456, the page number in hexadecimal would be:
(A) 123 (B) 1234 (C) 456 (D) 3456

(B) 12. If there are 64 frames, and the frame size is 1024 words, the length of physical address is:
(A) 15 bits (B) 16 bits (C) 17 bits (D) None of the above

(C) 13. The modified (dirty) bit is used for the purpose of:
(A) Dynamic allocation of memory used by one process to another
(B) Implementing FIFO page replacement algorithm
(C) To reduce the average time required to service page faults
(D) None of the above

(B) 14. Which page-replacement algorithm suffers from Belady's anomaly?
(A) Least recently used (LRU) (B) Clock (C) Not recently used (NRU) (D) None of the above

(A) 15. Working set model is:
(A) Used for finding the minimum number of frames necessary for a job, so that jobs can run without "thrashing"
(B) Used to find out the average number of frames a job will need in order to run smoothly without causing thrashing
(C) Used to determine whether page replacement is needed
(D) All of the above.

(B) 16. Which file allocation method suffers from disk fragmentation (except for internal fragmentation in the last block)?
(A) Linked list allocation (B) Contiguous allocation (C) I-nodes (D) None of the above

(D) 17. The UNIX system identifies a file as an executable binary file by
(A) File name (B) File extension
(C) File descriptor (D) Magic number in the file header

(B) 18. A table points to routines that handle interrupts is called:
(A) Interrupt handler (B) Interrupt vector (C) Interrupt indicator (D) Interrupt signal

(C) 19. In which of the four I/O software layers is computing the track, sector, and head for a disk read done.
(A) User-level I/O software (B) Device-independent operating system software
(C) Device drivers (D) Interrupt handlers

(D) 20. Which is an example of public-key cryptography?
(A) Caesar cipher (B) Transposition cipher (C) AES (D) PGP

(A) 21. A piece of code which lies dormant until triggered by by some event causing system damage is called:
(A) Logic bomb (B) Trap door (C) Virus (D) Worm

(D) 22. Which security method is not used in JVM?
(A) Sandboxing (B) Code Interpretation (C) Code signing (D) None of the above

Part 2: (120 + 10 (bonus) points)

1. (a) What is a process? Describe the process state.
   (b) What is a thread? Give two benefits of using threads.

   (12 points)

   (a) A process is a running program.
   running: Instructions are being executed.
   ready: The process is waiting to be assigned to a process.
   blocked: The process is waiting for some event to occur.

   (b) A thread is a lightweight process and a basic unit of CPU.
   - Responsiveness: Multiple activities can be done at same time. They can speed up the application.
   - Resource Sharing: Threads share the memory and the resources of the process to which they belong.
   - Economy: They are easy to create and destroy.
   - Utilization of MP (multiprocessor) Architectures: They are useful on multiple CUP systems.

2. How many processes will be created when the following program is executed? Assume that all fork system calls are successful. What will be printed? (Hint: Be careful and draw a picture.) (10 points)

```
main()
{
  int i = 3;
  int ret_val;

  while(i > 0)
  {
    if ((ret_val = fork()) == 0) { /* Child's code */
      printf("In child %d. \n", i);
      exit(0);
    } else {          /* Parent's code */
      printf("In parent %d. \n", i);
      i = i - 1;
    }
  }
}
```

There are 4 processes (1 parent and 3 child processes) created when this program is executed. The following could be printed:

```
In parent 3.
In child 3.
In parent 2.
In child 2.
In parent 1.
In child 1.
```

3. What are four criteria of a good solution to the critical-section problem? (8 points)

   (a) Mutual exclusion is guaranteed.

   (b) Progress is maintained. No process running outside its critical region may block other processes.

   (c) Bounded waiting is assured. No process should have to wait forever to enter its critical region.

   (d) No assumptions are made about the speeds of processes or the number of processors (CPUs).

4. Write a solution to the Producer-Consumer Problem using message passing (UNIX iipes). Recall that producers and consumers have access to a shared buffer that can hold up to N items. Use the notation `write(mbox,msg)` and `read(mbox,msg)`. (12 points)

```
Producer:
---------
  - produce item -
  recv(mbox1, msg);
  send(mbox2, item);

Consumer:
---------
  for (i=0; i<N; i++)
      send(mbox1, NULL_MSG);
  recv(mbox2, item);
  send(mbox1, NULL_MSG);
  - consume item -
```

5. Suppose that the following processes arrive for execution at time 0 in the order A, B, C: (12 points)

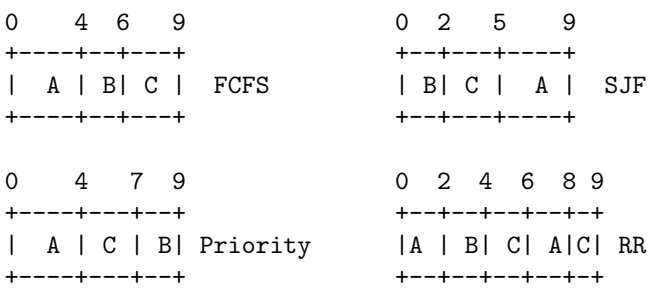
Process	Run Time	Priority
A	4	1=high
B	2	3=low
C	3	2

(a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 2) scheduling.

(b) What is the waiting time of each process for each of the scheduling algorithms?

(c) What is the turnaround time of each process for each of the scheduling algorithms?

(a) The four Gantt charts are



(b) Waiting time:

	FCFS	SJF	Priority	RR
A	0	5	0	4
B	4	0	7	2
C	6	2	4	6

(c) Turnaround time:

	FCFS	SJF	Priority	RR
A	4	9	4	8
B	6	2	9	4
C	9	5	7	9

6. P is a set of processes. R is a set of resources. E is a set of request or assignment edges. The sets P, R, and E are as follows: (10 points)

$P = \{P_1, P_2, P_3\}$ $R = \{R_1, R_2, R_3\}$

$E = \{P_1 \rightarrow R_2, P_2 \rightarrow R_1, P_2 \rightarrow R_2, P_2 \rightarrow R_3, R_1 \rightarrow P_1, R_2 \rightarrow P_3, R_3 \rightarrow P_3\}$

R_1 has one instance. R_2 has two instances. R_3 has one instance.

(a) Draw the resource-allocation graph.

(b) Is there any deadlock in this situation? Briefly Explain.

(a) See the graph.

(b) Consider the resource-allocation graph. There are no cycle in the system.
P1, P2, and P3 are not deadlocked.

7. Given memory partitions of 100 KB, 400 KB, 200 KB, and 500 KB (in order). How would each of the first-fit, next-fit, best-fit, and worst-fit algorithms place processes of 200 KB, 396 KB, 100 KB, and 290 KB (in order)? (12 points)

   First fit : (a) 400 KB (b) 500 KB (c) 100 KB (d) No fit
   Next fit : (a) 400 KB (b) 500 KB (c) remainder of 500 KB (d) No fit
   Best fit : (a) 200 KB (b) 400 KB (c) 100 KB (d) 500 KB
   Worst fit : (a) 500 KB (b) 400 KB (c) remainder of 500 KB (d) No fit

8. A small computer has five page frames. At the first clock tick, the R bits are 10011 (page 1 and 2 are 0). A subsequent clock ticks, the values are 10100, 10101, 00101, 01100, 01011, and 10101. If the aging algorithm is used, with a 4-bit counter, give the values of the counters after the last tick. Which page would be selected to be removed from memory? (12 points)

R-bits	10011	10100	10101	00101	01100	01011	10101
Page							
0	1000	1100	1110	0111	0011	0001	1000
1	0000	0000	0000	0000	1000	1100	0110
2	0000	1000	1100	1110	1111	0111	1011
3	1000	0100	0010	0001	0000	1000	0100
4	1000	0100	1010	1101	0110	1011	1101

   The page 3 will be evicited because it has the smallest couter value.

9. Consider a computer is equipped with associative memory that can hold 16 entries of the page table and can be accessed in 10 nanoseconds. The hit ratio is the percentage of the page table entry can be found in the associative memory. The CPU takes total 130 nanoseconds to search the page entry and access a data item when the page entry is not in the associative memory. (10 points)

   (a) Find a formula that expresses the effective access time as a function of the hit ratio (h).
   (b) What hit ratio is needed to achieve the effective access time to 82 nanoseconds?

   (a) Let E = associative memory lookup time, T = memory cycle time, h = hit ratio.
   $E + 2 \times T = 130$, $E = 10 \Rightarrow T = 60$. $E + T = 70$
   Effective Access Time (EAT) = $70 \times h + (1 - h) \times 130 = 70 \times h + 130 - 130 \times h = 130 - 60 \times h$
   (b) $130 - 60 \times h \leq 82$, $60 \times h \geq 130 - 82$, $60 \times h \geq 48$, $h \geq 80\%$

10. Suppose there are 16 virtual pages and 4 page frames. Determine the number of page faults that will occur with the reference string 0 1 2 3 2 4 1 3 5 6 1 3 2 7 4 8, if the page frames are initially empty, using each of the following page replacement algorithms: (a) FIFO (b) LRU (c) Optimal. (12 points)

    (a) FIFO: 13 (b) LRU: 11 (c) Optimal: 10

11. Encrypt the following plaintext using a transposition cipher based on the key `LEOPARD`. (10 points)
    plaintext = H A V E A G R E A T S U M M E R B R E A K

L	E	O	P	A	R	D
4	3	5	6	1	7	2
H	A	V	E	A	G	R
E	A	T	S	U	M	M
E	R	B	R	E	A	K

    ciphertext = A U E R M K A A R H E E V T B E S R G M A

12. What is a virus? What is a polymorphic virus? Give two examples of anti-virus products. (10 points)

- Virus is a program can reproduce itself by attaching its code to another program and additionally do harm.
- A virus that mutates on each copy is called a polymorphic virus.
- Norton Anti-virus, McAfee VirusScan

FINAL EXAM with SOLUTION
Dec 17, 2007

Operating systems, V22.0202, Yap, Fall'07

1. PLEASE READ INSTRUCTIONS CAREFULLY.
2. This is a closed book exam, but you may refer to two $8'' \times 11''$ sheets of prepared notes.
3. Please write ONLY on the right-hand side of a double page. USE THE left-hand side of the double page for scrap.
4. Attempt all questions.

PART 1. VERY SHORT QUESTIONS. (5 Points each)
One or two-sentence answers only. For True/False questions, it is essential to provide the justifications asked for.

---

1. Give three reasons why a CPU is sometimes interrupted. One sentence per reason.

   SOLUTION *(1) There is an error in execution.*

   (2) An I/O device has completed an operation.

   (3) A user needs to access a protected (system) function. ♠

2. TRUE or FALSE: Our Toy HardwarE (THE machine) has relocatable addressing.

   SOLUTION *TRUE. It uses a base register to relocate logical addresses.* ♠

3. TRUE or FALSE: Paging causes external fragmentation. If true, justify; if false, modify into a correct statement by changing one word.

   SOLUTION *FALSE. The correct formulation can be one of the following: "Paging causes internal fragmentation" or "Paging solves external fragmentation".* ♠

4. Describe Belady's anomaly.

   SOLUTION *This is the phenomenon in page-replacement algorithms in which an increase in the number of page frames can cause the number of page faults to increase.*

   NOTE: FIFO algorithm exhibits Belady's anomaly. ♠

5. What is the advantage of using page sizes that is a power of two?

   SOLUTION *This allows you to take a logical address (in binary) and take its lower order bits as the page offset, and the higher order bits as the page number.* ♠

6. Suppose our paging system uses a Translation Lookaside Buffer (TLB). Each memory reference takes 300 ns, and each look up of the TLB takes 20 ns. What is the effective memory reference time if 80 percent of page-table references are found in the TLB?

   SOLUTION *Effective time in nanoseconds is* $(0.8 \times 320) + (0.2 \times 620) = 380$. ♠

7. An I-Node does NOT contain the name of the file it represents. Explain why this ought to be so.

   SOLUTION *An I-Node represents a physical file, but there might be several logical files with different names that point (links) to the same physical file.*

   REMARK: the file name is stored in the directory containing the file. ♠

PART 2. SHORT QUESTIONS. (10 points each)
Answers must be at most 4 sentences. So please sketch a rough solution first!

1. The code for wait() and signal() on a semaphore S is as follows.

$$wait(S) \equiv \{\textsf{while } S \leq 0; S\textsf{--}; \}$$

$$signal(S) \equiv \{S\textsf{++}; \}$$

Assume S is initially 1 and two processes P and Q protects a critical region with the semaphore S. *Show that if one of these operations is not executed atomically, we may violate mutual exclusion.*

SOLUTION *Suppose $wait(S)$ is not atomic. Initially P reads S, finds the value positive, and proceeds to decrement S. But before it writes the value 0 to S, P is interrupted and Q begins to to execute $Wait(S)$. Q also reads $S = 1$, decrement the value to 0 and write out $S = 0$, and enters the critical region. Then Q is interrupted and P begins to run. P resumes by setting $S = 0$ (it is already set to 0 by Q) enters the critical region – violating mutual exclusion!!*

Note: $signal(S)$ does not have to be atomic. ♠

2. When we use paging, each memory reference can turn into two *or more* memory references. Explain this remark.

SOLUTION *Each memory reference must first involve looking up the page table to find the frame containing the page containing our logical address. Looking up the page table is one memory reference. The second memory reference is to read the particular offset in a frame, corresponding to the logical address.*

Now, if the page table is large, we need to organize this as a hierarchical table. If the depth of this hierarchy is more $h + 1$, then we will need $h + 1$ memory references. ♠

3. In virtual memory (demand paging), we associate an r-bit and m-bit with each frame. (a) How are these bits updated? (b) How are these bits used in implementing the NRU (Not Recently Used) policy for page replacement?

SOLUTION *(a) The m-bit is set each time we write into the frame while the r-bit is set each time we access the frame. However, the r-bit is periodically reset, depending on the system clock.*

(b) each page belongs to a category 0,1,2 or 3, where the category of a page is given by the pair (r-bit,m-bit), viewed (r-bit,m-bit) as a binary number. We choose a frame with lowest category for eviction. ♠

4. Segmentation serves a different purpose than paging. Describe three uses of segmentation of processes.

SOLUTION *(1) Sharing of code segments across processes. (2) Refined protection control for data and code. (3) Management of multiple address spaces for data structures that can grow and shrink during run-time. (4) Paging performance can be improved also.* ♠

5. Consider the organization of memory for processes, and the organization of disk for files. Describe one similarity, and one difference.

SOLUTION *(1) Similarity in discrete space allocation: We divide memory into fixed-size units called frames, and divide disk into fixed-size units called blocks. (Both are methods of reducing fragmentation.)*

(2) Difference in Caching: To speed up memory access in the presence of paging, we use a caching mechanism based on the Translation Lookaside Buffer (TLB). There is no corresponding hardware support for caching between disk and memory. The closest to this idea is the use of swap space on the disk, allowing raw disk blocks to transferred somewhat faster. ♠

6. Compute the inverse of 47 modulo 60. Explain your workings for full credit.

   SOLUTION *We use the extended Euclidean algorithm. Here are the details of the computation*

i	m_i	q_i	s_i	t_i
0	60		1	0
1	43		0	1
2	17	1	1	−1
3	9	2	−2	3
4	8	1	3	−4
5	1	1	−5	7

   Check: $1 = 60 \times (-5) + 43 \times 7$. ♠

PART 3. LONG QUESTIONS. (20 Points each)
You may use more than 3 sentences to explain, and ought to consider the issues in greater depth.

---

1. (Scheduling)

   Joe Smart says "If you want to schedule processes to minimize the TOTAL wait time of all processes, it is quite easy. Just use the Shortest Jobs First rule." *First explain what Joe means. Then prove or argue why Joe is right. Finally, discuss the issues raised by this Smart idea and how we can resolve them.*

   SOLUTION *Smart is right: here is the proof. Suppose P takes time T and P' takes time T' where $T > T'$. Let us compare the total wait time S_1 when P is scheduled ahead of P', with the total wait time S_2 when P' is scheduled ahead of P. We see that $S_1 - S_2 = (2T + T') - (2T' + T) = T - T' > 0$. Thus S_1 is more than S_2.*

   There are two issues.

   (1) This solution assumes that WE KNOW THE RUNNING TIME OF EACH JOB. But we don't usually. To get this idea to work, we must have some ways to estimate the "expected" run time of each job. One possible solution is to use past executions of each program (perhaps combined with "aging" – see midterm) to estimate the expected run time.

   (2) If the jobs arrive at different times, we need to use preemption to minimize the total wait time. I.e., we now look at the remaining time to completion of job, and schedule the one with the least remaining time.

   This raises an issue of fairness: what if a long job is REPEATELY preempted? Say a 2-unit job is repeated preempted by the arrivals of many 1-unit jobs. Now it is not so clear that total wait time is the best policy. Of course, policy issue is not easily decided. ♠

2. (Paging) Consider a paging system with a 32-bit logical address space. Each address refers to a byte in memory. Let the page size be 16 KB, and main memory size be 256 MB. What is the minimal size (in bytes) of the page table? Please show working.

   SOLUTION *ANSWER: 0.5MB.*

   Each page has 2^{14} Bytes, and main memory has 2^{28} Bytes or $2^{28}/2^{14} = 2^{14}$ frames. Hence the frames have 14-bit addresses. Thus, each entry of the page table need at least 14 bits,

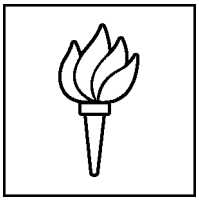
or at least 2 bytes. Since logical address space has $2^{32}/2^{14} = 2^{18}$ *pages, the page table has* $2^{18} \times 2 = 2^{19}$ *bytes, or* $0.5MB$.

REMARK: In practice, we probably want more than 2 bytes per entry: we may need space for the m-bit, r-bit, protection bits, present/absent bit, etc. ♠

3. (RSA Cryptosystem) Alice sent a secret message M to Bob using protocol A (which authenticates the receiver). Bob's public key is the $(e, n) = (43, 77)$. You are Jim Bond, the master spy in Her Majesty's secret service. You intercepted Alice's encrypted message $\widetilde{M}$. Normally, this code is unbreakable, but as Bond, you have access to unlimited computing power and can even factor numbers (gasp!). So you proceeded to break Alice's secret.

   What is the message M *given that* $\widetilde{M} = 2$*? Describe your method and show your computations.* You can get partial credits even if you could not compute M.

   SOLUTION $M = 51$. *Using the super-computer in Her Majesty's service, you managed to factor* $n = 77$ *after a long computation. You found that* $n = pq = 7 \times 11$. *So you compute* $\phi(n) = 6 \times 10 = 60$. *You know that* $e = 43$. *Using the Extended Euclidean algorithm, you compute* $d = e^{-1} \bmod 60 = 7$ *(see a problem in the Short Questions Part). Now the original message must be* $(\widetilde{M})^7 \equiv 2^7 \equiv 128 \equiv 51(\mod 77)$. ♠



Computer Science Department

New York University

G22.2250-001 Operating Systems: Spring 2009

Final Exam

Many problems are open-ended questions. In order to receive credit you must answer the question *as precisely as possible*. You have 90 minutes to answer this quiz.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/16)	II (xx/15)	III (xx/20)	IV (xx/10)	V (xx/14)	VI (xx/5)	Total (xx/80)

Final statistics:

Score range	Num of students
$[60, 80]$	2
$[55, 60)$	3
$[50, 55)$	4
$[40, 50)$	6
$[30, 40)$	6
$[0, 30)$	4

I Basic OS knowledge

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 4 points. Each missing or wrong answer costs -2 point.

A. In a typical x86-based OS, which of the following statements are true about virtual memory?

1. The kernel is mapped into each user-level application process' address space in order for applications to access the kernel's data conveniently.
2. A user-level application cannot modify its page table entries because the `PTE_U` flag of its page table entries (PTEs) is cleared by the kernel.
3. In 32-bit x86 with paging turned on, a machine instruction such as "mov 0x08000000, %eax" could potentially involve 3 accesses to main memory.
4. On x86, upon a TLB miss, the kernel traverses the 2-level page table to fill in the virtual to physical address mapping in the TLB.

Ans: 3 or 2,3.
1 is incorrect. Kernel is mapped into user-level process' address space for kernel to access applications' data conveniently (and to avoid TLB flushes when transitioning from user-level to kernel level execution.)
4 is incorrect. x86's TLB is filled by the hardware instead of the kernel (software).
2 is ambigious and hence we give credits for both. A user-level application cannot modify its page table entries bacause its page table resides in kernel's portion of the address space. And the kerne's portion of the address space is not accessible by user-level processes because its corresponding page table entries have `PTE_U` *flag cleared.*

B. Which of the following statements are true about file systems?

1. Reading a random location in big files is usually faster in inode-based file systems than the FAT file systems.
2. For better sequential read/write performance, the file system should try to allocate consecutive blocks to a file.
3. If the journaling file system does not immediately flush the committed transaction record of each completed FS operation to disk, the FS will become inconsistent upon crash and recovery.
4. If the journaling file system does not immediately flush the committed transaction record of each completed FS operation to disk, the FS might lose the last few FS operations but will remain consistent upon crash and recovery.

Ans: 1, 2 and 4
2 is correct. When performing sequential read/writes, the achievable throughput is limited by disk throughput, e.g. 40MB/s. This is much better than performing random read/writes whose achievable throughput is limited by disk seek time. E.g. if each read is 100 bytes, then random read throughput is $\frac{100}{10*10^{-3}} = 10KB/sec$ *for a disk with 10ms seek time.*

C. Which of the following statements are true about virtual machines?

1. A VMM can virtualize the x86 interface on top of any hardware architecture (e.g. the MIPS or ARM architecture)
2. Using dynamic binary translations to implement VMM slows down kernel-level function calls (`call`) and function returns (`ret`).

3. Using dynamic binary translations to implement VMM slows down user-level function calls (`call`) and function returns (`ret`).
4. The guest OS kernels cannot access the code and data of VMM (e.g. the shadow page table).

Ans: 2,4.
1 is incorrect because virtualization requires the VMM to be able to execute most of the instructions identically. If one is to simulate a x86 interface on top of MIPS, then every x86 instruction needs to be simulated using many MIPS instructions. This is called "simulation", not "virtualization".

2 is correct because dynamic binary translation cannot translate the `call` *and* `ret` *instructions identically. For direct* `call`*s, it's possible for the binary translator (BT) to directly substitute the call target's address with its translated code fragment's address at translation time, thus avoiding the cost of hash table lookups at execution time. For* `ret`*, no such optimization is possible and the translated code must call into BT to perform a hash table lookup to find the translated target's address, resulting in slow down at execution time.*

4 is incorrect because user-level processes are not subject to binary translation.

D. Which of the following statements are true?

1. A non-root user in UNIX can never launch programs that execute with root privileges.
2. Only the owner of a file can set the `setuid` bit on that file.
3. TOCTTOU bugs can only happen to `setuid` programs.
4. We can solve UNIX' security problems by having each user encrypt all his files.

Ans: 2.
In most UNIX systems such as Linux, the root cannot directly set the `setuid` *bit of programs not owned by root. However, the root can indirectly set the* `setuid` *bit by first changing its identity to be the owner of those files and then set the bit. Therefore, we give full credits to those who did not mark 2 as correct.*

3 is incorrect because TOCTTOU is a generic type of bug that can happen to any privileged programs, i.e. a normal root process can be vulnerable to TOCTTOU bugs.

II Synchronization

We often build higher-level synchronization primitives on top of lower-level ones. For example, we have seen how one can build mutexes using spin-locks during Lecture. Another useful high-level primitive is the reader/writer lock. A reader/writer lock allows multiple threads to acquire read access to shared data simultaneously, on the other hand, a thread modifying the shared data can only proceed when no other thread is accessing the data. A reader/writer lock can be implemented on top of mutexes and conditional variables.

Recall that mutex and conditional variable export the following interfaces:

- Mutex has type `mutex_t` and it implements `lock(mutex_t *m)` and `unlock(mutex_t *m)`.
- Conditional variable has type `condvar_t`. The function `cond_wait(condvar_t *cv, mutex_t *m)` releases the mutex pointed to by `m` and blocks the invoking thread until `cond_broadcast` or `cond_notify` is called. The function `cond_broadcast(condvar_t *cv)` wakes up all threads blocked in `cond_wait`. The function `cond_notify(condvar_t *cv)` wakes up one of the threads blocked in `cond_wait`.

The reader/writer lock has type `rwlock_t` and it needs to implement the following interfaces:
`read_lock(rwlock_t *rw)` //allow any number of readers to proceed if no writer has grabbed lock
`read_unlock(rwlock_t *rw)`
`write_lock(rwlock_t *rw)` //allow a writer to proceed only if no reader nor writer has grabbed lock
`write_unlock(rwlock_t *rw)`

1. [10 points]: Write C code to implement the 4 function interfaces of reader/writer lock using the mutex and condition variable. Hint: you might want to declare `rwlock_t` as follows (or you are welcome to use a different data structure for `rwlock_t`):

```
typedef struct rwlock {
  int nreaders; //number of readers who have grabbed the lock, initialized to 0
  int nwriters; //number of writers who have grabbed the lock, initialized to 0
  mutex_t m;
  condvar_t cv;
} rwlock_t;
```

```
void
read_lock(rwlock_t *rw) {
  lock(&rw->m);
  while (rw->nwriters > 0)
    cond_wait(&rw->cv,&rw->m);

  rw->nreaders++;
  unlock(&rw->m);
}

void
read_unlock(rwlock_t *rw) {
  lock(&rw->m);
  rw->nreaders--;
  cond_broadcast(&rw->cv);
  unlock(&rw->m);
}

void
write_lock(rwlock_t *rw) {
  lock(&rw->m);
  while (rw->nreaders > 0 && rw->nwriters > 0)
    cond_wait(&rw->cv, &rw->m);

  rw->nwriters++;
  unlock(&rw->m);
}

void
write_unlock(rwlock_t *rw) {
  lock(&rw->m);
  rw->nwriters--;
  cond_broadcast(&rw->cv);
  unlock(&rw->m);
}
```

The above code correctly implements the reader/writer lock. However, it is not fair in the sense that readers might starve a waiting writer: there could be arbitarily many readers coming and holding the lock while a writer is waiting for the lock. There are many ways to fix the writer starvation problem. For example, one could make new readers wait when writers have been waiting for a long time.

2. [5 points]: Why does the `cond_wait(condvar_t *cv, mutex_t *m)` function require a second argument pointing to a mutex? In other words, if you replace every line `cond_wait(cv, m);` in your reader/writer lock implementation with two lines `unlock(m); cond_wait(cv); lock(m)` is your code still correct? Explain.

Ans: `cond_wait` *function must perform the two steps (1. release mutex 2. block until cond_notify) atomically. If not, after the mutex is unlocked (*`unlock(m)`*) but before wait is called (*`cond_wait(cv)`*), the producer thread (i.e. threads doing* `read_unlock` *or* `write_unlock` *functions) might come in between and generate a signal that nobody is waiting for, resulting in the signal becoming lost. Please argue for yourself why performing unlock and wait atomically in* `cond_wait` *avoids such sleep-wakeup race.*

III File system Layout

3. [5 points]: In an i-node based file system implementation, the i-node typically stores 12 direct block pointers, one 1-indirect block pointer, one 2-indirect block pointer, and one 3-indirect block pointer. Recall that an indirect block is a disk block storing an array of disk block addresses (i.e. pointers). The pointers in a 1-indirect block point to disk blocks that store file data. The pointers in a 2-indirect (or 3-indirect) block point to other 1-indirect (or 2-indirect) blocks. Suppose the file system is configured to use a block size of 2^{10} bytes and each pointer takes up 4-byte. What is the maximum file size that can be supported in the file system? Explain your calculation.

Ans: Each 1-indirect block can address $2^{10}/4 = 2^8$ *data blocks. Each 2-indirect block can address* $2^8 * 2^8 = 2^{16}$ *data blocks. Each 3-indirect block can address* $2^8 * 2^8 * 2^8 = 2^{24}$ *data blocks. In total, the biggest file can contain at most* $12 + 2^8 + 2^{16} + 2^{24} \approx 2^{24}$ *data blocks (i.e.* $2^{24} * 2^{10} = 4GB$*).*

4. [5 points]: Ben Bitdiddle runs a program that reads 100-byte data chunks in random locations of a file. What's the maximum number of random reads can Ben's program hope to achieve in a second? (Explain. Write down your assumptions about hardware if there's any.)

Ans: The latency of reading a random location on disk is dominated by seek+rotation delay. Assume a typical seek time of 10ms, Ben can hope to perform at most 100 random reads/sec.

A number of students calculated random read throughput based on disk throughput, which is incorrect. One achieves disk throughput (e.g. 40MB/sec) only for sequential reads/writes.

5. [5 points]: Ben notices that his program gets close to the best possible random read throughput when running on small files. However, when reading from large files ($>$ 1GB), the actual random read throughput is much lower. Why? Explain with concrete performance numbers.

Ans: To read a random location in a big file ($>$ 1GB), the file system must read 3 indirect blocks in addition to the data block. For a random file data block, its indirect blocks also reside on random (or non-sequential) locations on disk. Thus, one could suffer 4 seeks for each random read in a big file, resulting in 25 reads/sec.

6. [5 points]: Please describe an alternative scheme to replace the indirect-block based mapping scheme. Your scheme should improve the performance of random reads in large files.

Ans: To improve random reads in large files, we need to be able to address all the data blocks of a big file with fewer bytes. Since the file system already tries to allocate consecutive blocks to a file (so sequential file accesses result in sequential disk reads/writes, maximizing throughput.), we can have the file system record the start and end block address of each consecutive run of blocks. For example, if a 1GB file consists of 10 consecutive regions of blocks, we only need 20 numbers to address all of 1 million ($2^{32}/2^{10}$) data blocks. The upcoming Linux ext4 file system will have this feature.

Some of you also propose to make the block size larger. This solution will increase internal fragmentation for small files. Since a large fraction of files are small ($< 4KB$), the increased amount of internal fragmentation is substantial.

IV File system crash recovery

The Linux journaling file system writes the content of all modified disk blocks to the log. Ben Bitdiddle finds such logging wasteful since copying the content of modified disk blocks to the log doubles the amount of disk writes for each logged file system operation.

Ben decides to implement a more efficient journaling file system. In particular, he decides to only record an operation's name and parameter in the log file instead of recording the content of all modified blocks. For example, for an operation that creates file "/d/f", the file system appends the transaction record [create "/d/f"] to the log. Ben's file system ensures that the corresponding transaction record is written to the log before the modified disk blocks are flushed to disk. Upon crash and recovery, Ben's file system re-executes the logged file system operations and truncates the log.

7. [10 points]: Ben's new logging mechanism is certainly more efficient since each transaction record is much smaller than that with Linux's logging. Is his design also correct? i.e. can it recover a file system correctly from crashes? Explain your reasoning and give concrete examples.

Ben's design is not correct. Consider the following example. The unlink("/D/f") operation involves modifying 5 blocks:

1. decrement the nlink field of "/D/f"'s i-node
2. modify direcotory D's dir block to remove a's entry
3. modify D's i-node to update mtime, length
4. modify i-node bitmap if a's i-node is now free
5. modify block bitmap to indicate a's data blocks are now free

The FS logs the operation [unlink "/D/f"] and crashes upon performing step 1,2 and before steps 3,4,5. Upon recovery, the FS attempts to re-execute the operation unlink("/D/f"). However, this operation fails upon noticing that there's no file "f" in the directory "D". Hence, the i-node freemap and block freemap become inconsistent (i.e. freed i-node and disk blocks are "lost").

Two students gave the following excellent example. Suppose two files ("/D/f1" and "/D/f2") are hard-links of each other, i.e. they have the same i-node whose nlink field is 2. Suppose the operation unlink("/D/f1") crashes after flushing modified data belonging to step 1 to disk, so the corresponding i-node's nlink field is decremented to 1. Upon recovery, the logged operation unlink("/D/f1") is re-executed, causing the nlink field to be decremented again to be zero and the i-node to be freed. This is incorrect because "/D/f2" still uses that i-node!

In summary, simply logging the operation's name is not correct because the high-level operations like create/link/unlink are not idempotent. Non-idempotent means applying an operation multiple times leads to different results than applying it once (because the state has changed). When log operations are not idempotent, the recovery process must apply them based on the same state as that before the crash in order to achieve identical results. Unfortunately, when crash happens, the disk might have already written some subset of modified data blocks. As a result, applying a non-idempotent operation on these modified data blocks yields different (and inconsistent) results.

V A Different VM interface

Ben Bitdiddle finds the VM interface provided by MemOS to applications limiting. In MemOS, applications use the `sys_page_alloc(va)` syscall to request for more memory. To handle this syscall, the MemOS kernel simply allocates a new physical page and maps it to the requested virtual address `va`.

Ben decides to adopt a different interface that gives applications greater control over how it manages its memory and address space. Ben's interface contains two system calls, `sys_phypage_alloc()` and `sys_map_va2pa(va, pa, perm)`.

The `sys_phypage_alloc()` syscall requests a physical page from the kernel and it returns a physical address to the application. The `sys_map_va2pa(va, pa, perm)` syscall asks the kernel to add a mapping from the virtual address `va` to the physical address `pa` with permission bits `perm`. It's easy to see that applications can use these two system calls in combination to achieve the equivalent effect of the original system call `sys_page_alloc`.

Ben has implemented his new interface based on Lab 3. Here's a snippet of Ben's system call implementation.

```
...
void
interrupt(register_t *reg) {
  current->p_registers = *reg;
  reg = &current->p_registers;

  switch (reg->reg_intno) {
    case INT_SYS_PHYPAGE_ALLOC:
      //page_alloc_free allocates an idle physical page
      //and records the ownership of the page in the pageinfo array
      reg->reg_eax = page_alloc_free(PO_PROC+current->p_pid);
      run(current);
    case INT_SYS_MAP_VA2PA:
      //%eax,%ebx,%ecx contains the 3 syscall parameters va, pa, perm respectively
      //pgdir_set(pgdir, va, pa|perm) modifies page table pgdir to add mapping
      //from va to page table entry pa|perm
      pgdir_set(current->p_pgdir, reg->reg_eax, reg->reg_ebx | reg->reg_ecx);
      reg->reg_eax = 0; //syscall returns 0 to indicate success
      run(current);
    case ... :
      ...
  }

}
...
```

8. [10 points]: Ben tells Alyssa Hacker about his new interface. Upon examining Ben's implementation, Alyssa declares that Ben's implementation is not secure, i.e. it does not provide proper isolation against malicious or buggy applications. Is Alyssa right? Explain and provide a plausible fix. You can describe your fix in plain English instead of C code.

Ans: Ben's implementation is not correct. A malicious user-level program can read/write arbitary memory belonging to the kernel or other processes by calling `sys_map_va2pa` *with arbitary physical addresses.*

The fix: when handling `INT_SYS_MAP_VA2PA`*, the kernel must check that* `pa` *argument is owned by the requesting process by examining the* `pageinfo` *array.*

9. [4 points]: Alyssa Hacker is very enthusiastic about Ben's new interface. By exposing physical addresses to applications, Alyssa argues, this interface could enable application-specific memory management. For example, database applications typically perform application-level caching. It's desirable to let the database application decide what physical page to swap out to disk when the system is under memory pressure (instead of having the kernel swap out pages without an application's knowledge using a fixed LRU policy). Augment Ben's interface to handle physical page de-allocation and swapping in a way that enables such application-level memory management. Describe how an example application such as the database application can use Ben's interface to better manage its memory.

Ans: When under memory pressure, the kernel could make an upcall to applications to request for the release of some memory pages. The application could pick a physical page to swap out. If the chosen page is dirty, the application writes it to disk first. Then, the application invokes a syscall like `page_dealloc(pa)` *to release the chosen page and unmap the appropriate page table entries.*

The advantage of this setup is that applications, who have the most information, can have full control of the page eviction policy. If you are interested in learning more about this approach, you can look up this research paper for more information: `http://www.news.cs.nyu.edu/~jinyang/sp09/readings/engler95exokernel.pdf`

VI G22.2250-001

10. [5 points]: We would like to hear your opinions about the class. Please tell us what you like the most/least about this class. Any suggestions for improving the class are welcome.

End of Final

CS4411 Intro. to Operating Systems Exam 2 Solutions Fall 2010

1. **Recycled Problem(s):**

   (a) **[6 points]** Define the meaning of a *race condition*? Answer the question first and use an execution sequence to illustrate your answer. **You will receive no credit if only an example is provided without an elaboration.**

   Answer: A *race condition* is a situation in which *more than one* processes or threads access a shared resource *concurrently*, and the result depends on the order of execution.

   The following is a simple counter updating example discussed in class. The value of `count` may be 9, 10 or 11, depending on the order of execution of the **machine instructions** of `count++` and `count--`.

```
int         count = 10;

Thread_1(...)              Thread_2(...)
{                          {
   // do something             // do something
   count++;                    count--;
}                          }
```

   The following execution sequence shows a race condition. Two threads run concurrently (condition 1). Both threads access the shared variable `count` at the same time (condition 2). Finally, the computation result depends on the order of execution of the `SAVE` instructions (condition 3). The execution sequence below shows the result being 9; however, switching the two `SAVE` instructions yields 11. Since all conditions are met, we have a race condition.

`Thread_1`	`Thread_2`	*Comment*
do somthing	do somthing	`count = 10` initially
`LOAD count`		`Thread_1` executes `count++`
`ADD #1`		
	`LOAD count`	`Thread_2` executes `count--`
	`SUB #1`	
`SAVE count`		`count` is 11 in memory
	`SAVE count`	Now, `count` is 9 in memory

   Stating that "`count++` followed by `count--`" or "`count--` followed by `count++`" would produce different results and hence a race condition is incorrect, because the threads do not access the shared variable `count` at the same time (*i.e.*, condition 2).

   See **p. 193** of our text and class notes. ∎

2. **Synchronization**

   (a) **[10 points]** The semaphore methods `Wait()` and `Signal()` must be atomic to ensure a correct implementation of mutual exclusion. Use an execution sequence to show that if `Wait()` is not atomic then mutual exclusion cannot be maintained. **You must use an execution sequence to present your answer as we did in class. Otherwise, you risk low or zero point.**

Answer: If `Wait()` is not atomic, its execution may be switched in the middle. If this happens, mutual exclusion will not be maintained. The following is a possible execution sequence, where `Count = 1` is the counter variable of the involved semaphore.

Process A	*Process B*	`Count`	*Comment*
		1	Initial value
`LOAD Count`		1	A executes `Count--` of `Wait()`
`SUB #1`		1	
	`LOAD Count`	1	B executes `Count--` of `Wait()`
	`SUB #1`	1	
	`SAVE Count`	0	B finishes `Count--`
`SAVE Count`		0	A finishes `Count--`
`if (Count < 0)`		0	It is false for A
	`if (Count < 0)`	0	It is false for B
Both A and B enter the critical section			

Note that this question asks you to demonstrate a violation of mutual exclusion. Consequently, you receive low grade if your demonstration is not a violation of mutual exclusion.

This problem was assigned as an exercise in class. ■

(b) **[8 points]** Enumerate and elaborate all major differences between a semaphore wait/signal and a condition variable wait/signal. Vague answers and/or inaccurate or missing elaboration receive **no** credit.

Answer: The following table has the details:

Semaphores	**Condition Variables**
Can be used anywhere, but not in a monitor	Can only be used in monitors
`wait()` does not always block its caller	`wait()` **always** blocks its caller
`signal()` increases the semaphore counter and may release a process	`signal()` either releases a process, or the signal is lost as if it never occurs
If `signal()` releases a process, the caller and the released **both** continue	If `signal()` releases a process, either the caller or the released continues, but **not both**

This is part of the monitors slides discussed in class. ■

3. **Process Scheduling**

(a) **[8 points]** What are *preemptive* and *non-preemptive* scheduling policies? Elaborate your answer.

Answer: With the *non-preemptive* scheduling policy, scheduling only occurs when a process enters the wait state or terminates. With the *preemptive* scheduling policy, scheduling also occurs when a process switches from running to ready due to an interrupt, and from waiting to ready (*i.e.*, I/O completion).

See **pp. 153–154** of our text. ∎

(b) **[8 points]** What is *priority inversion*? How could it happen? How can it be overcome? **Note that there are <u>three</u> questions here.**

<u>Answer</u>: If a high-priority process needs to access a protected resource that is currently being held by a low-priority process, the high-priority process is blocked by the low-priority process until the low-priority process completes its access. Thus, a low priority process actually has a "higher" priority because it forces higher priority processes to wait! This is *priority inversion*.

To overcome this priority inversion problem, processes that are accessing the resource that the high-priority process needs inherit the high priority until they are done with the resource. As a result, this will speed up the low-priority processes. When the low priority processes finish, their priority reverts to its original value. This is the *priority-inheritance protocol*.

See **p. 238** of our text. ∎

(c) **[20 points]** Five processes A, B, C, D and E arrived in this order at the same time with the following CPU burst and priority values. A smaller value means a higher priority.

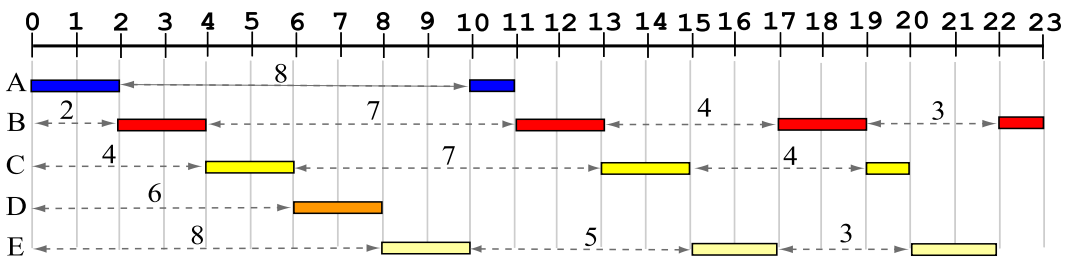
	CPU Burst	*Priority*
A	3	3
B	7	5
C	5	1
D	2	4
E	6	2

Fill the entries of the following table with waiting time and average waiting time for each indicated scheduling policy and each process. Ignore context switching overhead.

Scheduling Policy	*Waiting Time*					*Average Waiting Time*
	A	B	C	D	E	
First-Come-First-Served						
Non-Preemptive Shortest-Job First						
Priority						
Round-Robin (time quantum=2)						

<u>Answer</u>:

Scheduling Policy	*Waiting Time*					*Average Waiting Time*
	A	B	C	D	E	
First-Come-First-Served	0	3	10	15	17	45/5 = 9
Non-Preemptive Shortest-Job First	2	16	5	0	10	33/5=6.6
Priority	11	16	0	14	5	46/5=9.2
Round-Robin (time quantum=2)	8	16	15	6	16	61/5=12.2



The above diagram shows the execution pattern of the round-robin algorithm with time quantum 2, where dashed arrows indicate waiting periods.

See class notes for the details. ∎

4. **Problem Solving:**

(a) **[20 points]** Design a class `Barrier` in C++, a constructor, and method `Barrier_wait()` that fulfill the following specification:

- The constructor `Barrier(int n)` takes a positive integer argument `n`, and initializes a private `int` variable in class `Barrier` to have the value of `n`.
- Method `Barrier_wait(void)` takes no argument. A thread that calls `Barrier_wait()` blocks if the number of threads being blocked is less than `n`-1, where `n` is the initialization value and will not change in the execution of the program. Then, the `n`-*th* calling thread releases all `n`-1 blocked threads and all `n` threads continue. Note that the system has more than `n` threads. Suppose `n` is initialized to 3. The first two threads that call `Barrier_wait()` block. When the third thread calls `Barrier_wait()`, the two blocked threads are released, and all three threads continue. Note that your solution **cannot** assume `n` to be 3. Otherwise, you will receive **zero** point.

Use semaphores only to implement class `Barrier` *and method* `Barrier_wait()`. Otherwise, you will receive **zero** point. You may use type `Sem` for semaphore declaration and initialization (*e.g.*, "`Sem S = 0;`"), `Wait(S)` on a semaphore `S`, and `Signal(S)` to signal semaphore `S`.

You should explain why your implementation is correct in some details. A vague discussion or no discussion receives no credit.

Answer: This is a simple variation of the readers-writers problem, because the last thread must activate/do something. Compare the task of the n-th thread with what the last reader should do, and you should be able to see the similarity.

It is obvious that we need a counter `count` to count the number of waiting threads. Initially, `count` should be 0. Based on the specification, we need two semaphores: `Mutex` for protecting the counter `count`, and `WaitingList` for blocking threads.

When a thread calls `Barrier_wait()`, it locks the counter, and checks to see if it is the `n`-th one. If it is not the `n`-th one, the thread releases the lock and waits on semaphore `WaitingList`. This portion is trivial. Note that the order of "releasing the lock" and "waiting on semaphore `WaitingList`" is important. Otherwise, a deadlock will occur. (Why?)

If the thread is the `n`-th one, it must release all waiting threads that were blocked on semaphore `WaitingList`. Since we know there are exactly `n` − 1 waiting threads, executing `n` − 1 signals to semaphore `WaitingList` will release them all. Then, the `n`-th thread resets the counter and releases the lock.

Based on this idea, the following is the `Barrier` class:

```
class Barrier {
   private:
      int       Total;           // total number of threads in a batch
      int       count;           // counter that counts blocked threads
      Semaphore WaitingList(0);  // the waiting list
      Semaphore Mutex(1);        // mutex lock that protects the counter
   public:
      Barrier(int n) { Total = n; count = 0 };  // constructor
      Barrier_wait();                    // the wait method
};

Barrier::Barrier_wait()
{
   int  i;

   Mutex.Wait();                 // lock the counter
   if (count == Total-1) {       // if I am the n-th one
      for (i=0; i<Total-1; i++)  // release all waiting threads
           WaitingList.Signal();
      count = 0;                 // reset counter
      Mutex.Signal();            // release the lock
   }                             // I am done
   else {                        // otherwise, I am not the last one
      count++;                   // one more waiting threads
      Mutex.Signal();            // release the mutex lock
      WaitingList.Wait();        // block myself
   }
}
```

The protection of the counter `count` must start at the very beginning and extend to the very end so that the blocked `n`-1 threads can be released in a "single" batch. In other words, when the execution flow enters the "then" part of the `if`, all blocked `n`-1 threads are released as a single group. Otherwise, we may have the following problems. **First**, we may release a newcomer rather than the threads that were waiting in the barrier prior to the release. **Second**, threads just released may come back (*i.e.*, fast-runner) and be released again. In this case, the same thread is released twice and one of the originally blocked threads is not released. Hence, this violates the specification that the blocked $n-1$ threads must be released.

If the `Barrier_wait()` method is rewritten as the following to "increase efficiency," we will have problems:

```
Barrier::Barrier_wait()          // incorrect version
{
   int  i;

   Mutex.Wait();                 // lock the counter
   if (count == Total-1) {       // I am the n-th one
      count = 0;                 // reset counter
      Mutex.Signal();            // release the lock
      for (i=1; i<=Total-1; i++) // release all waiting threads
           WaitingList.Signal();
   }                             // I am done
   else {                        // otherwise, I am not the last one
      count++;                   // one more waiting threads
      Mutex.Signal();            // release the mutex lock
      WaitingList.Wait();        // block myself
   }
}
```

With this version, a thread just released from semaphore `WaitingList` may come back and call `Barrier_wait()` again. This thread can immediately change the value of `count` and wait on `WaitingList` again while the original `n`-th thread is still in the process of releasing the blocked `n`-1 threads. Since we cannot make any assumption about the order used for releasing threads, it is possible that a fast running thread is released again and one of the originally blocked thread will be blocked and released the next run. Or, it may be blocked forever!

The following is a similar solution. In this solution the `n`-th thread signals `n` times so that it will release itself at the end. However, this "solution" does have the *fast-runner* problem. Suppose `n` is 2. The first thread blocks as usual. When the second comes, it signals `WaitingList` twice, releases the `Mutex` lock, and is switched out by a context switch. Since the first thread was released by one of the two signals, it may come back, go through all steps, and wait on semaphore `WaitingList` faster than the second thread does. Since `WaitingList` was signaled twice, this returning thread is not blocked and can pass through. As a result, the same thread is released twice and the releasing thread is blocked. Of course, this is terribly wrong!

```
Barrier::Barrier_wait()          // incorrect version
{
   int  i;

   Mutex.Wait();                 // lock the counter
   if (count == Total-1) {       // I am the n-th one
      count = 0;                 // reset counter
      for (i=1; i<=Total; i++)   // release all waiting threads
           WaitingList.Signal(); //    including myself
   }
   else                          // otherwise, I am not the last one
      count++;                   // one more waiting threads
   }
   Mutex.Signal();               // release the mutex lock
   WaitingList.Wait();           // block myself
}
```

■

(b) **[20 points]** Using ThreadMentpr to design a Hoare monitor `Barrier` and method `Barrier_wait()` that fulfill the following specification:

- The constructor (*i.e.*, initialization) of monitor `Barrier`, `Barrier(int n)`, takes a positive integer argument `n`, and initializes a private `int` variable of the monitor to have the value of `n`.
- Method `Barrier_wait(void)` takes no argument. A thread that calls `Barrier_wait()` is blocked if the number of threads being blocked, including this one, is less than `n`, where `n` is the initialization value and will not change in the execution of the program. Then, the `n`-*th* calling thread will release all `n`-1 blocked threads. Suppose `n` is initialized to 3. The first two threads that calls `Barrier_wait()` block. When the third thread calls `Barrier_wait()`, all three threads will continue. Note that your solution **cannot** assume `n` to be 3. Otherwise, you will receive **zero** point.

Use ThreadMentor syntax to write the monitor code. **You must elaborate and justify your solution. Otherwise, you will receive low or even no grade. Hint:** This problem looks easy; but, if you forget this is a Hoare monitor you could end up with an *incorrect* solution.

Answer: If you know the semantics of the Hoare-style monitor well, this is actually an extremely easy problem. With a Hoare-style monitor, the signaling process (or thread) yields the monitor to the released process (or thread) **immediately**. In other words, if there are waiting threads on a signaled condition variable, one of them will take over the monitor and run. *This is the key to the solution of this problem.*

The following is a possible solution. It uses a condition variable `block` to block those threads that have to wait, variable `count` to count the number of blocked threads, and variable `n` for the control purpose. Note that `count` is initialized to `0`.

```
class Barrier : public Monitor {
     public:
          Barrier(int);        // constructor
          Barrier_wait(void);  // monitor procedure Access()
     private:
          Condition  block;    // C.V. for blocking threads
          int        n;        // maximum number of threads
          int        count, i; // working variables
};

Barrier::Barrier(int Max): Monitor(HOARE)  // constructor
{
     n     = Max;
     count = 0;
}
```

The following is the `Barrier_wait()` method. If the current `count` is less than `n - 1`, the calling thread increases the value of `count`, indicating one more waiting thread arrived, and waits. The key is the way of using `Signal()`. If the current `count` is `n - 1`, the newcomer should take the responsibility of releasing the `n - 1` previously blocked threads. In the `else` part, only one call to `Signal()` is made. This `Signal()` releases one blocked thread, which will reduce the `count` and `Signal()` the condition variable `block`. In this way, we start a "chain of reaction" and each released thread releases another thread. This is usually referred to *cascade release* or *cascade signal*.

```
void  Barrier::Barrier_wait(void)
{
     MonitorBegin();              // enter monitor
          if (count < n-1) {      // if not full count
               count++;           // increase the counter
               block.wait();      // wait to be released
               count--;           // reduce count
               block.signal();    // release one more blocked
          }
          else
               block.signal();    // initiate cascade release
     MonitorEnd();                // exit monitor
}
```

There are a few important notes about this solution:

- For the sake of simplicity, assume that the n-th thread T_n releases thread T_{n-1} that was blocked earlier, thread T_{n-1} releases thread T_{n-2}, ..., and thread T_2 releases T_1.
- When thread T_i releases T_{i-1}, T_i yields the monitor to thread T_{i-1}, and T_i and T_{i-1} become inactive and running, respectively, in the monitor. Consequently, starting from T_n's `Signal()` call to the release of thread T_1, the monitor is *non-empty*, and no other threads can enter the monitor. Therefore, all n threads, T_n included, are released in the same "batch" and no other threads can have any influence on this "chain of action."
- Note that the `Signal()` from T_1, the last released thread, is ignored because there is no waiting thread on condition variable `block`. Additionally, even though T_1's `Signal()` yields the monitor to a new thread not in this group, it does not matter because all n threads have been released properly, and will exit the monitor once they become running from inactive.
- From the above discussion, the value of `count` is maintained properly. In other words, the value of `count` increases by one when a new thread successfully enters the monitor via the call to `Barrier_wait()` until the value of `count` becomes `n - 1`. Then, the next thread's `Signal()` initiates cascade release, one at a time. Each released thread decreases the value of `count` by `1` before it yields to the next thread with `Signal()`. Since there cannot be any new threads in the monitor other than those involve in this chain, the value of `count` goes down to `0` steadily.
- In this way, thread T_n successfully releases the $n-1$ threads that called `Barrier_wait()` before T_n does.

Some added a number of statements, possibly with a new condition variable, to the beginning of `Barrier_wait()` to prevent "intruders" from entering when releasing threads. This is not necessary if your code can take advantages of the mutual exclusion property of a monitor. Additionally, once threads can get into `Barrier_wait()`, it is difficult to guarantee the generation of a correct batch of n threads.

Many used a `for` loop to release threads from condition variable `block`. A typical way is shown below. This is a risky move. Consider the following scenario. Suppose T_n is the n-th thread calling `Barrier_wait()` and starts releasing $n-1$ threads blocked on condition variable `block`. When T_n signals, it yields the monitor to a released thread, say T_1. Then, T_1 reduces the value of `count` by 1 and leaves the monitor. Now, the monitor is empty with T_n being inactive. What if T_1 comes back again (*i.e.*, fast-runner) and calls `Barrier_wait()`? Since we should not assume who would get the monitor between T_n and the newcomer T_1, it may well be T_1. If this

is the case, T_1 increases the counter `count` and blocks itself. Suppose T_n is picked to run (*i.e.*, from inactive to running). T_n's second signal may cause T_1 to be released again because one cannot assume any policy for releasing threads from a condition variable. As a result, we have an incorrect "batch" as T_1 is released twice!

```
void  Barrier::Barrier_wait(void)
{
     MonitorBegin();             // enter monitor
          if (count < n-1) {     // if not full count
               count++;          // increase the counter
               block.wait();     // wait to be released
               count--;          // reduce the counter
          }
          else
               for (int i = 1; i < n; i++)
                    block.signal(); // release n-1 blocked threads
     MonitorEnd();               // exit monitor
}
```

Some may suggest the following similar but a bit subtle solution. This solution is more "efficient" because it resets `count` rather than decreasing it by one every time a thread is released. If a fast runner comes back, it finds out `count` is still `n`-1 (since the releasing thread is inactive in the monitor and not able to reset `count`), and executes the `for` loop. As a result, we have **two** releasing threads!

```
void  Barrier::Barrier_wait(void)
{
     MonitorBegin();             // enter monitor
          if (count < n-1) {     // if not full count
               count++;          // increase the counter
               block.wait();     // wait to be released
          }
          else {
               for (int i = 1; i < n; i++)
                    block.signal(); // release n-1 blocked threads
               count = 0;        // reset counter
          }
     MonitorEnd();               // exit monitor
}
```

What if the `else` part is replaced as follows? A fast-runner will come back and wait on condition variable `block`. It may be released in this batch! Again, this is an incorrect solution.

```
          else {
               count = 0;        // reset counter
               for (int i = 1; i < n; i++)
                    block.signal(); // release n-1 blocked threads
          }
```

■

Final *Solutions*

CS 414 Operating Systems and Systems Competency Exam, Spring 2007
May 16th, 2007
Prof. Hakim Weatherspoon

Name (or Magic Number): _____NetId/Email:_____

Read all of the following information before starting the exam:
If you are a CS 414 student, write down your name and NetId/email NOW. Otherwise, if you are taking this as your Systems Competency Exam, write down your *Magic Number* (do ***NOT*** write your name, NetId, or Email).

This is a **closed book and notes** examination. You have 150 minutes (2 ½ hours) 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	20	
2	24	
3	15	
4	21	
5	20	
Total	**100**	

1. (20 points) Synchronization/Concurrency Control

For the following implementations of the "H20" problem, say whether it either (i) works, (ii) doesn't work, or (iii) is dangerous -- that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why and show how to fix it so it does work.

Here is the original problem description: 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) (10 points) Here is a proposed solution to the "H20" problem:

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

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

    return;
}
```

```
oReady()
{
    P(o_wait);
    makeWater();
    V(h_wait);
    V(h_wait);

    return;
}
```

This solution is dangerous. Threads calling hReady() *access shared data without holding a lock! For example, you could have N threads increment* count*, and because they do so without a lock, the result could be 1 instead of N -- in other words, no water would be made regardless of how many* H*'s arrived.*

The solution is to put a lock acquire before the first line in hReady*, and release before the first* P(h_wait) *and after the second* P(h_wait)*. Some put the lock acquire after the increment, and that simply doesn't work!*

(b) (10 points) Another proposed solution to the "H20" problem:

```
Semaphore h_wait = 0;
Semaphore o_wait = 0;

hReady()
{
    V(o_wait)
    P(h_wait)

    return;
}
```

```
oReady()
{
   P(o_wait);
   P(o_wait);
   makeWater();
   V(h_wait);
   V(h_wait);

   return;
}
```

This is dangerous, since it may lead to starvation. If two H*'s arrive, then the value of the* o_wait *semaphore will be 2. If two* O*'s arrive, then they can each decrement* o_wait*, before either can decrement it twice. So no water is made, even though enough atoms have arrived.*

The fix is to put a lock acquire before the first line in oReady*, and a lock release after the two* V(h_wait)*'s. This way, only one oxygen looks for waiting* H*'s at a time -- if there aren't enough* H*'s for the first oxygen, there won't be enough for any of the later oxygens either.*

2. (24 points) Synchronization via Monitors

Some monkeys are trying to cross a ravine. A single rope traverses the ravine, and monkeys can cross hand-over-hand. Up to five monkeys can hang on the rope at any one time. If there are more than five, then the rope will break and they will all fall to their end. Also, if eastward-moving monkeys encounter westward-moving monkeys, all will fall to their end. (This is the same problem setup from an earlier assignment, but the synchronization mechanism differs).

Assume that monkeys are processes.
(a) (18 points) Write a *monitor* with two methods WaitUntilSafeToCross(Destionation dst) and DoneWithCrossing(Destination dst). Where Destination is an enumerator with value EAST=0 or WEST=1.
(b) (6 points) Does your solution suffer from starvation? If so, briefly explain (e.g. give a sequence). Otherwise, simply state *starvation-free*. In either, case state your assumptions, if any.

The solutions below are starvation-free.

```
int crossing[2] = {0, 0}, waiting[2]= {0, 0};
Condition wantToCross[2];

WaitUntilSafeToCross(Destination dst)
{
        if(crossing[!dst] > 0 || waiting[!dst] > 0 || crossing[dst] == 5)
        {
                ++waiting[dst];
                wantToCross[dst].wait();
                --waiting[dst];
        }
        crossing[dst]++;
}

DoneWithCrossing(Destination dest)
{
        --crossing[dst];
        if (crossing[dst] == 0)
                wantToCross[!dst].signal();
        else if (waiting[dst] > 0 && waiting[!dst] == 0)
                wantToCross[dst].signal();
}
```

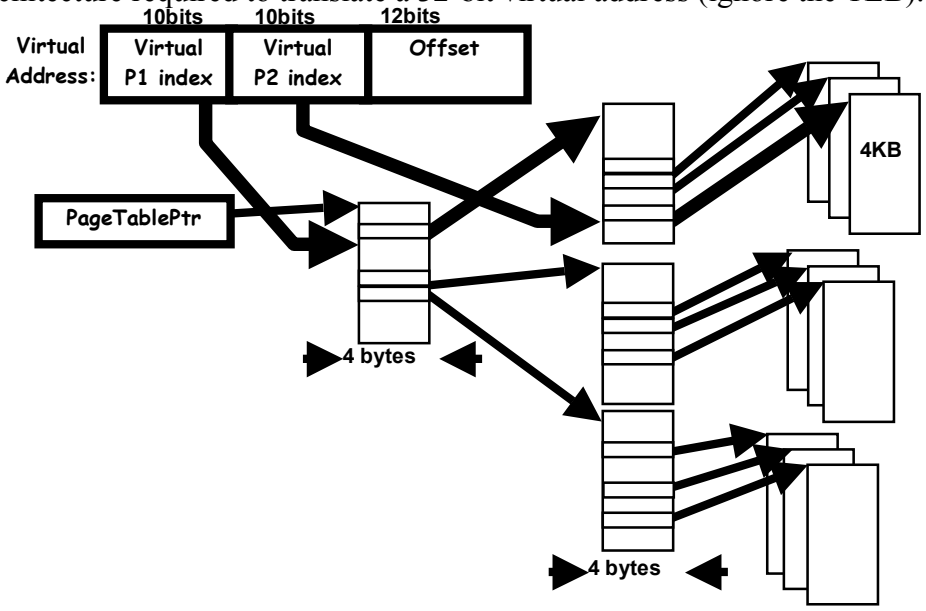
Alternative implementation

```
int wcrossing=0, ecrossing=0, wwaiting=0, ewaiting=0;
Condition wantToCrossWest, wantToCrossEast;

WaitUntilSafeToCross(Destination dest)
{
        if(dest == EAST)
                WantToGoEast();
        else
                WantToGoWest();
}

DoneWithCrossing(Destination dest)
{
        if(dest == EAST)
                DoneGoingEast();
        else
                DoneGoingWest();
}
WantToGoEast()
{
        if(wcrossing > 0 || wwaiting > 0 ||
          ecrossing == 5)
        {
                ++ewaiting;
                wantToCrossEast.wait();
                --ewaiting;
        }
        ecrossing++;
}

DoneGoingEast()
{
        ecrossing--;
        if (ecrossing == 0)
                wantToCrossWest.signal();
        else if (ewaiting>0 && wwaiting==0)
                wantToCrossEast.signal();
}
```

```
WantToGoWest()
{
        if(ecrossing > 0 || ewaiting > 0 ||
          wcrossing == 5)
        {
                ++wwaiting;
                wantToCrossWest.wait();
                --wwaiting;
        }
        wcrossing++;
}

DoneGoingWest()
{
        wcrossing--;
        if (wcrossing == 0)
                wantToCrossEast.signal();
        else if (wwaiting > 0 && ewaiting==0)
                wantToCrossWest.signal();
}
```

3. (15 points) Virtual Memory and Paging

a. (5 points) Suppose that we have a two-level page translation scheme with 4K-byte pages and 4-byte page table entries (includes a valid bit, a couple permission bits, and a pointer to another page/table entry). What is the format of a 32-bit virtual address? Sketch the paging architecture required to translate a 32-bit virtual address (ignore the TLB).



Grading Scheme: 2 point for the virtual address format (1/2 for [L1 page index, L2 page index, offset] and 1/2 point for # bits = 10,10,12) , 1 point for picture of single level page table,1 point for multiple page tables linked together correctly, and 1 point for the page table base register (PageTablePtr).

b. (10 points) Assume that a process has referenced a memory address not resident in physical memory (perhaps due to demand paging), walk us through the steps that the operating system will perform in order to handle the page fault.

Note: We are only interested in operations that the OS performs and data structures modified by the OS.

1. Trap to the OS
2. Save the process state (program counter, stack pointer, registers, etc)
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on disk
In particular, if the reference was invalid, terminate the process. If it was valid, but have not yet brought in that page, page it in.
5. Find a free frame (by taking one from the free-frame list, for example)
a. If there is a free frame, use it.
b. If there is no free frame, use a page-replacement algorithm to select a victim *frame.*
c. Write the victim frame to disk (if it was dirty)
i. Change the page table to mark the frame invalid (i.e. not present in memory).

6. Issue a read from the disk to the free frame:
a. Wait in a queue for this device until the read request is serviced
b. Wait for the device seek and/or latency time
c. Begin the transfer of the page to the free frame
7. While waiting, allocate the CPU to some other process
8. Receive an interrupt from the disk I/O subsystem (I/O completed)
9. Save the registers and process state for the other process (if step 7 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory
11. Add the process to the ready queue
12. When process selected to run, restore user registers, process tate, and new page table, and then resume the interrupted instruction.

4. (21 points) File Systems

Adding Links to a File System. This design question asks you to consider adding links to a file system that does not have any linking mechanism. ***This is a design question; you should not write code implementation unless stated explicitly.***

a. (12 points) The first set of questions is about adding **hard** links to the file system.

i. (3 points) What changes would you make to the internals (directory, fileheader, free map, etc.) of the filesystem to support hard links?

You would have to add a link count to the file header. No partial credit.

ii. (3 points) How do the semantics of the Remove system call change, and how do you implement that change to Remove and any other affected system calls?

Remove must decrement the link count (1 points) and check the on disk link count (2 points). No other system calls are affected and we deducted up to 2 points if you included other system calls.

iii. (3 points) What new system calls, if any, must be added to the system? Give the C language-style signature of any new calls, e.g., int Open(char *file).
List the signature and give a one sentence definition of each arg and return value.

*Add a int Link(char *src, char *dst) or int Link(int inodeNumber, char *dst) to create a new link to an existing file.*

iv. (3 points) Use the Remove system call and answer to question part iii above to show the implementation of the Rename system call.

```
int Rename(char *old, char *new) {
        Remove(new);
        Link(old, new);
        Remove(old);
}
```

b. (9 points) This set of questions is about adding **soft** links to the file system. Soft links are also called symbolic links.

i. (3 points) What changes would you make to the internals (directory, fileheader, free map, etc.) of the filesystem to support soft links?

Add a new type of file to the directory or the file header (2 points), and store the soft link's pathname in a file (1 points).

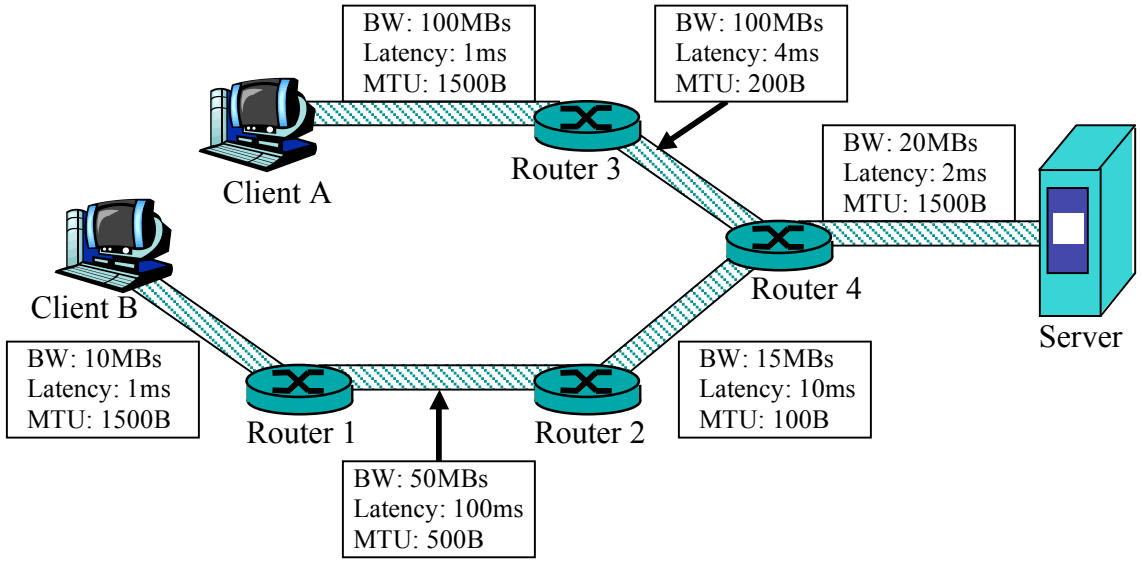
ii. (3 points) What existing system calls have to be modified to support soft links? How do the system calls need to be modified?

Open needs to be changed to implement the recursive lookup implied by soft links (1 point). If the file being opened resolves to a soft link, Open must open the file pointed to by the soft link (2 points). The dereferencing operation should count the number of dereference operations to prevent an error from loops (1 point). Remove must be modified in a minor way to remove soft links correctly. If you removed the target of a link (instead of the link), we deducted 2 points. If you said that system calls use file descriptors instead of names, we deducted 2 points.

iii. (3 points) What new system calls, if any, must be added to the system? Give the C language-style signature of any new calls, e.g., int Open(char *file).
List the signature and give a one sentence definition of each arg and return value.

*Add a int SymLink(char *src, char *dest) to create a new symbolic link to an existing file. No partial credit.*

5. (20 points) Networking



The above figure illustrates a network in which two clients (Client A and Client B) route packets through the network to the server. Each link is characterized by its Bandwidth (BW), one-way Latency, and Maximum Transfer Unit (MTU). All links are full-duplex (can handle traffic in both directions at full bandwidth).

a. (4 points) Under ideal circumstances, what is the maximum bandwidth that Client A can send data to the server without causing packets to be dropped (Assuming that the headers are of zero length)? How about Client B? Explain.

Here we are looking for the bottlenecks in the bandwidth and all we need to do is find the minimum bandwidth along the paths from the clients to the server. For A the minimum along its path to the server is 20MBps. And for B it is 10MBps

- We gave 2 points per correct answer (and took off 1 point if you swapped A & B) A lot of people tried to add in a lot of fancy stuff here, but there wasn't anything more complicated than finding bottlenecks.

b. (4 points) Keeping in mind that TCP/IP involves a total header size of 40 bytes (for TCP + IP), what is the maximum data bandwidth that Client A could send to the server through TCP/IP? Explain.

Here the interesting part was noticing that we had to take the minimum MTU to tell us what the largest packet we can create is. We see that A→S has a min MTU of 100 bytes of which 40bytes are used for header. This leaves us with 60 useable bytes of payload. Thus only (200-40)/200 = 80% of the bandwidth will be useful data bandwidth. Leaving us with 80% × 20MBps = 16MBps

- We gave 1 point for realizing that the min MTU was 200 bytes. 1 point for correctly seeing that only 80% of the bandwidth was usable and 2 points for multiplying the 80% with the correct bandwidth to give the maximum data bandwidth.

- Note: if you assume that the sender is not using an algorithm to avoid fragmentation, then they may try to send packets of size 1500. These will get fragmented into 15 IP packets when they cross the "MTU bottleneck". So, we will have 8×20 (IP header size) + 20 (TCP header size) overhead = 180/1500 ⇒ (1500-180) /1500 data = 22/25 data = 0.88%.

c. (4 points) Assume that Client A sends a continuous stream of packets to the server (and no other clients are talking to the server). How big should the send window be so that the TCP/IP algorithm will achieve maximum bandwidth without dropping packets? Explain. Hint: don't forget to account for the 40 bytes of header.

Remember that the optimal window size = roundtrip latency × effective bandwidth
The roundtrip latency along A's path to the server was (1+4+2)×2 = 14ms. The effective bandwidth from part b was 16MBps so the total was 16MBps × 0.014s = 0.224 MB
- We gave 2 points for telling us how to correctly calculate the correct window size, 1 point for giving the right latency term, and 1 point for the calculation itself.

d. (4 points) Assume that Clients A and B both send a continuous stream of packets to the server simultaneously. Assume that Bandwidth is shared equally on shared links. What must the sizes of their send windows be so that packets are not dropped? Explain.

Now the link between router 4 and the server gets shared equally between A and B and thus the bandwidth for each of them drops to 10MBps.
From part b we know that the MTU is still 200 bytes and therefore we can at most get 80% of the bandwidth so A's window size is 10 Mbytes/s × 0.014 s × 0.8 = 112 kB
- For B we need to calculate its effective bandwidth. We see that the min MTU is 100 bytes and therefore (100-40)/100 tells us that we can achieve 60% of the bandwidth. We also notice that B's roundtrip latency to the server is only 2×(1+100+10+2)=226ms. So running the same calculation as A we get B's window size is 10 Mbytes/s × 0.226s × 0.6 = 1.356 MB
- There were two points to each part and were distributed according to same ratios as part (c)

e. (4 points) Consider the situation of (5d). If the server is sending data back to Clients A and B at full rate, do the acknowledgements headed to the clients decrease the bandwidth in the forward direction (clients→server)? State your assumptions and explain your answer.

There were two answers here we accepted. The correct answer is that it doesn't affect the bandwidth because the acks are piggy backed on the reverse data packets and therefore the there is no loss of bandwidth for the acks.

COS318 Final Exam SOLUTIONS
Princeton University
Fall, 2011
Instructors: Profs. Margaret Martonosi & Vivek Pai

(Total Time = 180 minutes)

Question	Score
1-3	/15
4	/15
5	/15
6	/15
7	/10
8	/15
9	/15
Total	/100

- **This exam is closed-book, closed-notes. 1 double-sided 8.5x11" sheet of notes is permitted.**
- **Calculators are ok but unneeded. Laptop or palmtop computers are not allowed.**
- **Show your work clearly in the spaces provided in order to get full or partial credit.**
- **Excessively long and/or vague answers are subject to point deductions.**
- **If you are unclear on the wording/assumptions of a problem, please state your assumptions explicitly and work it through.**

Honor Code (Please write out these words and then sign): I pledge my honor that I have not violated the Honor Code during this examination.

MAX SCORE = 98.5

Mean = 76.5

Std Dev = 11.8

Print name clearly and sign:

Short Answer:

1. (3 points) What is a TLB and what does it do?

A TLB is a form of hardware cache that tries to store a small number of recently-used virtual-to-physical page translations, so that V-to-P mappings can be performed at hardware speeds, without invoking the OS on each memory reference.

2. (6 points) One program reads a large file via malloc/read and the other uses mmap. If the OS needs to evict dirty pages from these regions, where do they go?

The malloc/read program has its pages written to swap. Note that the read itself makes the pages dirty from the standpoint of the OS.

The mmap'd program would have its dirty pages written back to the file in question unless you told mmap to do otherwise. The question assumed that there were dirty pages, but if you said that reading via mmap doesn't cause pages to become dirty, so the OS can just discard the pages instead of writing them back to the file, that was also accepted.

3. (6 points) A multi-threaded database program uses a single file to store all customer balance information. Each thread performs a seek, followed by a read to obtain the old balance, and then a write to update the balance.

a) (3 points) Using pread and pwrite can avoid the race conditions inherent in the original design. Explain why

The original program can have a race if the two seeks occur before the reads. Both reads will occur at the location of the second seek. Other combinations are possible too. The pread/pwrite call specifies the location of the read/write, so there's no implied position that can be changed by other calls. Note that pread/pwrite don't lock the entire file, which is the whole point.

b) (3 points) If the system runs on a single disk, is it preferable to use pread/pwrite or just have each thread lock the file during a transaction?

With a single disk, it's actually better from a performance standpoint to lock the file so that the head doesn't move between the read and write. With pread/pwrite, you may service the next pread before seeing the pwrite, so you'll end up doing more seeks.

4. (15 points) Virtual Memory: Recall the x86 page table structure from project 5: given a 32-bit linear address, the high-order 10 bits index into the page directory, the next 10 bits index into the page table, and the low-order 12 bits index into the page.

Consider each the following changes **separately** (i.e., they are **NOT** cumulative):
a) (8 points) Given a linear address, the high-order 8 bits index into the page directory, the next 8 bits index into the page table, and the low-order 16 bits index into the page. What is the size of a page, in bytes, in the new system? What are the pros and cons of the change? Give an example usage pattern under which the change would be beneficial.

2 points:
2^16 = 65536

4 points:
Pros
- less overhead due to paging structures and swapping to/from disk
- more of virtual memory fits into the TLB at once
Cons
- more wasted space due to internal fragmentation
- less granularity for swapping
- swapping an individual page is more expensive

2 points:
Example workload: random access of a 64 KB array.

b) (7 points) Given a linear address, the low-order 10 bits index into the page directory, the next 10 bits index into the page table, and the high-order 12 bits index into the page. Will this work? If so, is it a good or bad idea? Justify your answer.

correct answers: 3 points
justification: 4 points
It will work, since there's still a valid mapping between virtual and physical addresses. It's a bad idea, though, because it eliminates the benefits of spatial locality. Neighboring addresses will translate to different pages, leading to high overhead in paging data structures and frequent swapping.

5. (15 points) A UNIX filesystem has 2-KB blocks and 4-byte disk addresses. Each i-node contains 10 direct entries, one singly-indirect entry and one doubly-indirect entry.

a) (5 points) What is the maximum file size?
10*2KB + (2048/4)*2KB + (2048/4) * (2048/4) *2KB = 20K + 1024K + 524288K = 525332KB (or 537939968B or 513.02MB etc.)
If you assume 1K equals to 1000 instead 1024, your answer is still considered to be correct.

Rubric: If the final result is correct, you will get all 5 points. Otherwise, size of singly- indirect worth 1.5 points; size of doubly-indirect worth 1.5 points; the formula of adding up everything together worth 1.5 points; final result worth 0.5 point.

b) (5 points) Suppose half of all files are exactly 1.5-KB and the other half of all files are exactly 2-KB, what fraction of disk space would be wasted? (Consider only blocks used to store data)

Both 1.5-KB and 2-KB file will use 2KB space. For each 2-KB file, 0KB is wasted; for each 1.5-KB file, 0.5KB is wasted. Therefore, the fraction wasted is (0/2)*50%+(0.5/2)*50% = 12.5%.

Rubric: If the final result is correct, you will get all 5 points. Otherwise, the correct formula worth 4 points final result worth 1 point.

c) (5 points) Based on the same condition as in b), does it help to reduce the fraction of wasted disk space if we change the block size to 1-KB? Justify your answer.

No. Nothing is changed. Both 1.5-KB and 2-KB file will still use 2KB space. For each 2-KB file, 0KB is wasted; for each 1.5-KB file, 0.5KB is wasted. Therefore, the fraction wasted is (0/2)*50%+(0.5/2)*50% = 12.5%, which is unchanged.

Rubric: No (2 points). Reasonable reason (3 points).

6. (15 points) Virtual Machines. Virtualized environments seek to provide isolation between different workloads by allowing multiple guest OS's to run distinct workloads simultaneously on one physical system. In theory, someone running a program within a perfectly-virtualized environment would find it indistinguishable from running on bare hardware. In practice, discuss some ways that a piece of running code might be written to be able to discern the difference.

The best strategies all revolve around timing the performance of functionality known to highlight differences between native and virtualized execution. The three main examples of this would be: specific instructions like (1) x86 popf, (2) I/O functionality, (3) memory management and allocation.

A reasonable but general discussion of timing strategies got 12 points. Detailed specifics on the categories above got 15. Attempts to get at the above sorts of information without mentioning/using timing usually earned 10 points, since they usually wanted to look at info unavailable to a virtualized client.

7. (10 points) Your boss decides that your first project will be to take a standard Unix filesystem, and make one change -- to place the full inode in the directory file instead of just placing the inode number.

a. (5 points) What benefits does this change provide? For each benefit, briefly explain why.

The biggest is that it reduces the number of seeks, since you don't have to do a separate seek for the inode. It also gives you more flexibility on the number of inodes, since they're only allocated as needed. This will either save space when you don't need as many inodes, or will give you greater flexibility when you need a lot more inodes. It may also save some space by using up more of the directory file for smaller directories. It may also help lessen the impact of damage to the disk, since scattered inodes reduce the chance of lots of inodes getting wiped out at the same time.

b. (5 points) What drawbacks/limitations does this change introduce. Briefly explain each one.

Hard linking becomes much harder, and you need some other mechanism for it. Without some other mechanism, hard links become impossible. Finding inodes also requires a new mechanism. Detecting corruption and recovering from it becomes more complicated since you have no idea where to look for inodes.

8. (15 points) Consider the following simplified implementation of the link() system call from an early Unix system.

algorithm link:
input: existing file name A, new file name B
output: none

```
{
1       get inode for existing file A; /* returns a locked, reference counted inode */

2       if (( too many links on file ) or (linking directories without superuser permission)) {
3         release inode A; /* removes reference count and unlocks the inode */
4         return (error);
      }
5       increment link count on inode A;
6       update disk copy of inode A;
7       unlock inode A;

8       get inode for parent directory to contain new file B; /* returns a locked, ref-counted
inode */
9       if ( ( new file name already exists ) or ( existing file, new file on different file systems ) ) {
10        undo update done above;
11        return (error);
        }
12      create new directory entry in parent directory of new file B:
13            include new file name B, inode number of existing file A

14      release parent directory inode for B; /* decrements ref-count and unlocks the inode */
15      release inode of existing file A; /* decrements ref-count */
}
```

Now answer the following questions, with no more than 3-4 sentences per part.

a. (3 points) You see mentions of "link counts" and "reference counts", both on inodes. What's the difference between them?

Link counts keep track of the number of directory entries anywhere under the filesystem root which refer to the file in question. The inode reference count is used to count the number of threads/processes¬ actively modifying the inode.

Scoring: 1.5 points each for link count and reference count definition/illustration of difference. 1 or 2 points deducted if you mention any incorrect usages of these counts.

b. (4 points) Why did early Unix system implementations require superuser permission to link directories (line 2)?

To simplify system call implementations having to deal with loops in the file system hierarchy--e.g., if a user were to link an existing directory (file A) from a node below it (file B) in the hierarchy. (Superusers are trusted to know what they're doing...) So, now system call implementations can assume that anything other than "../" does not lead it *up* the file system hierarchy.

Scoring: 2.5 points for mention of loops or circular links; 1.5 points for explaining why loops are bad.

A common answer to this question was that this helps avoid users from getting access to protected parts of the filesystem (e.g., /etc) or to other users' home directories. But this is incorrect because the file owner, group and permission information resides on the inodes, which are checked when the file is actually being accessed.

c. (8 points) Why do lines 6-7 update the disk copy and unlock inode A *before* it is clear that the link call can be successfully completed? (i.e., before the checks in lines 8-11)

The essence of this question is: why unlock inode A before including the new inode entry for B, and not after? The answer is that you would end up with the classic deadlock situation of acquiring resource locks in reverse order. These bad orderings of lock acquire()s are hard to foresee, because link()s can in principle be called from any node on the file system.

A concrete example: Suppose thread A wants to link "/e/f/g" (new file) to "/a/b/c/d" (existing file) and thread B wants to link "/a/b/c/d/ee" (new file) to "e/f" (existing file). Consider what happens if A finds and locks the inode for "/a/b/c/d" at the same time that B finds and locks the inode for "/e/f". Now they are deadlocked waiting to obtain the locks for the second half of the system call to complete.
It's important to note that the need to unlock the inode drives the need to update the disk copy, and not the other way around.

Scoring: If you mention the possibility of a deadlock, you receive 5 points. A correct, plausible example sequence of operations where deadlock happens gets you 3 more points.

If you don't mention deadlocks (with or without example) but provide valid performance or fault tolerance reasons, you get 2 points. These reasons don't address why you need to unlock the inode (though they may address why you update the disk copy).

9. (15 points) RAID. Consider that many RAID devices now ship with the following options:
RAID 0 - data striped across all disks
RAID 1 - each disk mirrored
RAID 5 - striped parity

Assume a system with 8 disks

For each level, how much usable storage does the system receive?
RAID 0 – 8 disks
RAID 1 – 4 disks
RAID 5 – 7 disks

Assume a workload consisting only of small reads, evenly distributed. Assuming that no verification is performed on reads, what is the throughput of each level assuming one disk does 100 reads/sec?
RAID 0 – 800 reqs/sec
RAID 1 – 800 reqs/sec – reads can be satisfied from both disks in a pair
RAID 5 – 800 reqs/sec – no need to read the parity, so no loss of read performance, only space

Assume a workload consisting only of small writes, evenly distributed. Again, calculate the throughput assuming one disk does 100 writes/sec
RAID 0 – 800 reqs/sec
RAID 1 – 400 reqs/sec – need to write to both disks in a pair
RAID 5 – 200 reqs/sec if you do two reads + two writes to update the parity, or 100 reqs/sec if you read all of the disks to recalculate the parity

For each level, what is the minimum number of disks that may fail before data **may** be lost?
RAID 0 – 1, but data loss is guaranteed at the first lost disk
RAID 1 – 2, if you happen to lose both disks in a pair
RAID 5 – 2, but data loss is guaranteed on the second disk

For each level, what is the minimum number of disks that must fail to **guarantee** data loss?
RAID 0 - 1
RAID 1 – 5, if you happen to get really lucky and lose one from each pair before losing the 5th
RAID 5 - 2

Final Exam Sample Questions
Solutions

SAMPLE PROBLEMS

1. Consider a FAT-based (File Allocation Table) file system. Entries in the table are 16 bits wide. A user wants to install a disk with 131072 512-byte sectors.
   a. What is a potential problem?

      Each 16 byte entry in the table is an address of a sector on the disk. The OS can address 65,536 sectors, and the disk has more than that.

   b. Describe a solution to this problem and explain the trade-offs involved.
      Make each FAT entry access a logical sector that is 2 physical sectors. This trades increased internal fragmentation against maintaining the size of the FAT, and backward compatibility.
      Alternatively, you can increase the size of the FAT to be 131,072 17-bit entries, which increases the table size to 278,528 bytes (at least) and increases the complexity of the decoding process. It also ruins any backward compatibility.

2. Generally we've talked about each operating system component in isolation. This question asks you to think about ways in which they interoperate. For each pair of systems below, give a specific way that they interact (or that they could interact). Writing that the file system and I/O system interact because they both use the disk is not worth more than a point, and may be worth none. Writing that the file system and I/O system interact when they determine the mapping from logical blocks → physical blocks which impacts the size of file system structures, and the efficiency of the disk usage because larger logical blocks imply more internal fragmentation on the disk is a more complete answer.
   a. How does a demand paged, lazy loaded virtual memory system interact with the process scheduling and creation system?

      A demand paging system will be at its worst at process creation, with an empty address space. The two systems have to work together to pre-fetch a reasonable working set without overloading the paging system by bringing in the entire address space. If the VM system supports shared text (or copy on write), the two need to work together to take advantage of it.

   b. Name another way (not the example above) that the file system and the hard disk drivers in the I/O system interact.

      Feature coordination. If the disk implements interleaving, the file system implementation should not do its own interleaving. The result can be to make both systems worse that useless - actually detrimental. Similar negotiations have to be done for features like disk caching or bad block renaming.

3. On some computer, the clock interrupt handler needs 2 msec (including context switch overhead) per clock tick to execute, and the clock runs at 75 Hz. What fraction of the CPU time is devoted to the clock?

   *The clock interrupt handler is run 75 times every second, which takes 75*2 = 150ms. This is 15% of the CPU time.*

4. List the terms that best describe each of the following:
   a. Operating system code executed when an asynchronous device signals the CPU

      Interrupt handler

   b. A type of disk arm scheduling policy

      FIFO, Shortest seek first (SSF), SCAN, C-SCAN, or elevator algorithm

5. If the TCP transmission window for unacked bytes is 1000, the one-way latency of a cross-country network link is 50 milliseconds, and the bandwidth of the link is 100 Megabits/second, then how long does it take TCP to transmit 100,000 bytes across the link? That is, how much time elapses from when the first byte is sent by the sender to when the sender *knows* that the last byte has been received by the receiver? You may assume that no packets are lost for this particular problem (but remember that TCP doesn't know that).

   A TCP transmission window size of 1000 implies that the sender can send 1000 bytes before having to wait for an ACK message from the receiver that will allow it to continue sending again. Assume the ACK is essentially 0 bytes long. Then the sequence of messages sent is:

   1. *1000 bytes from sender to receiver: requires 50 ms for first byte to get there and another 1000/(100 Mbps / 8 bytes/bit) secs for the rest of the 1000 bytes to get there after that.*
   2. *ACK msg from receiver to sender: requires 50 ms to get there. To send 100,000 bytes will require 100 round trips of this kind. So the total time required is:*
      *100 * (2 * 50 ms + 1000/(100,000,000/8)) = 100 * (100 ms + 0.08 ms)*
      = 10008 ms
      = 10.008 seconds

6. Security
   a. How would you use public-key encryption to implement secure remote procedure call (RPC) between an arbitrary client A and server B. Assume that A knows B's IP address and public key but that B does *not* know A's IP address and public key. Do not assume the existence of a "public key" server. "Secure RPC" means that no one other than A and B can understand the contents of either the request message or the response message. Please use a combination of English and pseudo-code of the form (*data*)^K to indicate that *data* is encrypted/decrypted using key K to answer the question.

   A must send B its IP address and public key so that B can talk back to A. To prevent anyone from being able to understand the RPC request – suppose it is f(x), A encrypts everything with B's public key:

   (f(x), IP-A, KA-public)^KB-public

   B decrypts this message with it's private key:

   ((f(x), IP-A, KA-public)^KB-public)^KB-private ➔ f(x), IP-A, KA-public

   B can now execute the requested f(x). Suppose that returns a value y. B then sends back the return value by encrypting it with A's public key so that only A can understand the returned message:

   (y)^KA-public

   B sends this message to IP-A. A decrypts the return message using its private key:

   ((y)^KA-public)^KA-private ➔ y

b. Assume that there is a "public key" server, S, that stores IP addresses and public keys for everyone, including A and B. Suppose that both A and B know the IP address and public key for S, but do not know the IP address and public key of anyone else. How could A reach B and how could B authenticate A? By authenticate I mean: how could B be sure that an RPC request came from the client that it claims to have come from?

Since A doesn't know B's IP or public key it must get them from the public key server. It does that as in part (a), except that the RPC now goes to the public key server. Note: since the public key server only stores "public" information, we might be tempted to use insecure RPC when talking to it. However, that would offer an additional opportunity for someone who is monitoring network traffic to try and "hijack" messages and replace them with altered ones.

In order to convince B that A is really the sender of an RPC, A must "sign" its RPC message with its private key and say something that B can verify, like its identify: "I'm A". When B uses A's public key to decrypt the message and gets a correctly formed message it will know that only A could have signed it. Note that A must still tell B who it is in its RPC request, so that B will know whose public key to ask for from the public key server.

So, the sequence of steps is:

1. *A ➔ S: ("Need info for B", "I'm A")^KS-public*
2. *S: decrypt and get info.*
3. *S ➔ A: (mbox-B, KB-public)^KA-public 4.*
4. *A: decrypt and use info.*
5. *A ➔ B: ((f(x), "I'm A")^KA-private, "I'm A")^KB-public*
6. *B ➔ S: ("Need info for A", "I'm B")^KS-public*
7. *S: decrypt and get info.*
8. *S ➔ B: (mbox-A, KA-public)^KB-public*
9. *B: decrypt and use info.*
10. *B: (((f(x), "I'm A")^KA-private, "I'm A")^KB-public)^KB-private*
11. *B: (((f(x), "I'm A")^KA-private)^KA-public*
12. *B: Check that decrypted message provided by A consists of a properly formed remote procedure call request and the expected declaration of identity: "I'm A".*
13. *B: f(x) ➔ y*
14. *B ➔ A: (y)^KA-public*
15. *A: decrypt*

Note that this solution is subject to replay attacks: someone could replay the message sent from A to B and B would assume that the same RPC is being called a second time.

Consider how you could solve this problem if you had to solve it.

c. Explain how an intruder who can break into the public key server could snoop on and alter all RPC traffic in the system described in (b) without any of the RPC clients and servers being able to tell.

If an intruder compromised the public key server, then the intruder can feed everyone false values for IP addresses and public keys. (We assume that the intruder has also managed to gain the private key of the public key server.) When a client A asks for the IP address and public key of a server B, the corrupted public key server returns a IP address that belongs to it and returns its own public key. A will then unwittingly send its RPC request to the intruder instead of to B. The intruder decrypts the message, reads it, perhaps alters it (e.g. replace f(x) with f(z)), and then sends it on to B as if it were A. When B later asks the intruder for information about A the intruder again returns information that points to itself instead of to the real A. So B will think the RPC request that the intruder forwarded is actually coming from A. The same "waylaying and forwarding" will happen on the RPC reply message.

d. Explain how you would modify the system to prevent the problem identified in (c). Be sure to state any assumptions you make.

If we are worried that someone has corrupted the public key server then we need to create additional sources/repositories of the information that the public key server was maintaining for the system. Suppose there were three public key servers, all running independently, so that an intruder who managed to corrupt one of them couldn't also thereby corrupt the other two.
Clients and servers now talk to all three public key servers when asking for information instead of just to one. They compare the information received from all three and check to see if it all matches up. If at most one public key server can be corrupted then clients and servers will see two correct values for each piece of information and one incorrect value. They discard the incorrect values and use the correct values to proceed to talk to each other.

7. In class, we discussed copy-on-write for memory pages shared among multiple processes. We cannot apply this same concept blindly to process creation using Unix fork(), but instead are forced to copy some parts immediately while other parts can be delayed.
   a. Knowing the components of general processes, which parts must be copied immediately, and which parts can be delayed and copied-on-write?

      This question refers to processes, not caching. When we create a new process, we must copy the stack space and registers, but need not copy the entire address space immediately. This is because often a call to fork creates a new address space into which a new process is immediately loaded, making the initial copy a waste of time and space because it is immediately overwritten. An example of this is when a command shell starts up a new process, first doing a fork to create a new copy of itself, but then replacing that copy with the new process which was executed on the command line.

   b. Why is copy-on-write potentially better than copying the entire process immediately upon creation?

      It can save time and space during the process creation (as mentioned above), avoiding the duplicate effort of making a copy, then immediately overwriting it.

8. Briefly describe the steps taken to read a block of data from the disk to the memory using DMA controlled I/O.

    1. *CPU programs the DMA controller by setting its registers so it knows what to transfer where.*
    2. *The DMA controller initiates the transfer by issuing a read request to the disk controller.*
    3. *The disk controller fetches the next word from its internal buffer and writes it to the memory.*
    4. *When the write is finished, the disk controller sends an acknowledgement to the DMA controller*
    5. *If there are more data to transfer (counter is greater than 0), then the DMA controller repeats steps 2 through 4. If all the data has been transferred, the DMA controller interrupts the CPU to let it know that the transfer is complete*

7. Explain what is symbolic link and list at least two of its drawbacks.

    Symbolic link is a way to achieve file sharing. When we want to share a file, we create a new file, or a link. This file contains the path name of the file that we want to share. Later when we access this link file, the operating system sees that the file being accessed is a link, so it can lookup the shared file and all accesses go to that file. Symbolic link is very useful in file sharing. But it also has some drawbacks. First, it needs some disk space for its i-node and data. Second, when we open a symbolic link, we have to first read the path name in that link, and then follow the link to the shared file. This needs some extra processing time. These two drawbacks are not present in a hard link. Both symbolic link and hard link have another problem, since a file may now have more than one path name, the backup of a file system must be done more carefully.

CSE 120 Principles of Computer Operating Systems

Fall Quarter, 2000

Final Exam Solutions

Instructor: Geoffrey M. Voelker

Name _____

Student ID _____

Attention: This exam has nine questions worth a total of 110 points, and the last one is a freebie. You have three hours to complete the questions. As with any exam, you should read through the questions first and start with those that you are most comfortable with. If you believe that you cannot answer a question without making some assumptions, state those assumptions in your answer.

Best of luck, and have a great holiday!

1	/12
2	/14
3	/12
4	/12
5	/8
6	/14
7	/10
8	/18
9	10/10
Total	/110

1. (12 pts) Potpourri: Answer yes or no, or with a single term or phrase, as appropriate. You should go through these quickly, answering with the first answer that comes to mind — it probably is the correct one. Only dwell on ones you are unsure of after finishing the other questions.

   (a) Which bit in the PTE does the operating system use for approximating LRU replacement?
   Reference/Use bit

   (b) The layout of disk blocks for a file using Unix inodes is always/sometimes/never contiguous on disk?
   Sometimes

   (c) Does a TLB miss always/sometimes/never result in a pagein from disk?
   Sometimes

   (d) In RPC, what does the server-stub call?
   The server procedure

   (e) What are the two components of a virtual address used in segmented virtual memory?
   Segment # and offset

   (f) In general, are there more/same/fewer inodes than directories in a file system such as Unix?
   More

   (g) In what Nachos file was StartProcess implemented?
   progtest.cc

   (h) What kind of pages in a process' virtual address space are usually protected as "Read Only"?
   Code pages; copy-on-write pages was also a good answer

   (i) The layout of virtual pages for a virtual address space is always/sometimes/never contiguous in physical memory?
   Sometimes

   (j) In what metric do RAID arrays improve performance?
   Throughput

   (k) Is it possible to implement a file system inside of a file stored in another file system?
   Yes; just like we can virtualize an OS and user programs in a single process, we can virtualize a file system in a single file

   (l) Was the exam was too easy/just right/too hard (this is a free point)?

2. (14 pts) Identify the following in one phrase or sentence (do more than just expand the acronym, though). Then state whether each item is related to an operating system *mechanism* or *policy*.

   (a) PTE – *Page Table Entry, used to map virtual to physical addresses, stores access and protection bits (mechanism)*

   (b) SCAN – *AKA Elevator, a disk scheduling algorithm that moves the disk head back and forth across the disk surface, servicing requests (policy)*

   (c) Working set – *In a given time interval, the set of pages required to prevent page faults; used in the Working Set model for page replacement (policy)*

   (d) LRU – *Least Recently Used, a virtual page replacement algorithm that evicts pages that were last used furthest in the past (policy)*

   (e) Thrashing – *A condition where the system spends most of the time transferring pages to and from the disk and little time making progress on application computations; usually due to memory being over-subscribed or a poor page replacement algorithm (relates to policy)*

   (f) Inode – *A Unix file descriptor for the layout of file blocks on the disk; it is index-based and hierarchical (mechanism)*

   (g) Access control list – *a protection mechanim that describes, for a given object, the subjects and the actions allowed by each subject on the object (mechanism)*

   Most people got most of these correct. The mistake that cost people the most points was forgetting to state whether each term relates to a policy or mechanism.

3. (12 pts) Operating systems frequently exploit locality to improve performance. *Briefly* describe two examples where operating systems do so, and state how locality is exploited.

1. Virtual memory and associated page replacement algorithms exploit the temporal and spatial locality of a program's accesses to its data. Programs tend to access pages that were accessed recently, enabling physical memory to be used as a cache for all of the data used by a program during its execution. Programs also tend to access data across a given page, which helps amortize the cost of bringing pages in from disk. Without such locality, virtual memory would be prohibitively slow.

2. The file buffer cache and associate replacement algorithms exploit the temporal and spatial locality of a program's accesses to file blocks. Programs tend to access recently-accessed file blocks, enabling the file buffer cache to serve a significant fraction of requests from memory instead of disk even though the file system is a couple of orders of magnitude larger than the file buffer cache. Programs also tend to access file blocks logically near previously-accessed file blocks (e.g., sequentially), enabling the system to use techniques like read-ahead to anticipate program behavior.

And there are others (e.g., TLB)...

I was looking for two separate examples, and at least a temporal or spatial motivation for how locality is exploited in each example.

4. (12 pts) One day famed student Joe Surfer had an inspiration while hanging ten at Mission Beach. He observes that most programs have most of their data at the beginning of the address space. For his homegrown SaltWater OS, he decides that he is going to implement his page tables similar to the way Unix implements inodes. He calls this page table design *Inode Page Tables*. Inode Page Tables are essentially two-level page tables with the following twist: The first half of the page table entries in the master page table directly map physical pages, and the second half of the entries map to secondary page tables as normal. Call the first half the entries *fast*, and the second half *normal*.

   For the following questions, assume that addresses are 32 bits, the page size is 4 KB, and that the master and secondary page tables fit into a single page.

   (a) How many virtual pages are *fast* pages?
   $4KB/4 = 1024$ *PTEs*
   $1024/2 = 512$ *PTEs* $=>$ 512 *or* (2^9) *pages are fast*

   (b) How many virtual pages are *normal* pages?
   The remaining 512 PTEs refer to second-level page tables. Each second-level page table has 1024 PTEs, so:
   $2^9 * 2^{10} = 2^{19}$ *pages are normal*

   (c) What is the maximum size of an address space in bytes (use exponential notation for convenience, e.g., 2^3)?
   The size of the address space is the total number of pages times the size of each page:
   $(2^9 + 2^{19}) * 2^{12} = 2^{21} + 2^{31}$ *bytes*

   (d) Inode Page Tables reduce the lookup time for fast pages by one memory read operation. Do you think that this is an effective optimization? Briefly explain.
   Not really. This optimization saves one memory access whenever there is a miss in the TLB. Since TLB misses are relatively infrequent, saving one memory access in the miss handler is not going to improve performance significantly. It also makes the PTE lookup a bit awkward since you have to check to see what kind of a page needs to be loaded into the TLB.
   A number of people answered that this shrank the address space too much to be useful. The address space is still > 2 GB, which is pretty large. Most programs will still easily fit inside that address space.

5. (8 pts) In lecture we said that, if the semaphore operations *Wait* and *Signal* are not executed atomically, then mutual exclusion may be violated. Assume that *Wait* and *Signal* are implemented as below:

```
void Wait (Semaphore S) {
  while (S.count <= 0) {}
  S.count = S.count - 1;
}

void Signal (Semaphore S) {
  S.count = S.count + 1;
}
```

Describe a scenario of context switches where two threads, T1 and T2, can both enter a critical section guarded by a single mutex semaphore as a result of a lack of atomicity.

Assume that the semaphore is initialized with count $= 1$. *T1 calls Wait, executes the while loop, and breaks out because count is positive. Then a context switch occurs to T2 before T1 can decrement count. T2 also calls Wait, executes the while loop, decrements count, and returns and enters the critical section. Another context switch occurs, T1 decrements count, and also enters the critical section. Mutual exclusion is therefore violated as a result of a lack of atomicity.*

6. (14 pts) Unix provides two mechanisms to link one file to another, *hard links* and *soft links*. With hard links, assume that the directory entry for the link maps the link file name to the *inode* for the file to which it is linked. With soft links, assume that the directory entry for the link maps the link file name to the *file name* of the file to which it is linked.

   Consider the situation where we want to create a link "/bin/ls" to the existing file "/sbin/ls". In this case, "/bin/ls" is the link file name and "/sbin/ls" is the file to which it is linked. For the questions below, assume that, each time Unix has to retrieve information from the disk, it takes only one disk read operation.

   I was looking for two things from this problem. One was an understanding of how path searching relates to the use of the on-disk data structures supporting file systems (directories and inodes). The second was understanding the difference between having a directory point to a name, which needs to be resolved again, and having it point to an inode, which can be referenced directly.

   The most common error on this problem was to forget that inodes need to be read from disk, and that both files and directories have inodes that need to be read at every step.

   I was lenient with respect to the master block.

   (a) First, succinctly describe what steps Unix will take, in terms of the disk data structures it must read, in order to resolve the path name "/sbin/ls" so that it can read the first byte of the file.

   Master block $->$ inode for "/" $->$ first block of "/" $->$ inode for "sbin" $->$ first block of "sbin" $->$ inode for "ls" $->$ first block of "ls".

   (b) How many disk reads will be required to resolve the path name and read the first byte of the file "/sbin/ls"?

   7

   (c) Consider when we use a hard link to link "/bin/ls" to "/sbin/ls". Succinctly describe the steps required to read the first byte of "/sbin/ls" starting with the link "/bin/ls". How many disk reads will it require?

   Requires the same steps and same number of reads as "/bin/ls", with "sbin" replaced by "bin". Regardless of the steps and read count, I also specifically looked to see if the read count was the same as "/sbin/ls".

   (d) Consider when we use a soft link to link "/bin/ls" to "/sbin/ls". Succinctly describe the steps required to read the first byte of "/sbin/ls" starting with the link "/bin/ls". How many disk reads will it require?

   First: Master block $->$ inode for "/" $->$ first block of "/" $->$ inode for "bin" $->$ first block of "bin".

   This gets us to the link name "/sbin/ls". We then follow the steps in part (a) to resolve the link and access the first block of ls.

   $7 + 5 = 12$ *reads.*

   I specifically looked to see if the read count was more (roughly twice as much) as the counts in (a) and (c). I gave full credit for an answer that said it required twice the number of reads as part (a).

   (e) Unix maintains a reference count of the number of hard links to a file, and it only removes a file's blocks on disk when this count reaches zero. If we remove the file "/sbin/ls", is the hard link still valid? Is the soft link still valid?

The hard link is still valid (the link count is non-zero, so the blocks are not removed and the pointer to the inode in the hard link is still valid), but the soft link is not (when it tries to resolve the name "/sbin/ls" the resolution will fail).

7. (10 pts) Briefly describe the most challenging bug that your group encountered in project 3, how you found it, and how long it took you to find it. Which of your group members do you think will give the best answer to this question?

   This was a gimme. If you wrote something, but didn't actually describe a bug, I gave you 8 points for not following directions.

8. (18 pts) In this problem you will outline an implementation of shared memory in Nachos. Shared memory will be implemented using two new system calls:

```
RegionID SharedRegionCreate (int beginAddress, int endAddress);
    void SharedRegionAttach (RegionID region);
```

SharedRegionCreate creates a shared memory region in the caller's address space defined by the begin and end addresses. If successful, it returns a RegionID identifying the shared region. SharedRegion-Attach maps the existing shared region identified by the RegionID into the caller's address space. A region can only be created by one process, but it can be attached by an arbitrary number of processes. You can assume that each PTE has an additional flag *Shared*, which should be TRUE if that virtual page is being shared. You can also assume that a shared memory region can only be created within a defined region of a process' virtual address space.

These operations can be used as follows. A parent process uses SharedRegionCreate to create a shared memory region, and then Execs a child process and passes the returned RegionID for the shared region to the child as an argument to Exec. The child then uses SharedRegionAttach to map that region into its address space so that it can share memory with its parent. As a result, whatever data the parent places and modifies in the shared region, the child can access it (and vice versa).

For each of the following questions, answer them descriptively at a high level. The answers should be brief, and capture the essence of what needs to be implemented at each stage. You do not need to use psuedo-code.

(a) What should SharedRegionCreate do to the caller's page table?

It should mark all pages in the region as Shared = TRUE. When implementing this, you will also want to allocate a RegionID, update a RegionID table, etc., but this question only asked about what happens with the page table.

(b) What should SharedRegionAttach do to the caller's page table?

It should (1) mark all pages in the region as Shared = TRUE, and (2) set the PTE fields in the caller's page table to be equivalent to those in the page table of the process that created the region. The main thing I looked for was setting the virtual to physical page mapping to point to the shared physical page. In effect, the caller's page table should copy the PTEs from the creator's page table.

(c) What *additional* page table operations must be performed when a shared page is paged out (evicted) from physical memory?

The problem introduced with sharing pages is that multiple page tables have virtual to physical mappings to the same physical page. So if a shared page is paged out (evicted), all of those page tables have to be updated to reflect the eviction. In particular, the valid bits for this page in all page tables sharing the page have to be set to FALSE because an access by any process to the shared page needs to produce a page fault. At least one of the page tables has to be updated with the location of the page on disk, but all of them can be, too.

Delaying eviction until the page is not shared does not answer the question.

Setting Shared = FALSE is very problematic because then you don't know which pages in which page tables need to be updated when the page is paged in.

(d) What *additional* page table operations must be performed when a shared page is paged in from backing store?

The problem here is the complement of the previous question. All PTEs referencing the shared page need to be updated to reflect the fact that the page is now in memory. In particular, the virtual to physical mapping needs to be updated, and the valid bit has to be set to TRUE; if you ignored other bits, like the dirty bit, I let it slide. In effect, the PTEs in all page tables sharing the page have to be set to the same values.

(e) How should the destructor for AddrSpace be changed to account for shared pages?

I accepted two answers. The first is to have the destructor only free the physical page if the AddrSpace is the last one referencing a shared page (preferred). The second is to have the destructor invalidate the PTEs in all address spaces that attached to the shared region (plausible, but harsh).

(f) Name three error conditions that an implementation of SharedRegionCreate and SharedRegionAttach will have to check for.

1. SharedRegionAttach – region does not exist
2. SharedRegionAttach – unknown RegionID
3. SharedRegionCreate – invalid VA range
4. SharedRegionCreate – region already exists
5. SharedRegionAttach – region already created or attached
6. ...

There was a plethora of error conditions to choose from. Some people answered that checking that the beginAddress was valid was one error check, and checking that the endAddress was a second. You have to be more creative than this – these are essentially the same check.

9. (10 pts) You get full credit for the problems below whether you answer them or not, or what your answer is. In other words, these questions will have no impact on your grade. **Do not** bother to write answers to these questions until you are finished with the rest of the exam, if you choose to do so at all.

   (a) What topic in this course struck you as the least interesting, least relevant, and made the least impact on your education?

   (b) What topic in this course was the most interesting to you?

   (c) What operating system topic were you hoping to learn about, but we didn't cover?

   (d) What message would you give to incoming students of the next CSE 120 class I teach? I will use some of these answers in the intro lecture next time.

Name: _____

Operating Systems
CSCI-UA.0202 Fall 2013

Final Exam
ANSWERS

Please write your name on this sheet. Answer question 1 on this sheet and put all other answers in the blue book.

1. **True/False**. Circle the appropriate choice on this sheet.

   (a) **T F** DMA is a mechanism for allowing an I/O device to transfer data to and from memory without involving the CPU in the transfer.
   Answer: T

   (b) **T F** Memory mapped I/O determines how the pages of an I/O-bound process are mapped to page frames.
   Answer: F

   (c) **T F** The root directory of a partition in a Unix system is named "/".
   Answer: T

   (d) **T F** There is only one MBR (master boot record) on a disk drive, but there could be several boot sectors.
   Answer: T

   (e) **T F** A context switch from one process to another can be accomplished without executing OS code in kernel mode.
   Answer: F

   (f) **T F** An advantage of implementing threads in user space is that they don't incur the overhead of having the OS schedule their execution.
   Answer: T

   (g) **T F** Deadlock can never occur if no process is allowed to hold a resource while requesting another resource.
   Answer: T

   (h) **T F** In round robin scheduling, it is advantageous to give each I/O bound process a longer quantum than each CPU-bound process (since this has the effect of giving the I/O bound process a higher priority).
   Answer: F

   (i) **T F** For machines with 32-bit addresses (i.e. a 4GB address space), since 4GB physical memories are common and cheap, virtual memory is really no longer needed.
   Answer: F

   (j) **T F** A TLB miss could occur even though the requested page was in memory.
   Answer: T

2. Virtual Memory

   (a) Suppose you had a computer that supported virtual memory and had 32-bit virtual addresses and 4KB (2^{12} byte) pages. If a process actually uses 1024 (2^{10}) pages of its

virtual address space, how much space would be occupied by the page table for that process if a single-level page table was used? Assume each page table entry occupies 4 bytes.

There are $2^{32}/2^{12} = 2^{20}$ pages in the address space. Therefore, a single level page table would have 2^{20} entries, where each entry is 4 bytes. Therefore, the page table would require $4 * 2^{20}$ bytes = 4 MB. This is independent of how many pages are actually being used.

(b) Suppose, in the same machine as above, a two-level page table was used, such that the first level page table was four times the size of each second level page table. Do you have enough information to know exactly how much space is occupied by the two-level page table for that process? Explain.

No. How the used pages are distributed across the address space of the process will determine how many second-level page tables are needed.

(c) If your answer to part (c) was "yes", compute the space occupied by the two-level page table for that process. Otherwise, compute the minimum amount of space that could possibly be required for the two-level page table for that process and compute the maximum amount of space that could possibly be occupied by the two-level page table for that process. Be sure to show your work.

First, we need to compute the sizes of the first and second level page tables. Since there are 2^{20} pages and the first-level page table is four times the size of each second level page table, the first level page table must have 2^{11} entries and each second level page table must have 2^9 entries (since their product must equal 2^{20}). Further, since each page table entry occupies 4 bytes (2^2 bytes), the first level page table occupies $2^{11} * 2^2 = 2^{13}$ bytes (8KB) and each second level table occupies $2^9 * 2^2 = 2^{11}$ bytes (2KB).

Since 2^{10} second-level page table entries are needed, in total, to support the 2^{10} pages in use, there needs to be, at a minimum $2^{10}/2^9 = 2$ second-level page tables. These two second-level page tables, along with the first level page table, would occupy a total of $(2 * 2^{11}) + 2^{13}$ bytes = 4KB + 8KB = 12KB.

At worst, each of the 2^{10} pages used by the process could be mapped by a different second-level page table (e.g. if the pages in use were spaced equally across the entire address space). In that case, 2^{10} second-level page tables would be required in addition to the first level page table, so the total space occupied by the page tables would be $(2^{10} * 2^{11}) + 2^{13}$ bytes = 2MB + 8KB.

(d) In an virtual memory system that supports the NRU (Not Recently Used) page replacement algorithm, how are the R and M bits used to determine which page to evict from RAM, if necessary? What hardware support is needed to maintain the R and M bits?

For a page in memory, the R bit for a page is set upon every access to that page. The M bit for a page is set when that page is written to. The R bits for all the pages in memory are occasionally cleared. When memory is full and a new page needs to be brought into memory, the NRU algorithm chooses a page to evict from memory in the following order of priority: 1) R=0, M=0; 2) R=0, M=1; 3) R=1; M=0; 4) R=1; M=1. The hardware must support the setting of the R and M bits of each page(since you don't want to incur the cost of having the OS do it). Depending on how often the R bits are cleared, you may want to have the hardware do this as well.

(e) Suppose you had a (very) simple virtual memory system with a physical memory that consists of only 4 page frames. Suppose also that R bits are supported and are cleared after every 6 memory accesses.

Assuming that memory is initially unoccupied, give a sequence of page accesses that, if generated by the processor, would cause a page to be evicted from memory – but would cause a *different* page to be chosen for eviction by each of following page replacement algorithms:

- NRU
- Second Chance
- LRU (least recently used, assume it is fully supported by the hardware).

Your answer should be of the form "0,2,9,1,0...", in this case indicating that page 0 was accessed first, page 2 was accessed next, etc. Your sequence should be fairly short. You should also indicate which page would be chose for eviction by each of the above page replacement algorithms.

One possible answer is the following sequence of page accesses: 3, 0, 1, 2, 3, 0, 1, 4.

Which page to evict when page 4 needs to be brought into memory depends on the page replacement algorithm as follows:

- **NRU: The request to page 4 would require the eviction of one of the four pages (0,1,2,3) already in memory. Since the R bits are reset after 6 memory accesses, only the R bit for page 1 would be set. Therefore, the OS would either choose one of the other pages randomly or might choose the lowest numbered page to evict (0).**
- **Second chance: Second chance would pick the page to evict that was first loaded into memory (like FIFO), unless the R-bit for that page was set. Since the R bits are cleared after 6 accesses, and only page 1's R bit would be set, page 3 would be chosen to evict.**
- **LRU: Since page 2 is the least recently accessed, LRU would choose page 2 to evict.**

3. Files

(a) Suppose a computer has a file system for a 128GB (2^{37} byte) disk, where each disk block is 8KB (2^{13} bytes). If the OS for this computer uses a FAT, what is the smallest amount of memory that could possibly be used for the FAT (assuming the entire FAT is in memory)? Explain.

There has to be a FAT entry for each disk block. Since the disk is 2^{37} bytes and a disk block is 2^{13} bytes, the number of disk blocks (and thus the number of FAT entries) is $2^{37}/2^{13} = 2^{24}$. Since there are 2^{24} entries, a block number (disk address) requires a minimum of $\log(2^{24})$ bits = 24 bits = 3 bytes. In this case, the minimum amount of space occupied by the FAT is the number of entries (2^{24}) times the 3 bytes per entry, namely $2^{24} * 3$ = 16MB * 3 = 48MB.

(b) Suppose that on a different computer, the OS uses i-nodes and each disk block is 4KB (2^{12} bytes). Assume that an i-node contains 12 direct block numbers (disk addresses) and the block numbers for one indirect block, one double indirect block, and one triple

indirect block. Assume also that a block number is 4 bytes. What is the largest possible file on that computer (assuming the disk is large enough).

Since a disk block is 4KB (2^{12} bytes) and a block number is 4 (2^2) bytes, there are 2^{10} = 1024 entries per indirect block. Therefore, the maxium number of blocks of a file that could be referenced by the i-node is $12 + 2^{10} + (2^{10})^2 + (2^{10})^3 = 12 + 2^{10} + 2^{20} + 2^{30}$**. Thus, the maxium size of the file would be** $(12 + 2^{10} + 2^{20} + 2^{30}) * 2^{12} = (12 * 2^{12}) + 2^{22} + 2^{32} + 2^{42}$ **= 48KB + 4MB + 4GB + 4TB.**

(c) On the computer from part (b) above, suppose you wanted to create a new file of the maximum size (that you computed for the answer to (b)). How many free blocks on the disk would there need to be in order to create that file?

As discussed above, you would need $12 + 2^{10} + 2^{20} + 2^{30}$ **blocks just to contain the data in the file itself. However, you would also need free blocks to use for your single indirect, double indirect, and triple indirect blocks. The total of those blocks is** $1+(1+2^{10})+(1+2^{10}+2^{20}) = 3+(2*2^{10})+2^{20}$**. Adding together the file blocks and the indirect blocks gives** $(12+2^{10}+2^{20}+2^{30})+(3+(2*2^{10})+2^{20}) = 15 + (3 * 2^{10}) + (2 * 2^{20}) + 2^{30}$**.**

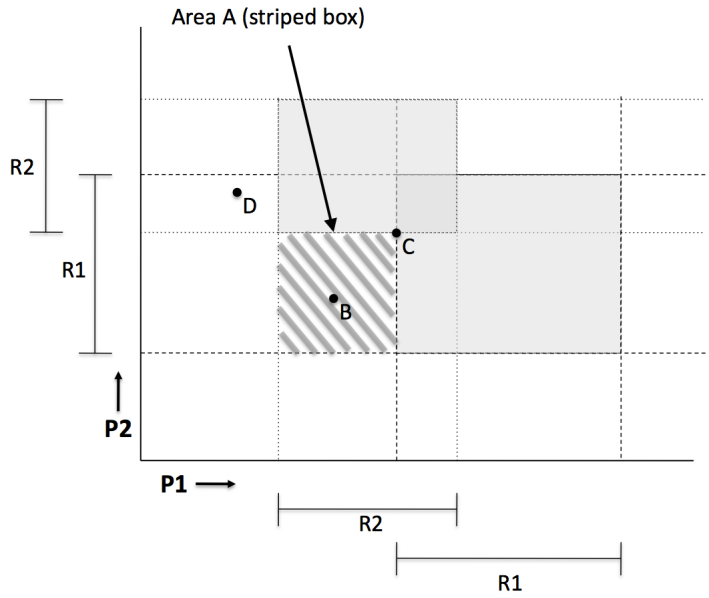
(d) As discussed in class, a hard link is an entry in a directory that refers to the same file as another directory entry (in the same directory or not). If you were to try to implement hard links in an OS that uses a FAT, what would be contained in the directory entry for a hard link?

Using a FAT, each entry in a directory contains the first block number for that file. This is because the blocks of a file is determined by its first block, i.e. the entry point into the FAT. Thus, a hard link might be implemented in a FAT system by having a directory entry contain the same first block number as another directory entry.

(e) If you didn't change the structure of a FAT (as found, for example, in DOS), what would be the problem with your hard link implementation?

The problem is that a FAT entry, unlike an i-node, doesn't keep track of how many directory entries refer to that entry. Thus, if a user deletes a file – even if another hard link to that file exists – the block of that file will be put back on the list of free blocks (and eventually get overwritten).

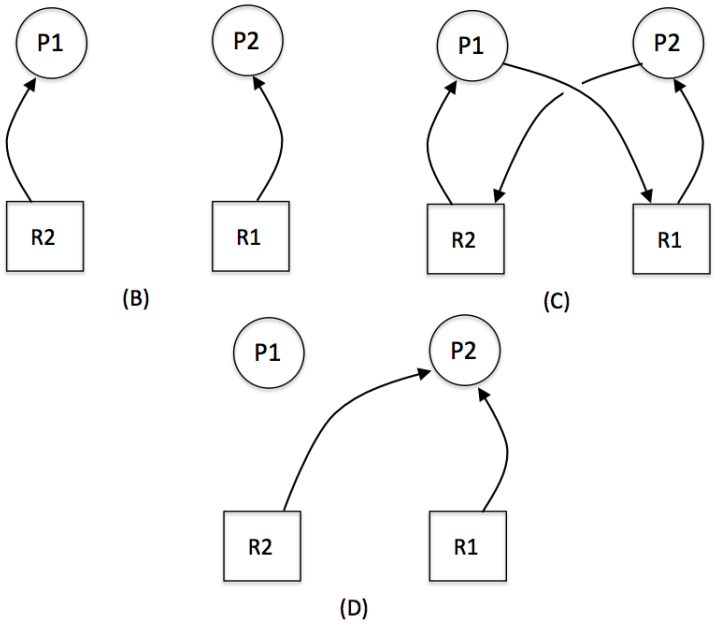
4. Deadlock Consider the following resource trajectory graph for a system with two processes (P1 and P2) and two resources (R1 and R2).



(a) What does the area identified as Area A indicate?

A region where the processes are not deadlocked, but will inevitably be deadlocked (when they hit point C).

(b) Draw three resource allocation diagrams (the ones with circles, squares and arrows), one for each of the points indicated above as B, C, and D.



(c) If you drew the same resource allocation graphs for both B and C, explain why. If you

drew different resource allocation graphs for points B and C, explain why. What does that tell you about the usefulness of using resource allocation graphs to avoid deadlock?

They're different because B is not actually a deadlocked state (no cycle), whereas C is a deadlocked state. The problem with using resource allocation graphs to avoid deadlock is that, as for point B, they can't be used to detect when deadlock is possible, or even inevitable. They can only distinguish between deadlock and non-deadlock.

(d) In what way is the banker's algorithm "pessimistic"?

It makes a worst case assumption about how processes will request and release resources. It still ensures that, even in the worst case, there is still a path that allows all the processes to eventually have their resource requests satisfied and thus terminate.

(e) Write, in C, three very simple, non-looping programs that use shared binary semaphores, such that deadlock among all three could occur if nothing was done to prevent it. Assume the programs don't use any other resource that could cause deadlock. Give a scenario of execution in which your three programs actually deadlock.

Here is just one of many possible answers. Assuming there are three binary semaphores, `S1`, `S2`, and `S3`, the programs are,

```
void prog1()
{
  down(S1);
  down(S2);
  down(S3);
  critical_section();
  up(S3);
  up(S2);
  up(S1);
}
```

```
void prog2()
{
  down(S2);
  down(S3);
  down(S1);
  critical_section();
  up(S1);
  up(S3);
  up(S2);
}
```

```
void prog3()
{
  down(S3);
  down(S1);
  down(S2);
  critical_section();
  up(S2);
  up(S1);
  up(S3);
}
```

A deadlock scenario is when `prog1` executes `down(S1)`, followed by a context switch which allows `prog2` to execute `down(S2)`, followed by a context switch that allows `prog3` to execute `down(S3)`. After that, each program will block when executing its next statement (another `down(...)` operation) – which leads to deadlock.

(f) Assume that each program has to declare, when it starts, which semaphores it will be using. In what way would the bankers algorithm, assuming the OS used it, prevent the deadlock in the scenario you gave in your previous answer?

Since each of the above programs will have declared that it will be using all three semaphores, once `prog1` has succesfully executed `down(S1)`, any successful `down` operation by the other programs would result in an unsafe state, therefore the OS would cause a `down` operation by `prog2` or `prog3` to block. In fact, `prog2` and `prog3` would block until `prog1` has finished, at which time whichever program is unblocked first would run to completion before the last program could unblock.

5. I/O

(a) Describe the elevator disk scheduling algorithm.

Given a set of requests for disk blocks, the controller services those requests by having the disk head move in a single direction, satisfying the requests for blocks on cylinders encountered along the way (i.e choosing the next block requiring the shortest seek in the current direction from the current head position). When the block request on the last requested cylinder in the current direction has been satisfied, the controller reverses the current direction of the head.

(b) What is the advantage of the elevator algorithm over the shortest-seek-first algorithm?

In the presence of a stream of incoming block requests, the shortest seek first algorithm, which always chooses the block request to satisfy that requires the least movement of the head, could lead to starvation of requests for blocks that are far from the current head position.

(c) When a user uses the mouse to "drag" an object across the screen, what is the sequence of events that the mouse controller will generate to communicate with the CPU? What data is communicated with each event?

The mouse controller will generate an event (e.g. an interrupt) indicating a mouse-button-pressed event, specifying which mouse button was pressed. Then it will generate a series of events indicating that the mouse has moved, specifying each movement by Δx and Δy displacement values. Finally, it will generate a mouse-button-released event, specifying the mouse button that was released.

6. Extra Credit: Describe how a modern optical mouse controller determines the direction and distance that the mouse has been moved (I discussed this in detail in class).

The controler will capture a rapid sequence of images of the surface under the mouse. The surface is lit by an LED shining light downwards at an angle, so that irregularities in the surface cause shadows. Each successive image is compared to the previous image, by overlaying images at successive relative offsets (in both the X and Y directions). The offset that minimizes the differences (e.g. by performing a least-squares computation) between the successive images indicates the direction and magnitude of the movement, which will be reported by the mouse controller.

CS140
Operating Systems and Systems Programming

Final Exam. Summer 2006.
By Adam L Beberg.
Given August 19th, 2006.

Total time = 3 hours, Total Points = 335

Name: (please print)_____

In recognition of and in the spirit of the Stanford University Honor Code, I certify that I will neither give nor receive unpermitted aid on this exam.

Signature:_____

- This exam is closed notes and closed book.
- No collaboration of any kind is permitted.
- You have 3 hours to complete the exam.
- There are 24 questions totaling 335 points. Some questions have multiple parts. Read all parts before answering!
- Please check that you have all 13 pages.
- Before starting, write your initials on each page, in case they become separated during grading.
- Please print or write legibly.
- Answers may not require all the space provided. **Complete but concise answers are encouraged.**
- SCPD students: If you wish to have the exam returned to you at your company, please check the box.

Page 2	/10
Page 3	/30
Page 4	/30
Page 5	/25
Page 6	/25
Page 7	/35
Page 8	/40
Page 9	/35
Page 10	/40
Page 11	/30
Page 12	/35
TOTAL	/335

SCPD?	

INSTRUCTIONS

Near the beginning of the course, if not the first lecture, I said that even the "Hello World" application involved millions of lines of code to make run. Now that the course is over, and with your experience with Pintos, you should understand what all of those lines of code are doing.

This is your chance to demonstrate that you understand operating systems. If you watched the lectures, did the assignment, and read the materials, all of these questions should be easy. This exam is 1/3 of your grade, so show us you know the material. You should have much more time then you need to do a great job.

All questions will refer to the source code on the LAST page of this exam as "the code". Feel free to rip that page off and refer to it, and use it as scratch paper, you do not need to turn that page in (we have a copy).

READ ALL THE QUESTIONS FIRST! Unlike most tests, this is not just a suggestion. In general you're asked what happens when things go smoothly, then about when things go a different way, each questions follows from the one before. If you're not about to be asked about an edge case, or an important issue, go ahead and mention it however.

BEGIN CS 140 FINAL EXAM

1. [10] You take the code and run `` `gcc -c final.c -o final.o` `` to compile the code. What stuff is in the final.o file? Exact names/number are not needed, you did that in the midterm.

   Main things you should mention:

   The compiled code (text segment), but with none of the addresses filled in.
   The data segment, with initialized data.

   Table of data definitions for all declared variables - defs.
   Table of code references to external and internal functions- refs.

2. [10] (a) Next you link final.o along with the standard C library. What does the linker spit out if the C library is a static library - libc.a?

   The linker puts together all the text (code) segments into one big text segment. Same for data, etc.

   Takes all the defs and refs and resolves them, inserts real addresses into all those locations.

   Puts a nice header on the resulting segments, with the start address and lots of descriptive info, and spits it out.

   [5] (b) What unused code will end up in the program, and why?

   Extra unused functions end up in the end binary because the linker isn't smart enough to only link in the functions it needs, but also puts any functions in the same .c files as functions it needs in.

3. [10] Same as question #2, but with a dynamic stared library - libc.so, how is this program different?

   Instead of adding in the code, and resolving all the addresses, it instead puts in jumps/pointers to addresses in a table that are filled in at run time. Glue logic is also added to load the library and it's data and fill in the table.

   If linking is delayed all the way to call time, then the table is filled in the first time that function is called, not when the program is loaded.

4. [5] List 3 things in the ELF header(s) of the program.

   Many things, too many to list here. Search for "Executable and Linkable Format" or do a `` `man elf` ``

   Common answers were the start address, segment start/sizes, CPU/OS types, etc.

5. [20] (a) Now you run `final`. What does the operating system do up to the point where you add it to the ready list? Assume lazy loading like in the project. (completely ignore the details of the file system for now, we'll get to that)

   Some of the main points:

   Load the header, and get the various segments and offsets from the file.

   Allocate a process - PCB, (and/or thread if they are the same) for the priority, owner, etc.

   Add page table entries for the segments of the file we need, and marked them as not-present.

   Allocate and setup a stack, parse arguments, make it look like it's coming back from an interrupt, and returning to the start address.

   Put the thread on the ready list (implied by question).

   [5] (b) And if we weren't so lazy?

   The entire program would be loaded into physical memory before we add the process to the ready list. This would probably involve loading in more pages then we may (ever) need to use, and thus evict more then we have to from other processes.

6. [5] When you run final, all physical memory is already in use. What happens?

   The OS will have to evict another process's pages before it can load ones for the new process. It will do this with some algorithm that best or most easily approximates the impossible MIN(), so an LRU like the clock algorithm etc.

   The key point was to mention something about how you picked the victim page.

7. [5] List 2 situations where the OS will not let the program run for you at that moment, but would let you if the situation was not happening.

   Some possible answers:

   OS/User process limit reached.
   OS/User memory limit reached.
   Machine is thrashing and not letting in any new process.

8. [5] (a) Finally everything is going according to plan, you're ready. When do you get to run?

   The scheduler has to pick your thread to run, taking you from the READY list to the RUNNING state (only one per CPU, not a list).

   [5] (b) Why could this be a while?

   Lots of threads and along timeslice in RR.
   Big job "convoy" effect in FCFS.
   Low priority in multi-queue algorithms with no aging of low priority processes.
   Too long in a STCF system with lots of new short jobs all the time.

   [10] (c) If this was a typical type of program on the machines you administered, what scheduling algorithm would you want to use? Justify your choice.

   Due to the high I/O, then CPU, then more I/O we want an algorithm that is adaptive to changing workloads, so that we can keep CPU and I/O both busy. Multilevel feedback queue is probably the best choice for that reason.

   Some relation to the code was needed in your answer, not just picking multilevel feedback queue.

9. [5] (a) Which lines of code *always* generate a systems call? List #'s.

5, 7, 9 - file operations

11, 12, 13, 14 - networking operations

[5] (b) Which lines of code *might* generate a system call? List #'s.

6 - malloc() may need a new page.

10 - free() may return pages.

10. [10] (a) Assume timeslices end right after every line of code by using the Pyschic Timeslice Algorithm™. If we're using an exponential feedback algorithm, what's happening to our priority as we go through the program. Use lower to mean low priority in your description, not a low queue number, and higher to mean higher priority.

1	1-4 are not run	8	lower
2	...	9	higher
3	...	10	higher
4	...	11	higher
5	higher	12	higher
6	higher or lower OK	13	higher
7	higher	14	higher

I/O and fast things makes it higher, long computation makes it lower.

[5] (b) What if we're billing all time spent in the system call time to the process, would this change your answers? Which ones?

5 - open() can get ugly with enough permissions etc.

6,10 - malloc/free can be long if much of memory is full, bookkeeping.

Arguments can be made for 7 & 9 taking a long time with enough fragmentation.

11-14 are still just commanding devices and moving on.

You should have had at least 1 or 2 of these reasons.

11. [20] (a) Now back to that file system were ignoring. If we're running on the Elaine machines and using an ACL based access control system, what happens when the open() in line 5 is run? Describe all the structures and OS ideas involved. (now you can ignore the syscall details)

First, the filename is parsed out into each directory and the filename.

Starting at the root directory, which is at a known location, the system loads up the inode of the directory, and then the directory and it's ACL, then checks them against the user.

It then checks for the next directory, "foo" in this case, and repeats.

When it gets to the file part of the path, we look for it in the last directory, and if so, we locate it's inode.

Once all those permissions have passed, we can store this inode, give it a per-process number, and use that number to service further syscalls on the file.

[10] (b) What if we were in a Capabilities based system? Also, What happened at login to make this possible?

Instead of the permissions checking against the list of allowed users on the directory/file, the system checks the object against the users list.

At login, your capabilities were assigned to you, and your login process.

[5] (c) What if the file isn't openable, what's different between how ACL and Capabilities handle it?

When you don't have access to something in a capabilities system, it's completely hidden from you.

12. [10] Now we're in a typical UNIX file system with inodes. What's going to be happening as far as figuring out where the blocks are as we read (many) blocks due to line 7.

    The inodes must be looked at the get the physical block numbers that the files data resides in.

    This will first involve the direct blocks, which are part of the inode.

    For longer files, this then leads to indirect blocks, which means you have to read another disk block full of pointers to blocks.

    Double-indirect blocks come into play after one block's worth of pointers, and then you traverse 2 layers to get to the block pointers.

    [5] (b) … and with an extent based file system?

    With extent, you have a series of (start, length) pairs, and blocks are laid out sequentially. This makes the messy inode structure/traversal unnecessary.

13. [10] How would this be different if before line 7 there was a call to mmap().

    Instead of blocks going into the buffer cache, the VM system is told to map the file's blocks into virtual memory, and from there on, the VM system pages in/out blocks as if it were a special swap file.

    This is generally much more efficient.

14. [10] (a) By now we know what disk blocks we need and have a long list, and so do other processes (maybe they have short lists). What do we tell the disk to do so everyone is happy?

    You need to schedule the disk accesses in a way that is fair and fast. We talked about the elevator algorithm that scans from the first to the last cylinder and reads/writes blocks as it goes. This way noone starves, but the disk is also busy.

    [5] (b) What does SCSI do that IDE doesn't that's makes this a non-issue.

    SCSI reorders the read/write requests for you, so you don't have to.

15. [10] After we start this program, the system comes to a standstill, and becomes unbearably slow. In what ways could `BIG_NUMBER` be involved in this.

Large amounts of memory are used, so we could trigger thrashing.

Large amounts of disk I/O may grow the buffer cache and force memory to swap.

[5] (b) Is this really avoidable at all if `BIG_NUMBER` is really big? How? Your answer should not involve violence ;)

There were many possible answers for this, but the short answer is no, but you can make it less bad.

You can try by buying memory, impose memory limits, kill processes, schedule using working sets, etc.

16. [10] (a) Is a log based file system likely to help this processes performance or hurt it or not matter? Why?

Since this program does just a large sequential write, and reads are essentially the same in a log file system, the overhead of the double-writes hurts you.

If the log was on another disk you may not notice much.
If you were just logging the metadata you also would not be much slower.

It means you won't crash as bad of course.

[10] (b) What would happen differently in line 7 and in line 9 in the case of a log based file system?

Line 7 is pretty much unaffected. Logging cares only about sequences of writes we're not done with yet.

Line 9, the writing is changed greatly. Everything we write first goes to the log, along with information on what we are doing, then the real file system. This allows us to do caching, and only do the real-write at the end.

17. [5] How do we battle latency and bandwidth limits in disk systems?

Latency is helped by read-ahead and caching.

Bandwidth can be helped by doing good disk scheduling, and with RAID.

18. [10] (a) Alice is sending Bob a message in a public key cryptosystem. What does she do in which order.

Alice signs the data with her private key which only she has. Then encrypts the data with Bob's public key which only Bob can undo. The other order leaves Alice's identity open, and Homeland Security will come to visit.

[5] (b) What does Bob do when he gets the message?

Decrypts the message with his private key, so only he (and the rootkit on his machine) can read it. Then he checks the signature from Alice, which only she could have made... unless Homeland Security has already been there and used the 4 B's.

[5] (c) If Alice and Bob have never talked directly to each other in person or by phone, is this system secure?

No. Alice doesn't know if Bob's public key is really his. And Bob doesn't really know about Alice's key either.

19. [10] (a) What happens during the connect() call in line 12?

The 3 way handshake.

1. We send an (open, seq #x)

2. The other side sends us an (ACK x, seq #y)

3. We reply with an (ACK y)

[5] (b) If it was a UDP socket instead?

Nothing happens except the default address get set.

20. [20] (a) Draw a diagram of what the packet sent in line 13 will look like going across the wire. You don't need to remember all the exact header fields, just the ones we highlighted in class. You _DO_ need every extra chunk added between line 13 and the wire. Label each addition as belonging to the end-to-end, network, or link layer. Here is a start: (since the page is only 8.5 inches wide, some arrows may be useful)

Looking for something along these lines, as long as you got the 3-layer wrapping right, and the main fields, it's probably OK.

Link Layer: Ethernet Header
Sentinel, source ethernet address, dest addr

Network: IP Header
Source IP, dest IP, length, header checksum

E-to-E Layer: TCP Header
Source port, dest port, offset, seq, ack, header+data checksum

data[128 bytes] (from user)

Link Layer: checksum, sentinel

21. [10] What 4 packet problems does TCP fix, and what does it use to solve them all.

Lost Packets, Lost ACKs, reordering of packets, reappearing old packets.

All 4 are solved with ACKs, retry timers and the sequence number, which is a combination of a unique stream ID and the packet number.

Also accepted: does sliding window for flow control.

22. [5] (a) Why do wireless networks normally use encryption in the link-layer, in clear violation of the end to end principle.

Because without wires, anyone can be listening, jamming, or using the connection.

Variations on that theme.

[5] (b) Why _doesn't_ encryption happen in the link later, since we should all obviously be using it for everything now that we're all being watched at all times.

Because encryption is done at the application layer when it is needed, and so this would be redundant.

23. [10] (a) Buffer overflows… what stupid thing do programmers keep doing even though it's 100% obvious when you look at the code.

Putting buffers on the stack. Fixed by using malloc, so the buffer is on the heap.

Also acceptable but not in the code: not checking bounds on all array/string operations.

[5] (b) What did the hardware folks do (or stopped doing for some reason) that made this a problem in the first place?

Data on the stack is mixed in with return addresses.

The stack is executable.

24. [10] Pick 2 lines of code that happen differently (but act the same of course) if we're in a Virtual Machine Monitor. Briefly, what's different?

Any line that did a system call was fair game. The summary is that this will trap into the VMM, and then have to be simulated as an effect on the guest OS, while the VMM actually controls the real hardware/memory/device.

END CS 140 FINAL EXAM

Assume during the "..." that any undeclared variables are declared, any variable that need initializing are initialized, and all function calls succeed. Those details are not important.

Line #	final.c
	`int main( int argc, char * argv[] )`
	`{`
1:	`  int file;`
2:	`  int socket;`
3:	`  char packet[128];`
4:	`  void * matrix = NULL;`
	`  ...`
5:	`  file = open( "/tmp/foo.txt", "r+" );`
	`  ...`
6:	`  matrix = malloc( BIG_NUMBER );`
7:	`  read( file, matrix, BIG_NUMBER );`
8:	`  do_matrix_invert( matrix );`
9:	`  write( file, matrix, BIG_NUMBER );`
10:	`  free( matrix );`
	`  ...`
11:	`  socket = socket( PF_INET, SOCK_STREAM, 0 ); /*TCP*/`
12:	`  connect( socket, &address, sizeof(address) );`
13:	`  send( socket, &packet, sizeof(packet), 0 );`
14:	`  len = recv( socket, &packet, sizeof(packet), 0 );`
	`  ...`
	`}`

CPSC 457
OPERATING SYSTEMS

FINAL EXAM SOLUTION

Department of Computer Science
University of Calgary
Professor: Carey Williamson

December 10, 2008

This is a CLOSED BOOK exam. Textbooks, notes, laptops, calculators, personal digital assistants, cell phones, and Internet access are NOT allowed.

It is a 120-minute exam, with a total of 100 marks. There are 18 questions, and 11 pages (including this cover page). Please read each question carefully, and write your answers legibly in the space provided. You may do the questions in any order you wish, but please USE YOUR TIME WISELY.

When you are finished, please hand in your exam paper and sign out. Good luck!

Student Name: _____

Student ID: _____

Score: _____ / 100 = _____ %

Multiple Choice

Choose the best answer for each of the following 12 questions, for a total of 12 marks.

1 1. Three **file descriptors** associated with every Linux process are:

(a) standard input, standard output, and standard pipe

(b) **standard input, standard output, and standard error**

(c) standard input, standard output, and standard deviation

(d) standard input, standard output, and standard terminal

(e) standard input, standard output, and standard transmission

1 2. User Mode Linux (UML) is an example of a **virtual machine** environment in which:

(a) Linux runs on top of Windows

(b) **Linux runs on top of Linux**

(c) Windows runs on top of Linux

(d) Windows runs on top of Windows

(e) none of the above

1 3. During the **boot process**, a computer obtains its initial bootstrapping information from:

(a) **a special "boot block" on disk**

(b) the superblock in the root file system

(c) a pre-configured file `vmunix` within the file system

(d) the `/tmp` file system

(e) none of the above

1 4. The **copy-on-write** mechanism provides:

(a) an efficient way to create new processes

(b) a clever way to share virtual memory pages (at least temporarily)

(c) a way to avoid unnecessary page copying

(d) **all of the above**

(e) none of the above

1 5. In memory management, **global** page replacement is usually preferable to **local** page replacement because:

(a) most processes are well-behaved

(b) most processes have small working sets

(c) most processes have large working sets

(d) most processes are highly synchronized

(e) **the set of pages from which to choose is larger**

1 6. Implementing **LRU** precisely in an OS is expensive, so practical implementations often use an approximation called:

(a) MRU

(b) MFU

(c) LFU

(d) LFU with aging

(e) **none of the above**

1 7. For two processes accessing a shared variable, **Peterson's algorithm** provides:

(a) mutual exclusion

(b) progress

(c) bounded waiting

(d) **all of the above**

(e) none of the above

1 8. **Counting semaphores**:

(a) generalize the notion of a binary semaphore

(b) are used for managing multiple instances of a resource

(c) have increment and decrement operations

(d) can use queueing to manage waiting processes

(e) **all of the above**

1 9. The **Banker's Algorithm** is an example of a technique for:

(a) deadlock prevention

(b) **deadlock avoidance**

(c) deadlock detection

(d) deadlock recovery

(e) stabilizing turbulent financial markets

1 10. With **asynchronous I/O**, file system changes will be committed to disk when:

(a) the in-memory inode is updated

(b) **the `sync` daemon runs**

(c) the system administrator feels like doing it

(d) nightly file system backups are run

(e) the system is rebooted

1 11. The operation of **defragmenting** a hard disk:

(a) uses compaction to combat internal fragmentation

(b) **uses compaction to combat external fragmentation**

(c) uses compression to combat internal fragmentation

(d) uses compression to combat external fragmentation

(e) all of the above

1 12. Which of the following is an **idempotent** request?

(a) read the next byte from file `foople`

(b) **read block 3 from file `foople`**

(c) write this block to the end of file `foople`

(d) append file `foople` to file `boople`

(e) link file `foople` to file `boople`

OS Concepts and Definitions

15 13. For each of the following pairs of terms, **identify** the context(s) in which they occur. Then **define** each term and **clarify** the key difference(s) between the two terms.

(a) (3 marks) "host OS" and "guest OS"

```
context:  virtual machines
host OS: underlying OS layer, with access to physical hardware
guest OS: runs on top of host OS, provides services to user,
using the resources provided by the host OS
Virtual machines provide flexible execution environments for users.
```

(b) (3 marks) "page" and "frame"

```
context:  (virtual) memory management
frame:  fixed-size basic unit of physical memory allocation
page:  fixed-size logical unit of process address space;
a page fits in a frame; any page can be put in any frame
```

(c) (3 marks) "reference bit" and "dirty bit"

```
context:  paging-based memory management
reference bit:  indicates if a page has been accessed (recently)
dirty bit:  indicates if a page has been modified (relative to disk)
These bits influence decision-making in page replacement policies
```

(d) (3 marks) "file" and "directory"

```
context:  file systems
file:  a named logical collection of related info, as defined by user
directory:  a logical collection of related files, as defined by user
The directory has metadata about files (name, size, location, etc)
```

(e) (3 marks) "disk partition" and "file system volume"

```
context:  file and storage systems
disk partition:  a logical piece of a disk (or set of disks)
file system volume:  a partition that contains a file system
(as opposed to being empty or used as raw disk or swap space)
A partition can hold a file system
```

Processes

16 14. Answer the following questions about processes.

(a) (4 marks) What is a **process**? What is a **thread**? How are they similar/different?

```
process:  a program in execution
thread:  a flow of control within a process
similar:  active entities, with many attributes, that consume system resources
different:  process is heavyweight, thread is lightweight (part of a process)
```

(b) (6 marks) There are many **system processes** active on any Linux system. These are typically created at system startup, and operate in the background as daemon processes. Give **three examples** of system (daemon) processes in a Linux system, and briefly state their role in the operation of the system.

```
Many possible answers here:
init:  system initialization, spawns other system processes on boot
swap/pageout:  do system paging or swapping when needed
sched:  system scheduler
syslogd:  log system-related events
sync:  periodically flush file system modifications to disk
cron:  system timekeeper process (clock, time of day, scheduled jobs)
logind:  handle user login events, verify password, launch shell
sshd/telnetd:  handle remote terminal sessions
nfsd:  handle remote file system requests from clients
ftpd:  handle remote file transfer requests from clients
```

(c) (6 marks) When multiple processes need to cooperate, there is a choice between **shared memory** and **inter-process communication** (IPC). Compare and contrast these two techniques. Make sure to clarify the role of the operating system in each.

```
shared memory:  OS allocates a region of memory that is shared by
more than one process (must be on the same machine to do this!).
Usually done with page tables; processes can then read/write
the shared locations at memory speeds without OS intervention
IPC: message passing; processes communicate using send and receive
these are system calls that invoke OS services; the OS is involved
in every interaction to copy messages to/from address spaces.
IPC generalizes to processes on different machines using sockets
```

Memory Management

15 15. Answer the following questions about OS memory management.

(a) (4 marks) One of the design decisions in OS memory management is the choice between **swapping** and **paging**. Define each of these terms, and clarify their respective roles in OS memory management.

swapping: copies entire process image between memory and disk. Assumes contiguous allocation, entire process needed for execution. Used to limit multiprogramming level and avoid thrashing.
paging: divides logical address space of process into fixed-size pieces; process can execute with only a subset of these pages being resident in memory at a time. Provides flexible and efficient memory management, with lots of processes active at a time.

(b) (5 marks) Another key design decision in OS memory management is the choice between **paging** and **segmentation**. Compare and contrast these two approaches to memory management, making sure to identify the strengths and weaknesses of each.

paging: separates physical organization from logical address space
Uses fixed-size units called pages, which are stored in page frames.
Requires page table to keep track of (possibly many) pages.
segmentation: preserves user's structural view of logical address space
Uses variable-size units called segments. Can go anywhere in memory.
Requires segment table to keep track of the (few) segments.
Need base and limit register for each segment.
(These two techniques can also be combined!)

(c) (6 marks) In pure on-demand paging, a **page replacement policy** is used to manage system resources. Suppose that a newly-created process has 3 page frames allocated to it, and then generates the page references indicated below.

(i) How many page faults would occur with **FIFO** page replacement? 12

A B C B A **D A B C D A B** A C B **D**

The **bold font** indicates the references that cause a page fault.

(ii) How many page faults would occur with **LRU** page replacement? 10

A B C B A **D** A B **C D A B** A **C** B **D**

The **bold font** indicates the references that cause a page fault.

(iii) How many page faults would occur with **OPT** page replacement? 7

A B C B A **D** A B **C** D A **B** A C B **D**

The **bold font** indicates the references that cause a page fault.

File and Storage Systems

15 16. Answer the following questions about file systems in general.

(a) (3 marks) In Unix, Linux, and Windows file systems, there are multiple **timestamps** (usually 3) associated with each file. What do each of these timestamps represent?

```
creation:  time when file was first created
modification:  time when file was most recently modified (e.g., written)
access:  time when file was most recently accessed (e.g., read)
```

(b) (6 marks) In class, we discussed three different techniques for organizing the data blocks for each file in a file system, namely **contiguous** allocation, **linked** allocation, and **indexed** allocation. Briefly describe each approach, identifying the strengths and weaknesses of each.

```
contiguous:  allocate file blocks consecutively on disk.
Easy to do sequential or direct (random) access to file.
Simple to implement, but prone to (external) fragmentation,
and very difficult to ‘‘grow’’ file dynamically.
linked:  linked list of available blocks from anywhere on disk.
Each block points to the next.  Ameliorates fragmentation problem,
and easy to grow a file, but random access is difficult and slow
because of many seek times required to traverse chain of pointers.
indexed:  best of both worlds; table of (direct and/or indirect)
pointers to data blocks, which can be anywhere on disk.
Solves fragmentation problem.  Supports sequential and random access.
Can optimize data block layout using cylinder groups, as in Unix.
```

(c) (6 marks) In a storage system with conventional magnetic-media disks, several different **delays** occur when servicing a request. Identify **at least three** of these delays, and comment on their relative contribution to the total delay for servicing a request.

```
seek time:  time to move read/write head from its current position
to the desired track.  May take 5-10 milliseconds (often dominant).
rotational latency:  time for desired sector on target track to spin
under the read/write head.  May take 1-2 milliseconds (medium)
transfer time:  time to read the target sector and transfer bytes
to host computer.  A millisecond or less (low).
wait time:  time spent in I/O queue waiting for service.
Could be 0 of more milliseconds, depending on number of requests
queued and the disk request scheduling policy used.
```

File System Details

12 17. The following page shows some output from some file-system related commands on a local Linux system. Use this output and your knowledge of Linux file systems to answer the following questions.

(a) (1 mark) **How many** different file systems are accessible on this Linux system?
`11`

(b) (1 mark) Which file system is the **fullest** (in terms of percent occupancy)?
`/home/research`

(c) (1 mark) Which file system has the **largest** physical storage capacity?
`/home/scratch`

(d) (1 mark) Which file system has the **fewest** bytes currently stored?
`/boot`

(e) (1 mark) Which disk partition (if any) is being used for **swap space**?
`/dev/sda3`

(f) (1 mark) What is the **type** of the `/tmp` file system?
`ext3`

(g) (1 mark) What is the **type** of the `/proc` file system?
`proc`

(h) (1 mark) How many file systems are remotely mounted using **NFS**?
`7`

(i) (1 mark) Which file system is remotely mounted on server **nsh**?
`/home/grads`

(j) (1 mark) Is this NFS service provided using **UDP or TCP**?
`TCP`

(k) (2 marks) What **block sizes** are used for reading and writing via NFS?
`32 KB for reading, 4 KB for writing`

```
[carey@csl]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             20315844   7513824   11753380 39% /
/dev/sda5             10581704   5181092    4854404 52% /tmp
/dev/sda1              1019208     50356     916244  6% /boot
tmpfs                  1684784     77164    1607620  5% /dev/shm
nsi:/export/research 247709760 203879488   31247360 87% /home/research
nse:/export/scratch 3361364788 528637368 2661979928 17% /home/scratch
nsj:/export/proj/dsl 485097928 184513976  275942416 41% /home/dsl
nsg:/export/ug       381885660  79136396  283350608 22% /home/ugc
nsf:/export/ug       381885660  88768580  273718424 25% /home/ugb
nsh:/export/grads    789574392 507946744  241519616 68% /home/grads
nsb:/export/ug       381885660  76958700  285528304 22% /home/uga

[carey@csl]$ cat /etc/fstab
LABEL=/                 /                       ext3    defaults        1 1
LABEL=/tmp              /tmp                    ext3    defaults        1 2
LABEL=/boot             /boot                   ext3    defaults        1 2
tmpfs                   /dev/shm                tmpfs   defaults        0 0
devpts                  /dev/pts                devpts  gid=5,mode=620  0 0
sysfs                   /sys                    sysfs   defaults        0 0
proc                    /proc                   proc    defaults        0 0
LABEL=SWAP-sda3         swap                    swap    defaults        0 0

[carey@csl]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
nsi:/export/research on /home/research type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
nse:/export/scratch on /home/scratch type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
nsj:/export/proj/dsl on /home/dsl type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
nsg:/export/ug on /home/ugc type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
nsf:/export/ug on /home/ugb type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
nsh:/export/grads on /home/grads type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
nsb:/export/ug on /home/uga type nfs (rw,intr,tcp,rsize=32768,wsize=4096)
```

General Operating Systems Knowledge

15 18. Throughout CPSC 457 this year, there were several recurring **themes** (i.e., ideas that applied quite broadly across several topics).

(a) (5 marks) One of these themes was **virtualization**. Identify **three** contexts in which virtualization was used as a solution technique. Briefly discuss the technical issues involved, and the benefits of the virtualization approach to the problem.

```
virtual machines:  can run multiple OS on same machine at same time
virtual memory:  separates physical memory management from logical view
virtual file system:  common API for all file systems (local or remote)
Virtualization abstracts away the physical hardware and its constraints,
providing simplicity and flexibility for the users.
```

(b) (5 marks) A second theme was **hardware support**. Identify **three** contexts in which hardware support was used as a solution technique. Briefly discuss the technical issues involved, and the benefits of a hardware-based approach to the problem.

```
multi-processors:  hardware support for parallel computing
multi-core:  hardware support for fine-grain thread concurrency
MMU/TLB: hardware support for address translation, page table lookup
test-and-set, SWAP: atomic hardware instructions for mutual exclusion
protection:  distinguish user-mode priviliges from kernel-mode
Hardware solution provides fast, specialized, high-performance
solution for key OS features and requirements.
```

(c) (5 marks) A third theme was **caching**. Identify **three** contexts in which caching was used as a solution technique. Briefly discuss the technical issues involved, and the benefits of caching as a solution.

```
CPU: on-chip instruction/data caches to avoid memory latency
buffer cache:  in-memory caching to avoid disk latency
disk cache:  remember recent data blocks to avoid disk access latency
write cache:  buffer outgoing writes to improve system performance
storage:  sequential read-ahead and prefetch for I/O controller
file system:  in-memory caching of superblock, inodes, directory info
NFS: client-side caching of info to avoid network latency
Caching optimizes for the common case, and improves performance.
```

*** THE END ***