Exam review notes (2nd half of lecture 6)

The exam review starts at 24:30 on the video of the second half of the lecture. The blackboard isn't very readable in the video; this is my transcription along with some additional comments.

The first half of the class (not counting virtualization) can be divided into three parts: (a) memory maps and context switching, (b) synchronization, and (c) virtual memory.

Memory maps and context switching

Memory maps – different ways of organizing code, data, heap, stack, and operating system.

Mechanisms for connecting programs to the operating system –

- library interface (just link the OS with the program)
- separate OS call table
- interrupt instruction with hardware-interpreted vector table

Separation of user and supervisor mode

- interrupt with user-to-supervisor mode transition, return restores user mode

Simple memory-mapped I/O devices

Device interrupts –

- hardware forces a function call to an interrupt handler when a device needs attention; interrupt handler does something (e.g. reads input) and then returns
- timer interrupt – forces a call to the timer interrupt routine at periodic intervals

Stack frames – stack is used for

- stack is used for arguments (push them on the stack before CALL instruction)
- return address (pushed on stack by CALL instruction)
- local variables (stack pointer is incremented by sizeof(locals) at beginning of function to reserve space, and decremented by the same amount before returning

Synchronization

Mutexes

Monitors – a monitor is a user-defined object (i.e. class) with:

- fields (e.g. int, bool, etc.)
- conditions – special variables that can only be used for wait(), signal(), broadcast()
- methods

A monitor has an implicit mutex – only one thread can be “in” the monitor at once.

Thread A enters the monitor by either:

- calling a method, or
- returning from wait()

In either case, if another thread B is in the monitor, thread A blocks until B leaves the monitor.

You leave the monitor by:

- returning from a method, or
- invoking wait(<condition>)
Classic synchronization problems:

- bounded buffer
- rendezvous

`signal(<condition>)` causes one thread currently waiting on the condition to become runnable, but it does not start immediately because the call to 'signal' doesn't cause a thread to leave the monitor.

`broadcast(<condition>)` causes all threads waiting on a condition to become runnable, although they will have to return from `wait()` one at a time.

If no thread is waiting, `signal()` and `broadcast()` do nothing.

Things to remember for writing concurrent logic:

- the values of shared variables can change while you wait. In fact, they should, since typically you're doing things of the form:
  
  ```
  if (not ready yet)
  wait
  ```

  and when you wake up, variables have been changed so that 'not ready yet' has become false.

  This means that if you're accessing any variables that change with each arriving thread (e.g. a sequence number, or assigning first/second, etc.) you should sample the value and save it in a local variable if you need to refer to it after the `wait()` returns.

- For non-threaded code, you only need to think about a single thread going through your function. For a monitor method you need to be able to think about multiple threads proceeding through – only one thread moves at a time, but whenever it stops at a `wait()` another thread can start moving through.

- If you're trying to synchronize e.g. 2 threads (for instance, the 2-thread rendezvous discussed in a previous lecture), you also have to think about what happens if threads 3 and 4 enter the monitor before threads 1 and 2 have returned. (e.g. if thread 2 signals thread 1, then thread 3 might enter before thread 1 returns from 'wait')

- [based on questions I've received on homework 2] No wishful thinking. If you put a `wait()` in your code for a specific purpose, you have to make sure that the program logic directs the right threads to that `wait()` statement.

Virtual memory

We discussed four aspects of virtual memory and memory management:

- memory allocators and fragmentation (we won't cover allocators on the exam)
- base & bounds address translation (not covered on exam)
- paged address translation
- page fault handling
- page replacement algorithms
For address translation we assume that all the world is a 32-bit Pentium – i.e. 32-bit virtual and physical addresses with 4K \( (2^{12}) \) byte pages and 2-level page tables. Each page is identified by the top 20 address bits – i.e. in hexadecimal, page 00001 refers to the addresses:

00001000 … 00001FFF

This gives us the following address layouts:

<table>
<thead>
<tr>
<th>Virtual page number (20 bits)</th>
<th>Offset in page (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number (20 bits)</td>
<td>Offset in page (12 bits)</td>
</tr>
</tbody>
</table>

The process of address translation involves translating the virtual page number (VPN) to a physical page number (PPN) – the offset remains unchanged. For this we divide the virtual page number into two 10-bit parts:

<table>
<thead>
<tr>
<th>Top 10 bits</th>
<th>Next 10 bits</th>
<th>Offset (12 bits)</th>
</tr>
</thead>
</table>

We create a page table, which is a tree:

Each box in the tree (i.e. page directory, page tables, and pages) is a physical page, and can thus be specified by its physical page number (20 bits) instead of the full physical address (32 bits).

The top 10 bits are used to index into the page directory, which has 1024 entries with index 0…1023. Since the entries are 4 bytes each, if an entry has index \( i \) (e.g. 2) then its offset in bytes from the beginning of the page is \( i \times 4 \) (e.g. 8). So if the page directory is located in physical page 00003, the hardware can retrieve the entry with index 2 by reading 4 bytes at physical address 00003008.

The entry in the page directory contains two fields we are concerned with: a page number (20 bits) pointing to a page table, and a present bit, indicating whether the entry is valid (present) or not. The next 10 bits are used to index into a page table to retrieve another 4-byte entry, and the 20-bit page number found in that entry points to the actual physical page where the data is found.

Page translation doesn't translate each 10-bit section of the virtual address separately; instead 10 bits is consumed by indexing into the page directory, and the second 10 bits is consumed indexing into the page table. All 20 bits of the final physical page number are found in the 2\(^{nd}\) entry, in the page table.
Page fault handling

A page fault can be handled by one of the following mechanisms:

- segmentation fault (i.e. kill the process)
- demand allocate (allocate a page, zero it, plug it into the page table, return)
- demand paging (allocate a page, read its contents from disk, plug into the page table, return)
- copy-on-write (allocate a page, copy it, replace old read-only mapping with new mapping to read/write page, return)

Page replacement algorithms

We talked about three page replacement algorithms:

- FIFO (first-in first-out) – replace the page that was brought into memory first.
- LRU (least recently used) – replace the page that was least recently used. If you simulate this on paper, it helps to keep the list of pages in order – every time you get a hit, move the corresponding page to the top of the list, and every time you miss, remove the bottom page and put the new one at the top of the list.
- OPT (optimal) – look forward into the future, and replace the page which is not used for the longest time. Obviously impossible in practice.