CSE 451 Midterm Exam
November 9, 2012

Your Name:

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 50 points in all. Write all of your answers directly on this paper. Make your answers as concise as possible (you needn't cover every available nano-acre with writing). If there is something in a question that you believe is open to interpretation, then please go ahead and interpret **but** state your assumptions in your answer.

Problem	Points Scored	Points Possible
1		10
2		6
3(a)		9
3(b)		25
Total		50

Problem 1: (10 points)

Consider a demand paging system, where a dedicated disk is used for paging, and file system activity uses other disks. Measured utilizations (in terms of **time**, not space) are:

CPU utilization	20%
Paging disk	99.7%
Other I/O devices	5%

For each of the following changes, say what its likely impact will be on **CPU** utilization: will it will probably significantly increase, marginally increase, significantly decrease, marginally decrease, or have no effect on the CPU utilization, and why.

(a) Get a faster CPU

Significantly decrease. Because the system is thrashing, speeding up the CPU will just lead to more time spent waiting for page faults.

(b) Get a bigger paging disk

No change. A larger disk is not necessarily a faster disk.

(c) Increase the degree of multiprogramming

*Significantly **decrease**. Because the system is thrashing, increasing the number of tasks will increase the frequency of page faults, and decrease the amount of useful computation between each page fault.*

(d) Decrease the degree of multiprogramming

Significantly increase. Reducing the number of tasks will increase the amount of useful computation between each page fault.

(e) Get faster other I/O devices

No effect or insignificant increase. The other I/O is not limiting the CPU performance, so speeding that up will have little to no effect.

Problem 2: (6 points)

Suppose an architecture with paged segmentation has a 32-bit virtual address that is divided into fields as follows:

4 bit segment number	12 bit page number	16 bit offset

The segment and page tables are as follows (all values are in hexadecimal):

Segment Table

0	Page Table A
1	Page Table B
x	(rest invalid)

Page Table A

0	CAFE
1	FEED
2	BEEF
3	BAAA
x	(rest invalid)

Page Table B

0	F000
1	B0B0
2	CACA
x	(rest invalid)

Find the physical address corresponding to each of the following (hexadecimal) virtual addresses (answer "invalid virtual address" if the virtual address is invalid):

a) 00000000

CAFE0000. The entry in the page table is the physical page frame number – many students took it as the page frame address. However, you can tell it is not the full address, as it does not have 0000 as the bottom sixteen bits (the page size).

b) 20022002

invalid

c) 10015555

B0B05555

Problem 3: (34 points = 9 + 25)

Some researchers have advocated using message passing as an alternative to shared memory, as a way of eliminating the possibility of race conditions from concurrent programs. In one proposal, every instance of an object has its own thread, and only that thread can touch the data of the object. To call a procedure on an object, the program sends the object a "message" specifying the requested procedure (and its parameters); the thread in the object receives the message and executes the operation. If there is a return value, the receiver in turn sends the result to the original sender.

Thus instead of locks and condition variables, we have two primitives:

send(message): send message to the object, and return only once the message has been received

receive(): wait until there is a message to receive, and then return it

a) Give an example program using send() and receive() that can deadlock.

Because the sender waits for receipt, the easiest deadlock is a thread with itself:

Object 1:

```
while (msg = receive()) {
  obj1->send();
}
```

However, we accepted anything that would produce infinite circular waiting.

b) To show that message passing is no more powerful than condition variables, implement send(msg) and receive() using locks and Hansen (=Mesa)-style condition variables. (Your implementation should use shared memory, of course.) You may assume the existence of a cloneMessage(msg) routine that allocates and makes a copy of the message for the receiver to use internally.

A synchronous send/receive is a way of implementing a zero-length bounded buffer – both the sender and the receiver must be present at the same time to pass the contents of the message. We gave partial credit for hygiene: accessing shared state inside of a lock, using while around every "wait". We gave more partial credit for noticing that both the sender and the receiver need to wait, and for having signal in both the sender and the receiver. Full credit only applied to solutions that worked with an arbitrary number of senders – the solution is a bit easier if you handle one sender at a time. In the draft solution, we have a separate lock for that, but several approaches work. The danger is another sender slipping in and grabbing the wakeup for a previously delivered message (not what you intended). (By the definition of the problem, there can only be one receiver per object.)

```
Lock lock, sendLock;
Condition receiveWaiting, sendWaiting;
int senderArrivals, receiversDone;
Message clone;

send(msg) {
  sendLock.Acquire();
  lock.Acquire();
  clone = CloneMsg(msg);
  senderArrivals++;                // tell receiver clone is ready
  receiveWaiting.Signal();         // wake up a receiver if they are waiting
  while (receiversDone < senderArrivals)  // wait for receiver to fetch clone
    sendWaiting.Wait(lock);
  lock.Release();
  sendLock.Release();
}
receive() {
 Message myMessage;

  lock.Acquire();
  while (senderArrivals <= receiversDone)  // wait for sender to initialize clone
    receiveWaiting.Wait(lock);
  myMessage = clone;        // ok for me to grab clone
  receiversDone++;
  senderWaiting.Signal();  // tell sender I'm done
  lock.Release();
  return myMessage;
}
```