Midterm review

scope -- lectures, assignments, exercises, examples
strategies -- two pass process, my limits, etc.
samples

Some caveats

My exams are difficult.
Always curved.
Professionals (you know who you are) are not included in calculating curves.
Looks like a loooong exercise.
10 answers, 10 points per answer, points do not indicate difficulty.
Roughly 1/3 relatively straightforward questions.
Roughly 1/3 intermediate difficulty questions.
Roughly 1/3 "unfair:)" questions.
2-pass process

Do problems you understand first.
Then do harder ones.
Budget time to check your answers for whether I am being sneaky.
Some strategic hints for open book exams

Index your notes -- with postits -- so you can find things you need quickly. Review the examples -- I will not veer far away from them.

If you do this carefully, you'll find that you don't have to refer to the notes!
I suggest that you outline the answer first, and then write it nicely if you have more time. You will definitely get partial credit for the outline, but no credit for a blank.
File I/O
Bounded buffers and pipes
Lock ordering
Banker's algorithm
Deadlock.
Race conditions - nondeterminism
Livelock - speed
Buddy system- fragmentation
TLB -- performance
Pre IA64: load full segment table from OS to processor.
Post-IA64: skip segment table load, load just the TLB.

Profound performance implications.
Page fault procedure

Page fault is defined as a reference to a page that isn't in the TLB.

When you get a page fault
Interrupt the CPU
Go into kernel mode
Look up page
If it's resident, then load it into the TLB.
If it's not resident (virtual), read it in and then load into TLB.
Resume process execution.

For the TLB to work, need an optimized page table in the OS.
Can I have a deadlock if I execute two sequences of lock(a), lock(b), lock(c), lock(d) concurrently?

How can the following deadlock?

<table>
<thead>
<tr>
<th>lock(a)</th>
<th>lock(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(b)</td>
<td>lock(b)</td>
</tr>
<tr>
<td>lock(c)</td>
<td>lock(a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock(b)</td>
<td>p1 holds a, b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock(c)</td>
<td>p2 holds c</td>
</tr>
<tr>
<td></td>
<td>lock(b)</td>
<td>p2 holds c, blocks on b</td>
</tr>
<tr>
<td>lock(c)</td>
<td></td>
<td>p1 blocks on c</td>
</tr>
</tbody>
</table>
Banker's algorithm

Request grants dibit

P1 2 2 2 0
P2 0x3 0x3 0x0
R1 R2 R1 R2 R1 R2

\[ \text{avail} \]
\[ \frac{2.4}{R1 R2} \]

24
deadlock: blocked and cannot run
livelock: can run but will repeatedly be refused resources.
Why is there a state in which a process has successfully requested resources, requested > granted, and deficit > 0?
The basic problem: pipes are block buffered. Thus write is only called when a block is ready. If we write something small to a pipe with fprintf, it will actually stay in application memory. fflush writes it.

```
#define READ 0
#define WRITE 1

int p1[2], p2[2];
pipe(p1);
pipe(p2);
if (fork()) {
    FILE *write = fdopen(p1[WRITE], "w");
    FILE *read = fdopen(p2[READ], "r");
```
char buf[1024];
fprintf(write, "hi\n");
fscanf(read, "%s", buf);
} else {
    FILE *write = fdopen(p2[WRITE], "w");
    FILE *read = fdopen(p1[READ], "r");
    char buf[1024];
    fscanf(read, "%s", buf);
    fprintf(write, "ho\n");
}
Buddy system

The classic time/space tradeoff.

\(O(1)\) time

Enormous space (internal fragmentation)

up to \(2^{(n-1)}\) bytes wasted when you request \(2^n\)