CS140
Operating Systems and Systems Programming

Midterm Exam

July 25th, 2006

Total time = 60 minutes, Total Points = 100

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 60 minutes to complete the exam.
- There are 8 questions totaling 100 points. Some questions have multiple parts. Read all parts before answering!
- Please check that you have all 8 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.

1	
2	
3	
4	
5	
6	
7	
8	
Total	

SCPD?	

1. [20] A processor has a 32 bit address space, and combines both segmentation and paging. 4 bits for the segment, 16 for the page, and 12 for the offset. A PTE is 4 bytes. Describe **in detail** what happens in the MMU and OS (use generic terms not the x86 terms) when:

   (a) [10] A user process does a read of address 0xC0DEDBAD and it is in memory.

   Segment 0xC, Page 0x0DED, Offset 0xBAD.

   Lookup the base/bound and where the page table if for segment 0xC. Check them all for validity.
   Look at the 0x0DED'th PTE (0x0DED * 4 bytes into the page table) and see what physical page it is in.
   Add 0xBAD to that physical page number shifted 12 bits to the left.
   Access memory.

   Hopefully all that was in the TLB, so it's all cached in the hardware.

   (b) [5] What different/more/less happens if it is paged out to disk?

   If it is paged out to disk, when you look at the PTE, instead of a physical page number you will see the present bit un-set and a disk offset/pointer to the page on disk, and generate a page fault.

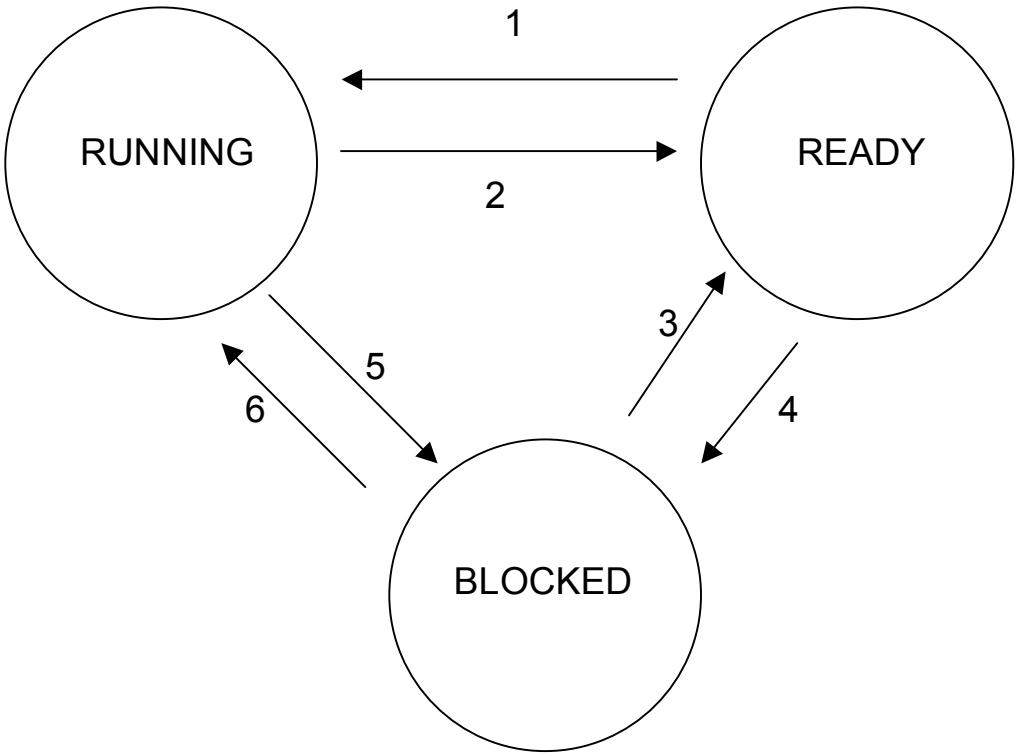
   The OS then requests that block be loaded in, and puts the thread on the blocked list.

   When the page is returned by the I/O device, a physical page is allocated, the block put there, the PTE present bit is set and the physical page number set. The thread is then put on the ready list, and the instruction restarted.

   (c) [5] What different/more/less happens if it is not a valid address?

   At any of the steps in part A) we could find an invalid PTE, an address out of bounds etc. In which case the process would raise a page fault, and since it's not valid, probably be killed.

2. [10] Briefly explaining what conditions cause a thread to move between each of the 3 states, and what causes each arrow. Label it N/A if it doesn't happen.



Arrow 1: Process is scheduled and run by the scheduler.

Arrow 2: Time slice runs out, but process is still wanting to run. Yield().

Arrow 3: I/O completes, or lock is acquired. Woken up by a semaphore or conditional.

Arrow 4: N/A

Arrow 5: Any blocking action. I/O request, lock blocks.

Arrow 6: N/A

3. [10] If you have an OS that's designed for a single CPU, and you want to adapt it to a multi-CPU system, describe the types of changes/additions to the software or hardware you would need to make related to the synchronization primitives (locks, etc…) and why. List three.

   The hardware must handle synchronization of shared memory, like making sure swap is atomic.

   The scheduler must take into account that moving a thread from one processor to another is expensive. When you pick which thread to wake when you unlock/up this can be taken into account.

   On a multi-CPU system, you can use spinlocks much more extensively, because there is another CPU to unlock what you are waiting for.

   …

4. [5] What do semaphores do that condition variables not do, that means you need to be careful how you start the threads?

   Condition variables have no history. So if you signal before you are waiting, it is lost forever. This means you need to be sure you're waiting thread is active before the signaling one.

5. [10] Deadlocks are bad.
   (a) [5] How can you prevent them?

   Order all resources in the system, and only allow ascending allocations.

   Require processes to request all resources at startup.

   (b) [5] How can you detect them?

   Check for cycles in the resources of the blocked processes. All solutions to the problem are variations on methods for doing that.

6. [10] A system you admin is thrashing, and the users are mad.
(a) [5] What are some ways to get into this mess?

Running too many processes for too short a timeslice

Processes that use more memory then you have in their working set.

Bad page replacement algorithm that picks bad pages.

Global page replacement where processes are cannibalizing each other.

Stupid users.

Malicious users.

(b) [5] How would you prevent it from happening in YourOS?

Optimize your scheduler for high-memory loads.

Calculate and use working sets, and use long-term scheduling.

Use local page replacement, and slow moving barriers.

Good security and educated users.

7. [20] Linking

(a) [15] Given the following code, list the data segment, text(code) segment, def's and ref's that the linker would create. Assume each line of code is magically 4 bytes, data is also 4 bytes.

```
extern int printf( char * , ... );
extern float func( float );
float foo;
float bar;

float myfunction()
{
  printf( "hello world" );
  foo = func( 1 );
  printf( "foo 1 = %f\n", foo );
  bar = func( foo );
  return( bar );
}
```

```
Text(code) segment:
(word "call" not required)

0:  call printf( "hello world" );
4:  foo = call func( 1 );
8:  call printf( "foo 1 = %f\n", foo );
12: bar = call func( foo );
16: return( bar );

Data:

0: foo
4: bar
```

```
Defs:

myfunction @0:T
foo @ 0:d
bar @ 4:d

Refs:

foo: @ 4:T, 8:T, 12:T
bar: @ 12:T, 16:T
printf: @ 0:T, 8:T
func: @ 4:T, 12:T
```

(b) [5] If this function was placed in a dynamic shared library, describe how it gets called.

When the library was loaded, an array with the function offsets would be created. When it's called, it's not a normal jump instruction but a jump to array[function #]. The question didn't ask about lazy loading, but if it was lazy, that array would be filled in on the first call to that function.

8. [15] Scheduling

(a) [3] What's good and bad about First Come First Served?

Pro: Trivial.

Con: Convoy effect, avg time to completion is very bad.

(b) [3] What's good and bad about Round Robin?

Pro: Simple to implement, fair.

Con: Avg time to completion for lots of large jobs is worst case.

(c) [3] What's good and bad about STCF?

Pro: Theoretically best.

Con: Future prediction required. Unfair to long jobs.

(d) [3] What's good and bad about Multi-Level Feedback Queueing?

Pro: Adapts to the actual workload. Keeps I/O busy.

Con: Complex, difficult to model, can be slow.

(e) [3] What's good and bad about Lottery scheduling?

Pro: Simple, Easy to do priority donation, very hard to "cheat" or drive system to bad behaviors.

Con: Only fair in theory, jobs may never actually run.