Chapter 2 – Dietel – Lesson Plan for Abrams' Section

Items to Highlight

- **Processor**:
  - Slide 5b in http://courses.cs.vt.edu/~cs3204/spring2004/Notes/OS3e_02.pdf: Contains registers, ALU, instruction fetch/decode, …
  - Register transfer notation (see slides 15 and 20 from Dave Patterson notes: http://people.cs.uct.ac.za/~gaz/teach/arch/lec10-muticycle.ppt).
  - Microinstructions used to create each instruction of instruction set
  - Decisions made by processor architect:
    1. Addressing means
    2. Protection (e.g., for memory)
  - Is processor cycle speed equal to memory cycle time (2GHz vs. 333MHz SDRAM)? NO!
    3. So what does processor do while waiting for memory read to complete? Wait state OR parallel instruction launch (Pentium 4 option – hyperthreading)
    4. Caching (L1) allows R/W at processor speed, and L2 is a bit slower.
    5. Why can't RAM function at cache speed? Cache is on-chip. Electrical signals traveling faster require longer cycle time. Physical limits due to memory taking space vs. desired cycle time.
    6. Does your computer have enough RAM for all the running processes? How would you check? [HW EXERCISE!] Use ps or Windows task manager. SHOW TASK MANAGER! ALSO SHOW KERNEL TIME VIA OPTION.
    7. What if program cannot fit into RAM? Use paging/segmentation – covered later in course. Use multi-level cache (even 3rd level device). SHOW SLIDE 7A.
- **Addressing**
  - What unit is addressed in cache? (Byte – but unit transferred to/from Main Memory is a cache *line*) Main memory? (Byte or Word) Disk? (Blocks).
  - What contributes to latency in main memory? (Electrical signals, refresh cycle for dynamic RAM) What contributes to disk? (Move head, rotate, transfer time for block)
- **Bus**
  - What is a front side bus? (Processor-Memory connection)
  - What is implied for hardware if a processor is 32-bit or 64-bit? (address field width in instructions and registers; number address lines for memory, number data lines because word size is 32 or 64 bits, ALU width for arithmetic and logical operations)
- **DMA**

- How does programmed I/O work? Where could it still be used today? (simple embedded processor designs where cost is minimized)
- How did I/O Channels and interrupts improve computer performance over programmed I/O?
- How does multiprogramming work (in terms of concurrent execution of instruction and disk transfers)?

8. DMA (SHOW SLIDE 9B – not a great picture)
    - Processor instructions used to load DMA registers (start, length, R/W, other control bits)
    - Processor than executes other process while DMA controller independently R/W disk.
    - What happens when DMA transfer done? Interrupt raised. Processor jumps to interrupt vector address to run handler. Handler changes process state from wait to ready. Execution of current process resumes. Later in day process blocked on I/O is scheduled, and continues using R/W data.
    - ERROR IN FIGURE 2.4: Disk does NOT send interrupt to processor. Disk notifies I/O Controller when done, and I/O Controller interrupts processor (as text says) – literally the disk is not connected to the interrupt bus!

- What resources are shared? Memory with CPU and DMA device – memory can be multiported or CPU/DMA can hang off same bus.
- What happens when disk transfer is concurrent with processor execution?

9. I/O cycle steals from processor. (Why not other way around?)

- **How does computer stop user process from executing privileged instruction?**
    - Processor starts operation in privileged mode.
    - OS kernel code runs initially.
    - When OS code is ready to transfer to user code, OS *first* switches processor to non-privileged mode.
    - User code runs.
    - Either timer interrupt occurs or user code requests privileged operation by calling OS API, which in turn raises software interrupt. Software interrupt causes program counter to jump to interrupt vector table, then to interrupt handler. CPU state changes to privileged mode, but that is OK because interrupt handlers were loaded long ago by user and cannot be overwritten (because they are in privileged memory locations).
    - Repeat cycle.
- **Memory (section 2.4)**
    - **How does computer restrict user program to certain memory areas?**

10. Privileged instructions to load memory bounds registers, then load program, then jump to program to execute program.

    - **Why does a computer today need virtual in addition to physical memory addresses?**
        - An OS today will always multiprogram many programs as processes. The programs must have distinct memory areas in RAM. Therefore there must be two sets of addresses – the virtual

ones emitted during compilation, and the physical ones when running as a process.
        - Even if you didn't multiprogram, you still potentially need two sets of addresses. Suppose you want to run a very large program (or one that accesses very large data structures) on may different computers with varying amounts of RAM – possible too little RAM to hold the address space of the program.
    - **Is VA->PA translation done in HW, SW, or both?**
        - MUST have HW support, since addr translation required on EVERY instruction fetch and execute. However, some can be done in SW – such as allocation of memory areas, handling of page faults (access to VA that is not currently in RAM), interrupt handlers, etc.
- **Timer vs. clock**
    - Timer = processor sharing
    - Clock = time of day
- **How does OS get into memory and processor start executing it? (2.4.3)**
    - BIOS (Basic Input Output System) in ROM.
    - Processor starts by executing code at known address – jumps to BIOS.
    - BIOS loads bootstrap loader in OS from disk area known as *boot sector*
    - SEE SLIDE 14A.
    - Bootstrap loader loads OS kernel.
    - Kernel initiates user processes, adding the rest of the OS, resulting in GUI starting (with login request) or command login.
    - **HOW does diskless workstation work?**
- **Plug/Play (2.4.4)**
    - Note: See slides on plug & play at www.cpe.gmu.edu/courses/it212/Lectures/INFT_212_Intro/sld054.htm
    - ***What is the difference between device drivers and Plug & Play***? (What does Plug & Play add to device drivers?)
    - Consider an OS w/ and w/o Plug and play. ***What does a Plug/play OS do when you plug in an external CD player that the other does not (so you don't have to reboot to use CD player)?***
        - Device identifies itself to OS, along with resources & services & DEVICE DRIVER needed
        - OS CONFIGURES device (allocate memory, handle interrupts)
- **Caching/Buffering**
    - ***List two places where caching is used in today's PCs***
        - 1. In processor chip (L1, L2 caches)
        - 2. In file system (main memory cache of parts of files either on local disk or on network-accessible file server)
    - ***What are buffers & why needed?***
        - Asynchronous operation due to speed mismatches (example: when you do read() in POSIX to a network, a buffer is returned which holds a group of bytes read from a network. Network and processor bus work at different and asynchronous speeds.)

- Passing data between kernel & user processes (across privileged boundary)
- **Is a buffer a resource?** Yes – often amount of buffer space is an OS parameter
- **What is historical reason for SPOOLing? (simult. peripheral ops online)** Put disk between processor and slow I/O device (e.g., printer). This is really another type of buffer to handle speed mismatch of processor and printer.(STORY on IBM salesman with early spoolers)

- **APIs**
  - Layer on top of user/kernel boundary - SEE FIG 19b/Slide 38
  - Can be object-oriented (e.g., Microsoft Foundation Classes)
  - Analogous to protocol
  - Allows division of software development among teams
  - POSIX, Windows API
- **Compilation (2.8.1)**
  - Review machine code, assembly code, HLL, intermediate code (bytecode).
  - **What advantage does Java bytecode have over traditional compilation to machine code?** Java bytecode is portable.
  - Review declarative vs. imperative (OO, procedural, rule-based, …)
  - Compiler vs. interpreter
    11. Compiler steps (PAGE 21A/Slide 41):
        - Lexical analysis
        - Parsing
        - Intermediate code generator (front end output)
        - Optimizer
        - Code code generator (back end output)
    12. STORY: Michael Tiemann, 1980's, gnu compiler, sold for $600M to Red Hat?
  - ***When source is compiled, what addresses are used in machine code that compiler emits? What happens when you compile functions or objects separately?*** (They all use the same starting address.)
  - Output of compiler is OBJECT MODULE.
    See Fig. 2.9/Slide 43.
    Note symbol table, etc.

- **Linking (2.8.2)**
  - ***What does a linker do?*** It resolves EXTERNAL REFS generated by separate compilation of modules of a single program.
    (Static: it modifies addresses to lay out the code in memory. Dynamic: upon reference to function not in memory, locate function's object code in disk file, allocate memory, and load into memory. Perform magic to resolve external refs [e.g., make CALL instruction's target address map to starting address where function was loaded into memory].)
  - SHOW Fig. 2.10/Slide 44: combining separate object modules into one module.

        - Linker must do SYMBOL RESOUTION and RELOCATE instructions and data to different virtual address ranges, as shown in bottom of Fig. 2.10 & 2.11/Slide 44 & 45.
        - ***What types of addresses require symbol resolution?***
            - CALL? (yes)
            - JUMP? (no)
            - LOAD/STORE? (maybe – if memory location is external ref)
        - ***What types of addresses must be relocated?***
            - CALL? (yes)
            - JUMP? (yes)
            - LOAD/STORE? (yes)
    - **Could a computer JUST use static linking?**
        - YES, in the old days. But the problem is shared library code would be redundantly stored in every binary on disk. Dynamic linking allows each program that is executed to dynamically link to the shared library. Also, if library is reloaded, with dynamic linking the code using module does not have to be statically relinked.
- **What does a loader do?** (2.8.3)
    - Puts load module into main memory. See Fig. 2.12/SLIDE 47
    - In PRE_VM days:
      In pre-VM days, loaders were more important.
      How can the hardware do dynamic address translation? (Note: base+offset addressing by processor allows code to be relocatable. Later, we'll see how a paging or segmented memory allows unpopular parts of code to be shuttled between main memory and disk, and then back to a different area of main memory.)
    - With VM, only the initial page is put in memory, and page faults load other pages as needed. Furthermore, each VA space for a program could start at same starting address assumed by compiler (e.g., 0 or 1000), so no relocation is needed by loader.
- **REVIEW** of comile/link/load
    - Figure 2.13/Slide 48
- **FIRMWARE (2.10)**
    - **Microprogramming:** level of programming below today's programming. Think of the register transfer language we saw from Patterson slides on pipelining (http://people.cs.uct.ac.za/~gaz/teach/arch/lec10-muticycle.ppt).
        - Not really used in CPU design today.
        - Still used for slower embedded processors, like disks
    - **Firmware** is executable instructions, such as microprogramming. Often can be replaced (e.g., FLASH memory in network switch or car)