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

Spring 2012 Anthony D. Joseph and Ion Stoica

Final Exam *Solutions*

May 11, 2012
CS162 Operating Systems

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

General Information:
This is a **closed book and TWO 2-sided handwritten notes** examination. You have 170 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	21	
2	19	
3	24	
4	12	
5	12	
6	12	
TOTAL	100	

1. (21 points total) Short answer questions.

a. (12 points) True/False and Why? **CIRCLE YOUR ANSWER.**

i) The size of a process' Working Set may depend on the number of processes currently running.

TRUE FALSE

Why?

TRUE. *The correct answer was worth 1 point. Good reasoning was worth 2 points. A process' Working Set is determined by the process' memory reference pattern. Since virtual time is used to track this behavior, it is unaffected by other processes.*

ii) Doubling the block size in the UNIX file system will double the maximum file size.

TRUE FALSE

Why?

FALSE. *The amount of data in each data block is doubled AND the size of each singly, doubly, and triply indirect block is also doubled increasing the total number of data blocks that can be indexed. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

iii) Adding a memory cache never hurts performance.

TRUE FALSE

Why?

FALSE. *Adding a cache increases the time to perform a reference IN ALL CASES. Caching can hurt performance when the hit rate is low. Also, the first access is always slower with a cache.*

iv) Address translation (virtual memory) is useful even if the total size of virtual memory as summed over all possible running programs is guaranteed to be smaller than a machine's total physical memory.

TRUE FALSE

Why?

TRUE. *Address translation simplifies the loading/linking process and allows programs to be placed anywhere in physical memory. Also enforces isolation between programs.*

b. (4 points) List two reasons why operating systems like Nachos disable interrupts when a thread/process sleeps, yields, or switches to a new thread/process?

Interrupts are disabled for two reasons:

(1) To prevent context switching from occurring while registers are being saved/restored.

(2) To enable a thread/process to add itself to a queue (e.g., a wait queue) without allowing another thread to interrupt the action.

Each answer is worth 3 points. If the answer was not specific about the potential conflict that could occur, we deducted 1 point.

c. (2 points) Give a definition for a non-blocking write operation.

Caller (sender) returns immediately and the caller is informed later whether the write was successful or not. (one pt for immediate return and one point for being informed about success later).

d. (3 points) Explain what causes thrashing in a system that uses virtual memory management AND how to reduce the likelihood of thrashing occurring.

Thrashing occurs when memory pages are constantly being swapped out of main memory and to disk at a rate too rapid to accomplish any real work. In other words, the system is spending all of its time swapping pages and not doing computation.

2. (19 points total) Networking.

a. (5 points) If the TCP transmission window for unacknowledged bytes is 1000 bytes, 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 receiver has received the last byte? You may ignore the TCP handshake and assume that no packets are lost for this particular problem (but remember that TCP doesn't know that) and that ACKs are essentially 0 bytes long. *Write out your answer in symbolic form for partial credit.*

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. 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.*

b. (6 points) Consider a distributed system with two communication primitives: SEND and RECEIVE. The SEND primitive sends a message to a specified process on a remote machine. The RECEIVE primitive specifies a remote process to receive from, and blocks if no message from that remote process is available, even though messages may be waiting from other remote processes. There are no shared resources, but processes need to communicate frequently. Do the primitives allow deadlock? *Discuss why or why not. (Use no more than two sentences.)*

Yes. Suppose that all the processes are idle (i.e., not waiting on other processes). Now A sends to B and waits for a reply, B sends to C and waits for a reply, and C sends to A and waits for a reply. All the conditions for deadlock are now fulfilled.

c. (8 points). Your new boss at Orange Computer proposes a new sliding window-based reliable network transport protocol.

The protocol works as follows: instead of sending an acknowledgement for every packet, it only sends an acknowledgement for every 5 packets (by sequence number). Thus, ACK #1 is sent when packets with sequence numbers 0-4 have been received. ACK #2 is sent when packets with sequence number 5-9 have been received.

Acknowledgements do not cover packets from earlier sequence numbers and can be sent out of order (i.e., ACK #2 can be sent when packets with sequence numbers 5-9 have been received successfully, even if packets with sequence numbers 0-4 have not been received). The protocol also uses a retransmission timer for each packet sent.

List a ***one sentence*** advantage and ***one sentence*** disadvantage of using this new protocol.

i). Advantage:

Lower overhead for acknowledgements versus sending one per packet. This reduction is important for low bandwidth links (e.g., cellular links).

Each answer was worth four points, 2 points for the correct choice and 2 points for a good argument.

ii). Disadvantage:

More retransmissions, since an ack covers four packets, the loss of one packet requires that all the packets be retransmitted.

3. (24 points total) Shared Memory.

In this problem you will outline an implementation of shared memory in an operating system like 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. *The addresses must be page-aligned.* If successful, it returns a RegionID identifying the shared region. **SharedRegionAttach** 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. Regions must be non-overlapping.

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 **Exec**'s 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 and modify (and vice versa).

Answer each of the following questions descriptively *at a high level*. Your answers should be brief, and capture the essence of what needs to be implemented at each stage. *You do not need to provide psuedocode.*

a. (5 points) What should SharedRegionCreate do to the caller's page table and any other system data structures? For part a, you may assume that all the shared pages are in memory.

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.

b. (4 points) What should SharedRegionAttach do to the caller's page table and any other system data structures?

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 (i.e., setting the virtual to physical page mapping to *point to the shared physical page). Basically, the caller's page table should copy the PTE's from the creator's page table.*

c. (4 points) 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. (3 points) 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 PTE's 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 PTE's in all page tables sharing the page have to be set to the same values.

e. (2 points) How should the process of destroying an address space change to account for shared pages?

Two possible 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. (3 points) List THREE error conditions that implementations 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. However, answering that checking that the beginAddress was valid was one error check, and checking that the endAddress was a second is not sufficiently creative, since they are essentially the same check.

g. (3 points) What changes would your solution need if a shared region was shared by more than two processes (e.g., a parent and two child processes).
No changes would be necessary as the existing changes would already support more than two processes sharing the same shared region.

4. (12 points total) Two-Phase Commit.
You're hired by Macrohard Corporation to develop a system for distributing updates to their operating system. To ensure proper operation, all clients systems must be running the same version of the software. New OS versions are first distributed to clients using a Content Distribution Network. Next, two-phase commit is used in order to ensure that, despite computer crashes, either (a) everyone eventually switches to the new OS version identified by the version number, NEWVER, or (b) no one switches to the new OS version and everyone continues using OLDVER. Assume that a machine takes *some* significant amount of time to reboot and recover after a crash.

The participants in this problem are the OS Version Server (OSVS) and three client nodes (C1, C2, C3). The steps of the two-phase commit protocol are listed below, in time order:

1. OSVS: write "begin transaction" to its log
2. OSVS → C1: "New version is NEWVER." (→ means OSVS sends message to C1)
3. C1: write "New version is NEWVER." to its log
4. C1 → OSVS: "Prepared to commit"
5. OSVS → C2: "New version is NEWVER."
6. C2: write "New version is NEWVER." to its log
7. C2 → OSVS: "Prepared to commit"
8. OSVS → C3: "New version is NEWVER."
9. C3: write "New version is NEWVER." to its log
10. C3 → OSVS: "Prepared to commit"
11. OSVS: write "New version is NEWVER." to its log
12. OSVS: write "commit" to its log
13. OSVS → C1: "commit"
14. C1: write "got commit" to its log
15. C1: OS Version = NEWVER
16. C1 → OSVS: "ok"
17. OSVS → C2: "commit"
18. C2: write "got commit" to its log
19. C2: OS Version = NEWVER
20. C2 → OSVS: "ok"
21. OSVS → C3: "commit"
22. C3: write "got commit" to its log
23. C3: OS Version = NEWVER
24. C3 → OSVS: "ok"
25. OSVS: OS Version = NEWVER

a. (3 points) If OSVS crashes after step 11 and no one else fails, what OS Version will everyone end up using, once OSVS reboots and recovers? ***Give the reason why.***

OLDVER. The transaction does not commit until step 12, so OSVS will abort the transaction when it reboots.

b. (3 points) If C3 crashes after step 9 and no one else fails, what key will everyone end up using, once C3 reboots and recovers? ***Give the reason why.***

OLDVER. OSVS will time out on trying to communicate with C3 and will abort the transaction.
***Note**: Since message communications take on the order of milliseconds to execute and crashes take **much** longer to recover from (think of how long it takes to reboot Unix or Windows), it's not reasonable to assume that C3 will recover before step 11, so that OSVS will continue as if nothing had happened. Therefore OSVS will always notice that C3 crashed and will abort the transaction.*

c. (3 points) If C3 crashes after step 12, what recovery steps must it take after it reboots in order to achieve the correct global state with respect to which OS Version to use?

C3 will look at its log and see that there is an entry for a transaction for which it is "prepared to commit" but hasn't heard a final resolution of commit or abort yet. It sends a message to OSVS asking whether to commit or abort the transaction. OSVS will tell it to commit the transaction, implying that C3 will do steps 22, 23, 24, and will end up using the new OS Version.

d. (3 points) If OSVS crashes after step 23, what recovery steps must it take after it reboots in order to achieve the correct global state with respect to which OS Version to use?

OSVS will look at its log and see that there is an entry for a transaction that it has commited but not finished. (If it were finished, the "front" pointer for the log would have been moved past it.) OSVS will contact C1, C2, and C3, and tell them to commit the transaction and then finish doing the transaction itself. It has to tell all of the clients because it doesn't know whom it told to commit before it crashed -- it only knows that it wrote "commit" to its log. Put another way, OSVS "restarts" the transaction at step 13 since the only thing it is sure of during recovery is that it executed step 12, which wrote "commit" to the log.

5. (12 points total) Caching.
Consider a computer with 8-bit addresses that uses an LRU (Least Recently Used) cache consisting of four 4-byte blocks. Assume a program that accesses the memory according to the following sequence: 0x20, 0x45, 0x11, 0x44, 0x20, 0x70, 0x71, 0x72, 0x10, 0x44, 0x32, and 0x12. Initially, the cache is empty.

a) (4 points) Assume a direct mapped cache. Show the size (in bits) of the cache tag, cache index, and byte select fields. In addition, show the content of every cache block after each memory access: each entry of the table below should list **all** the addresses of the bytes that are cached in the corresponding block.

cache tag	cache index	byte select
4	*2*	*2*

Cache content

Block	0x20	0x45	0x11	0x44	0x20	0x70	0x71	0x72	0x10	0x44	0x32	0x12
0	*0x20 -23*		*0x10 -13*		*0x20 ,23*	*0x70, 0x73*	*0x70, 0x73*	*0x70, 0x73*	*0x10 -13*		*0x30 -33*	*0x10 -13*
1		*0x44 -47*		*0x44 -47*						*0x44 -47*		
2												
3												

b) (4 points) Repeat question (a) for a fully associative cache.

cache tag	byte select
6	*2*

Cache content

Block	0x20	0x45	0x11	0x44	0x20	0x70	0x71	0x72	0x10	0x44	0x32	0x12
0	*0x20 -23*										*0x30 -33*	
1		*0x44 -47*										
2			*0x10 -13*									
3						*0x70 -73*						

c) (4 points) Repeat question (a) for a 2-way associative cache. For each block indicate the set is belonging to.

cache tag	cache index	byte select
5	*1*	*2*

Cache content

Set	Block	0x20	0x45	0x11	0x44	0x20	0x70	0x71	0x72	0x10	0x44	0x32	0x12
0	0	*0x20 -23*					*0x70 -73*					*0x30 -33*	
0	1			*0x10 -13*						*0x10 -13*			
1	2		*0x44-47*										
1	3												

6. (12 points total) Circular queue and concurrency.
Circular queue is a data structure often used to implement various functionalities in the OS. For example, the sending and receiving buffers in TCP are implemented using circular queues. Consider a circular queue of size N. The figure below shows one instance of a circular queue for N=8, where the `head` points to entry 1, and the `tail` to entry 4. The `head` points to the first element in the queue, while the `tail` to the last element in the queue. If the queue is empty, both `head` and `tail` are initialized to -1. Note that both the `head` and `tail` are incremented using modulo N operator. The pseudocode below implements the `enqueue()` operation. We assume that both these functions are synchronized, i.e., they lock the queue object when executing.

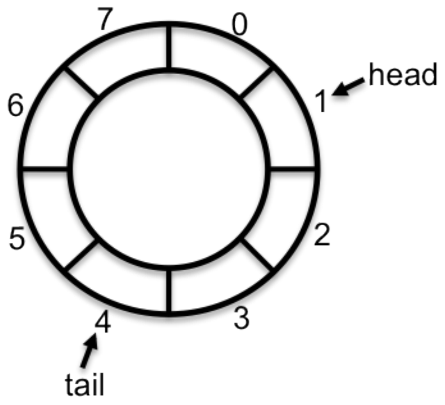

Figure: A circular queue of size N=8 storing 4 entries at positions 1, 2, 3, and 4, respectively.

```
// enqueue new item "data"; return false if queue full
synchronized bool enqueue(item data) {
      if (head == -1) { // queue is empty
            head = tail = 0;
            queue[tail] = data;
      } else { // there is at least one item in the queue
            if ((tail + 1) modulo N != head) {
                  tail = (tail + 1) modulo N;
                  queue[tail] = data;
            } else { // queue full
                  return false;
            }
      }
      return true;
}
```

a) (4 points) Write the pseudocode for the `dequeue()` operation. `dequeue()` should return the item at the head of the queue if queue not empty, and `null`, if queue is empty.

```
synchronized item denque() {
     if (head == -1) {
          Return null;
     } else {
          item = dequeue[head];
          if (head == tail) {
               head = tail = -1; // empty queue
          } else {
               head = (head + 1) modulo N;
          }
          return item;
     }
}
```

b) (2 points) Assume N = 8, and assume there were 100 `enqueue()` operations, out of which 5 have failed, and 98 `dequeue()` operations, out of which 7 have failed. Assume initially the queue is empty, and it never gets empty again after the first item is inserted. What are the values of `head` and `tail` after all `enqueue()` and `dequeue()` operations take place?.

The tail is incremented for every successful eqnueue() operation, while head is incremented for every successful dequeue() operations. Since there are 95 successful enqueue() operations, tail = (95-1) modulo 8 = 6. Similarly, since there are 91 successful dequeue() operations, head = 91 modulo 8 = 3.

c) (6 points) Assume circular queue of integers with N = 3, and assume each of the following three operations is called by a different threads. In particular, thread T1 calls `enqueue(10)`, thread T2 calls `enqueue(12)`, while thread T3 calls `dequeue()`. Initially, the circular queue is empty. Please specify what are the possible entries in the circular queue after all three operations are performed, and what is the value returned by the `dequeue()` operation. Use the following table to show your answer. (If a queue entry is outside [head:tail], then use "-" to represent its content.)

thread execution order	head	tail	queue[0]	queue[1]	queue[2]	dequeue value
T1, T2, T3	*1*	*1*	-	*12*	-	*10*
T1, T3, T2	*0*	*0*	*12*	-	-	*10*
T3, T1, T2	*0*	*1*	*10*	*12*	-	*null*
T2, T1, T3	*1*	*1*	-	*10*	-	*12*
T2, T3, T1	*0*	*0*	*10*	-	-	*12*
T3, T2, T1	*0*	*1*	*12*	*10*	-	*null*