CSE 451 – Spring 2006
Midterm 1

1. *[3 points]*
   (a) What is "multiprogramming"?

   Loading more than one process in memory and switching the CPU among them.

   (b) What is "multiprocessing"?

   Running on hardware that has more than one processor.

   (c) What is "direct memory access (DMA)"

   Devices that can transfer into/out of memory, without involving the CPU for each byte transferred.

2. *[4 points]*
   Briefly describe **two** methods by which a process can escalate privilege – that is, can accomplish some operation that it doesn't have the privilege to do itself.

   1. *make a system call*
   2. *invoke a setuid program (as a separate process)*
   3. *communicate with a running server process that has higher privilege*

3. *[2 points]*
   What is meant by "the separation of mechanism and policy," as applied to operating systems?

   Mechanism is how things are done – what functionality is implemented. Policy addresses how to use the mechanism to achieve some higher level goal(s) – what should the effect be at an abstract level.

4. *[2 points]*
   (a) What is the main idea of, and the main goal of, the layered approach to OS implementation??

   To structure the OS code into a set of layers, so that each can use only the interfaces provided by the one(s) below it. The idea is to simplify building, debugging, and maintaining the OS by providing some useful structure to its

implementation.

(b) What is the main problem with trying to implement OS's this way?

The actual interactions among the natural components of the OS do not follow a layered organization – there are cycles. (Additionally, there can be some runtime overhead penalty involved with having to strictly adhere to the layered organization.)

5. *[2 points]*
(a)What is the main idea of, and the main goal of, the microkernel approach to OS implementation?

The main idea is to implement the minimum functionality possible in the OS kernel, and to push everything else up to user-level. The goals are to provide structure (see Q4), debuggability (it's easier to debug user-level modules than kernel modules), and flexibility (the user-level modules can be replaced by customized ones, if the user finds that useful).

(b) What is the main problem with trying to implement OS's this way?

Overhead – all communication between normal apps and these user-level extensions of what is traditionally part of the OS must flow through the kernel, for protection reasons. The message passing required can add considerable overhead compared to a monolithic kernel.

6. *[3 points]*
Suppose you decide to distribute your source as pre-compiled binaries, packaged in a library.

(A) What is the advantage of using a library over just distributing the .o files?

Convenience / bullet-proofing – there is just one file for the user to fetch and manage, rather than hundred's. Using a library, it would be very difficult for the user to end up with an inconsistent set of .o's (those compiled from different versions of the code, say).

(B) Give two mechanisms you could use in your code (distributed this way) to allow users to customize its actions.

1. normal API definitions provide parameters that customize the actions of individual calls.
2. the code could fetch environment variables and react to their settings. (You can

view this is another way to pass arguments to the routines.)

(C) Suppose protecting your source isn't a concern for you. What advantage to the user is there in your distributing a library rather than (just) source?

Simply compiling the code may be complicated, requiring that many tools and packages beyond the C compiler (say) be available. Distributing pre-compiled versions of the code relieves the end-user of having to establish the build environment for the library.

7. *[2 points]*
   What is the (primary) problem with locks and semaphores that monitors are intended to solve?

The entire burden of using locks and semaphores correctly falls on the programmer, who is notoriously unreliable. Monitors add some language/compiler analysis, which absolutely prevents certain types of bugs (e.g., failing to release the monitor lock on some code paths out of the monitor).

8. *[4 points]*
   The traditional Unix process creation call, fork(), takes no arguments and creates a new process by copying the parent. In contrast, the Windows process creation call, CreateProcess(), takes many arguments, including the name of an initial application to be loaded into the address space of the new process.

   (a) Give an advantage of the Windows approach over Unix fork().

   It may be more efficient, because an essentially useless copy of the parent's address space is avoided. (Note: there are ways to mitigate this overhead that we didn't talk about, but are important.)

   (b) Give an advantage of fork() over the Windows approach.

   fork() is completely general – the author of the parent code has the opportunity to do whatever s/he wants to initialize the child, because the child starts by running that person's code. With CreateProcess(), the ways in which the parent can affect the child are limited to what is provided by the many parameters of that call.

9. *[4 points]*
   (a) What is the principle advantage of user-level thread implementations over kernel-level?

Performance.

(b) What is the principle disadvantage?

If a user-level thread blocks, it blocks the kernel-level thread with it, making it difficult to fully exploit the available resources of the system.

10. *[4 points]*
(a) What properties must a correct solution of the **critical section** problem have?

1. mutual exclusion
2. progress
3. bounded waiting

(b) What conditions are required for **deadlock** to be possible?

1. mutual exclusion
2. hold and wait
3. no preemption
4. circular wait

11. *[2 points]*
What is the difference between Mesa and Hoare monitors?

In a Hoare monitor, a signal'ing thread blocks, and the signaled thread (if any) runs immediately. In a Mesa monitor, the signaling thread continues execution, and the signaled thread is simply made eligible for execution. [Operationally this boils down to "if (condition)…" vs. "while (condition) …"]

12. *[4 points]*
Here is Dekker's Algorithm to provide critical sections where there are exactly two processes, named i and j. It works. (The code is shown for process i.)

```
bool flag[2];
int turn = i;
do {
        flag[i] = true;
        while ( flag[j] ) {
                if ( turn == j ) {
                        flag[i] = false;                /* (A) */
                        while ( turn == j )             /* (B) */
                                /* nothing */ ;
                        flag[i] = true;                 /* (A) */
                }
```

```
}
… critical section …
turn = j;                                   /* (C) */
flag[i] = false;                            /* (C) */
```

(A) What goes wrong if the statements labeled (A) were omitted?

If both threads arrive at this code at about the same time, both will set their flag[.] entries to true. Both will then spin forever in the outer-most while loop. (I.e., neither will ever make it to the critical section.)

(B) Does anything go wrong if the statements labeled (C) were re-ordered? If so, what? If not, briefly justify that answer.

No, nothing goes wrong. Turning flag[.] off allows the other thread to enter the critical section; as this one has exited, that is fine. turn is used just to break a tie, when both threads arrive to this code at about the same time. Before this thread can get back to the head of this code, it must reset turn, even if the two statements are reversed. Thus, all of the conditions of question 10a are preserved.

13. *[5 points]*
Suppose you have an incorrect implementation of a singly linked list of foo's, with a global head pointer. Here is part of the implementation:

```
struct foo {
        struct foo * next;
        …
}
struct foo *head = NULL;

void enqueue(struct foo *newFoo) {
        newFoo->next = head;
        head = newFoo;
}
```

Which of the following can occur if you run this code? (Circle the asterisks for the ones that can occur.)

* An item appears twice on the list, although it has been enqueued only once.
No – the only way for the item to show up more than once in a traversal of the list is if there is a loop in the list.
* An item successfully enqueued on the list disappears before being dequeued.
Yes – if a thread-switch occurs after fetching head (in enqueue()), and then an item is enqueued, and then we complete execution of that first enqueue(), the new item will

be lost.

* A list with more than one foo in it contains a cycle.

Yes – X fetches head, which points to Y, and then is preempted. Y is dequeued. X is dispatched and completes, setting head to point to X. Now Y is enqueued, so points at X. The queue is now head -> Y -> X -> Y…

* A list with just one foo in it contains a cycle.

No – the only way for a foo to point at itself is if head points at it when enqueue() is called. The foo shouldn't be being enqueued at a time when it is already on the queue.

* The list contains one or more foo's that have already been dequeued.

yes – same mechanism as the above: fetch head, then a context switch occurs, then a head of list foo is dequeued, then the enqueue completes.

14. *[4 points]*

Suppose a system contains two resources, X and Y, and three processes, A, B, and C, and that the Banker's Algorithm is used to avoid deadlock.
There are 3 units of X and 5 units of Y in total. The maximum resource requirements of all three processes have been declared to be 2 X and 4 Y.
Process A currently holds one X and no Y, process B holds no X and one Y, and process C holds nothing.

(A) Give an example of a next request for a single resource by one of the processes that **will** be granted.

As it turns out, any request is okay – e.g., B requests one Y.

(b) Is there any next request for a single resource by any of the processes that **will not** be granted? If so, give an example. If not, explain briefly why not.

All single resource requests will be granted. There are enough available resources of all types to provide the maximum possible request by any process, so it is always possible to reduce the graph by (simulating) completing the process that makes the next request, for instance.

15. *[2 points]*

Briefly explain why assigning a unique integer index to each lock, and ensuring that threads obtain locks only in decreasing numerical order, prevents deadlock.

This prevents circular wait, because no "waiting for" path can ever lead from a lower numbered lock (already held by some process) to a higher numbered one (that the process is waiting for). Since all paths can only go from higher to lower numbered locks, there can be no cycles.

16. *[2 points]*

Most thread packages provide no support for detecting deadlock. Why not?

Too expensive – you need to keep track of all outstanding allocations, for one thing. For another, you have to do an expensive analysis of the graph to determine whether or not deadlock exists. (If it weren't expensive, you'd provide this support, even if you couldn't sensibly break deadlock when it was detected – it would still be a useful debugging aid to have the thread package notice and explain what the deadlock was.)

Bonus Question *[2 points]*

In Section 6.3 the book presents Peterson's Solution to the critical section problem when there are only two processes. The text says "Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures." It then proceeds to prove Peterson's algorithm correct.

Explain what is going on.

Peterson's solution assumes that read and writes are atomic, and that code executed by the hardware follows the sequential order implied by the C-level specification. There are two reasons modern hardware might not do this: caching (might result in out of date values being provided, in some multiprocessor systems), and out-of-order execution (might reorder loads/stores in ways that could potentially break the algorithm).