Principles of Computer System Design
Midterm Exam
Wednesday, Oct 8th, 2014

NAME: Solution

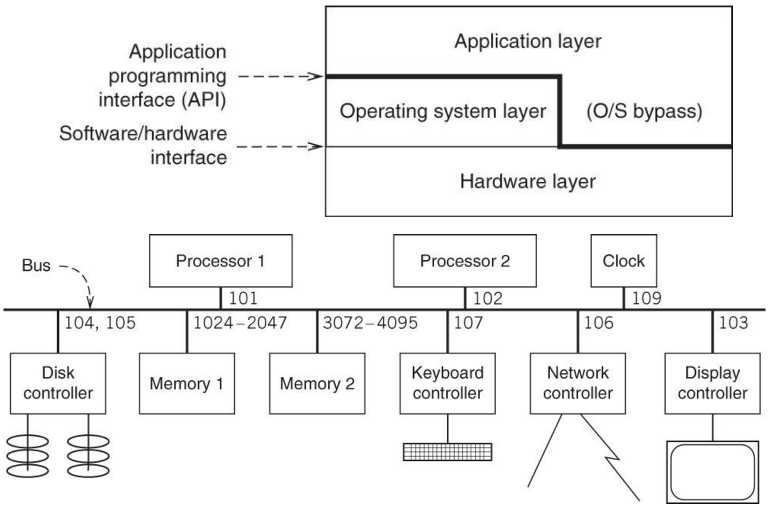
UFID:

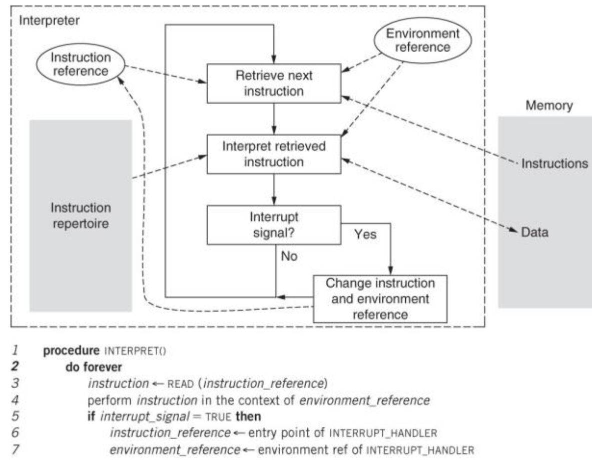
Please read each question carefully, to avoid any confusion. This exam should have 10 pages printed double-sided (the last 2 pages are blank); before you begin, make sure your copy contains all pages. The exam is closed book, closed notes. Each question has its number of points identified in brackets.

GOOD LUCK!

QUESTION	***POINTS SCORED***
1 [25]	
2 [25]	
3 [30]	
4 [20]	
TOTAL	

1) [25] Naming and layers:





1.a) [10] Suppose each processor above has a single kernel/user bit, and a single memory domain register that enforces bounds (lower L, upper U) and permissions (Read,Write,eXec). Suppose processor 1's domain register has (L=1024,U=1535;perm=R,X). Suppose an application in P1 issues the following instructions; *which of these instructions result in crossing from the application to OS layer? Why*? *Notation*: instruction address: opcode, register, data address

3072: LOAD R1, 1030

Crosses – instruction address 3072 outside domain

1024: STORE R2, 1400

Crosses – permission for 1400 does not allow writes

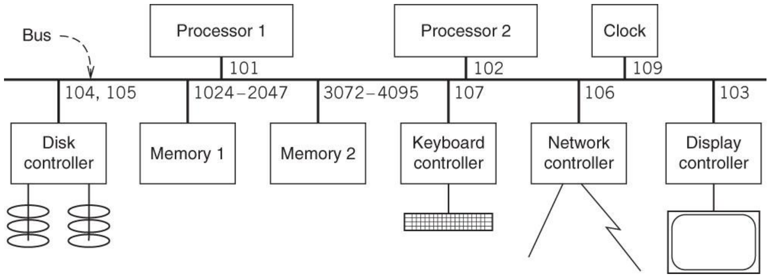
1300: LOAD R3, 1200

Does not cross; both instruction, data within bounds, permission ok

1240: SVC

Crosses – it's the behavior of this instruction

1.b) [15] Now assume that the processor supports page-based virtual memory. Suppose the page size is 1024 bytes. Given the page tables for processors P1 and P2 to be as follows (VA, PA: virtual and physical addresses, respectively).



P1's page table			**P2's page table**		
VA	*PA*	*Perm*	*VA*	*PA*	*Perm*
1024	1024	RW	0	1024	RW
0	3072	RX	1024	3072	RX
3072	0	R	2048	0	R

i) Would it be possible for a thread in P1 to send a message to a thread in P2 using a shared buffer? If so, describe where the buffer and lock need to reside in physical memory

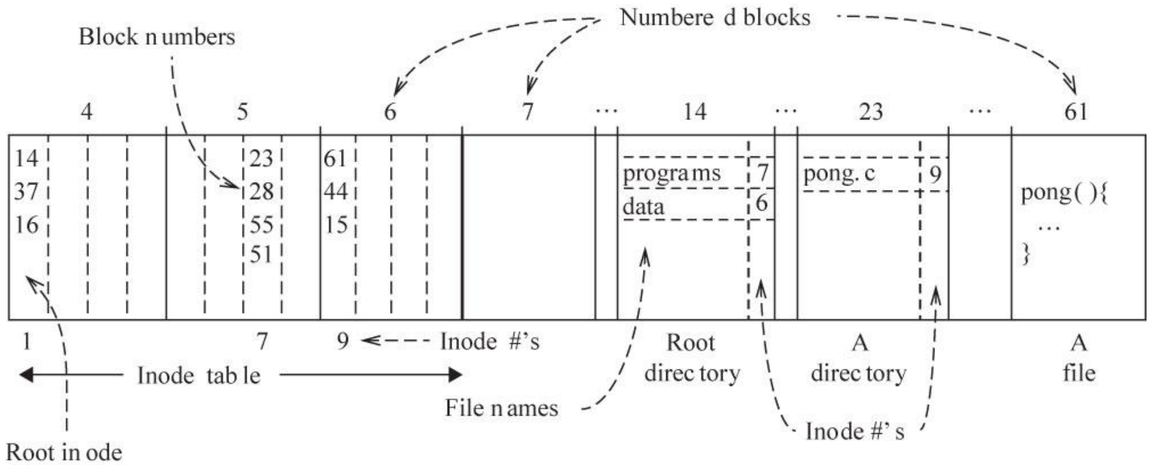
Yes – page in PA 1024 shared read/write by both. Buffer and lock should be in this page

ii) Would it be possible for threads in P1 and P2 to use *memory mapped I/O* to *read* data from the keyboard controller? What about to *send* data over the network interface?

Possible to read from keyboard controller (PA 0 mapped readable; keyboard is in address 107)
Not allowed to send to network (PA 0 mapped readable; network in address 106)

2) [25] File systems, RPC, NFS:

a) [10] Consider the figure below discussed in class.



i) Suppose an application opens and reads /programs/pong.c. Name *all* block numbers read from disk, and the order in which they are read.

4, 14, 5, 23, 6, 61 (44,15)

ii) Suppose you create a hard link "/myhardlink" that links to /programs/pong.c, and a soft link "/mysoftlink" that also links to /programs/pong.c. *Which i-nodes and blocks*, if any, would need to be modified to accommodate these two links? Explain.

myhardlink: change block 16 to add "myhardlink | 9"
increment reference count of inode #9

mysoftlink: create inode "n"; stores name "/programs/pong.c"
change block 16 to add "mysoftlink | n"

b) [15] Alice is using both her desktop D and laptop L to work on a file served by an NFS server S. The NFS file system is mounted on both D and L in /home. Alice opens the file "/home/alice/file.txt" in both computers. Assume there is no caching.

i) What LOOKUP messages are sent to S when the system call to open() the file is issued by D and L? State your assumptions.

Two lookups: LOOKUP (root file handle, "alice") -> alicefh
LOOKUP (alicefh, "file.txt") -> filefh

ii) Are the generation counts the same in the file handles received by D and L? Briefly explain.

Yes; counts only differ if file unlinked and inode reused

iii) At a later point in time, Alice notices that two subsequent NFS reads issued by D, for the same file handle and offset, result in a successful read followed by a stale file handle error. How could that happen?

Client L has unlinked file.txt in between the reads

3) [30] Virtualization and enforced modularity

i) [5] What must be the permissions associated with the memory domain where the bounded buffer is stored in order to enforce modularity?

K, R, W – must be readable and writable by kernel only

ii) [5] What privilege level is the processor on when the RSM instruction is issued to acquire the bounded buffer's lock? Why?

Kernel mode – the memory domain where the lock resides has K permission

iii) [10] Can a race condition occur in the bounded buffer send/receive with a *single* client and *single* service? Explain

Yes (in the version that did not use locks) – e.g. writes to multiple-cell variables by sender (for instance, in) can be partially read by service

Refer to the YIELD implementation:

```
1  shared structure processor_table[7]  // each processor maintains the following information
2      integer topstack                   // value of stack pointer
3      byte reference stack               // preallocated stack for this processor
4      integer thread_id                  // identity of thread currently running on this processor
5  shared structure thread_table[7]     // each thread maintains the following information
6      integer topstack                   // value of the stack pointer
7      integer state                      // RUNNABLE, RUNNING, or FREE
8      boolean kill_or_continue           // terminate this thread? initialized to CONTINUE
9      byte reference stack               // stack for this thread

10 procedure YIELD ()
11     ACQUIRE (thread_table_lock)
12     ENTER_PROCESSOR_LAYER (GET_THREAD_ID(), CPUID)     // See caption below!
13     RELEASE (thread_table_lock)
14     return
15
16 procedure SCHEDULER ()
17     while shutdown = FALSE do
18         ACQUIRE (thread_table_lock)
19         for i from 0 until 7 do
20             if thread_table[i].state = RUNNABLE then
21                 thread_table[i].state ← RUNNING
22                 processor_table[CPUID].thread_id ← i
23                 EXIT_PROCESSOR_LAYER (CPUID, i)
24                 if thread_table[i].kill_or_continue = KILL then
25                     thread_table[i].state ← FREE
26                     DEALLOCATE(thread_table[i].stack)
27                     thread_table[i].kill_or_continue = CONTINUE
28         RELEASE (thread_table_lock)
29     return                                          // Go shut down this processor

30 procedure ENTER_PROCESSOR_LAYER (tid, processor)
31     thread_table[tid].state ← RUNNABLE
32     thread_table[tid].topstack ← SP                 // save state: store yielding's thread SP
33     SP ← processor_table[processor].topstack        // dispatch: load SP of processor thread
34     return

35 procedure EXIT_PROCESSOR_LAYER (processor, tid)    // transfers control to after line 14
36     processor_table[processor].topstack ← SP        // save state: store processor thread's SP
37     SP ← thread_table[tid].topstack                 // dispatch: load SP of thread
38     return
```

iv) [10] Suppose the ACQUIRE and RELEASE of lines 18 and 28 were moved to be between lines 20-21 (ACQUIRE) and 21-22 (RELEASE). Would this approach work? If so, explain why; if not, provide a concrete example of a race condition that might occur.

It does not work; for instance, two processors reach line 20 and decide that the same thread “i” is runnable; then, same thread will run in two processors

4) [20] Multiple choice and true/false questions

i) [2] The file descriptor of an open file is stored in its inode
[True | <u>False</u>]

ii) [2] An open file's offset for reads and writes is stored in its inode
[True | <u>False</u>]

iii) [2] Symbolic links can be used both for directories and files
[<u>True</u> | False]

iv) [2] A name resolver that uses a table allows for synonyms
[<u>True</u> | False]

v) [2] The SVC instruction provides address of a gate as its argument
[True | <u>False</u>]

vi) [5] In contrast to a hierarchical file system, consider a "flat" file system which only has one context. *Circle all correct statements:*

i) A flat file system cannot support access permissions
ii) A flat file system cannot support symbolic links
iii) <u>A flat file system cannot hold two files with the same name</u>
iv) A flat file system cannot support 'hard' links
v) <u>Lookups do not need to be recursive in a flat file system</u>

vii) [5] Application A in client C opens file F in an NFS mounted directory. While A holds F open, the server suffers a short power outage and quickly reboots. *Circle all correct statements*:

i) The application's file descriptor becomes invalid
ii) The file handle for file F becomes invalid
iii) <u>The NFS client stub may re-send multiple RPC calls for the same idempotent NFS operation during the reboot period</u>
iv) The NFS client must re-mount the file system to recover
v) The NFS server sends an RPC error message to the client upon reboot

Scratch space

Scratch space