Chapter 4 – Deitel – Lesson Plan for Abrams' Section

Items to Highlight

NOTE: GTA Corban Rivera went over Deitel slides 1-18. Notes below cover subsequent slides.

- Tell class about the location of these notes on the Web.
- Section 4.3: Start class with a thought problem on threads
  - Consider a web server.
  - What performance measures do you as a user care about from a Web server? Latency and transfer rate. What does server admin care about? Throughput.
  - Suppose you and I want to retrieve web pages in parallel. I ask for a 1Gbyte doc. You ask for a 1k page. But your request reaches server after mine. Will you be happy? (No way! Your small file waits for my big file!)
  - How can web server use threads to essentially achieve Shortest Job First (SJF) scheduling? (Concurrent processes/threads allow HTTP request that arrives later to be processed concurrently with HTTP request that arrives earlier.)
  - What does the outline of a web server's code look like (given fork/exec from earlier chapters)?
    - loop forever {
      - listen for request from browser
      - if (fork()>0) ServiceRequest()
    - } exit()
    - ServerRequest() { parse HTTP request; retrieve document via read(); send doc to client(); exit(); }
  - Why do threads help improve above code? (Performance!)
  - Can you do a denial of service attack on my server code above? (Yes – just launch so many HTTP requests in parallel that my server chews up all memory for forked processes.) How can you prevent this? (Set max # forked processes/threads at any time.)
  - Can we use threads in place of forked processes? (Yes – can share memory OK, and can inherit socket descriptors from parent.) What is benefit of threads (less latency to start, plus less memory area per thread meaning more threads per given amount of server RAM).
  - Why does a web server need a pool of threads? (Reduce latency to process each HTTP request due to thread creation overhead)
  - Do you think the web server has a limit on the max number of concurrent HTTP requests it will try to handle at any time? (Yes – max # threads it will spawn).
  - Why does a Web server need more RAM than your average workstation? Lots of memory needed for thread execution contexts, and file buffering, and socket descriptors.

- How does max # threads parameter (discussed above) and amount of RAM relate? If admin sets max # threads too high, then machine will start swapping, which kills web server performance (latency and throughput).
- If we run our server on a dual or quad processor machine, could we live with a user-level thread package or must we have a kernel-level package? Would need kernel-level; in user-level the OS only sees one process (in which all threads live), so that process can only use 1 processor, leaving the other processors idle. Hybrid threading model only works if you bind each thread to its own activation record.
- If we wrote our Web server in Java, would we get user or kernel level threads? (See http://java.sun.com/docs/hotspot/threads/threads.html for info on Java threads implemented on Solaris.)
  - Green threads are user-level
  - Native threads are kernel-level

- Review: SHOW SLIDE 9 (thread life cycle).
  - Give some practical uses of a sleep() thread API call.
    - Animation – to control speed.
    - Performance monitor – to control how often the monitor samples system state.
    - Calendar like Outlook pops up reminders of your appts, and you ask it to wait for an hour to notify you again.
- Section 4.7.1: Signals
  - NOTE: Did not get time to cover Unix signals last week as part of Chapter 3. So let's start with UNIX signals
    - Unix has a bunch of signals:

```
[cs3204@ap1 cs3204]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      35) SIGRTMIN    36) SIGRTMIN+1
37) SIGRTMIN+2  38) SIGRTMIN+3  39) SIGRTMIN+4  40) SIGRTMIN+5
41) SIGRTMIN+6  42) SIGRTMIN+7  43) SIGRTMIN+8  44) SIGRTMIN+9
45) SIGRTMIN+10 46) SIGRTMIN+11 47) SIGRTMIN+12 48) SIGRTMIN+13
49) SIGRTMIN+14 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8
57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5  60)
```

SIGRTMAX-4
61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1 64) SIGRTMAX

- API allows a program to specify action when signal is received via signal() system call:
  - #include <signal.h>
    typedef void (*sighandler_t)(int);
    sighandler_t signal(int signum, sighandler_t handler);
  - Returns value from sighandler_t, or SIG_ERR on error.
- Let's look at how to handle ^C in a shell (SIGINT)…
  - To ignore a signal:
    signal(SIGINT, SIGN_IGN);
  - To allow default behavior:
    signal(SIGINT, SIG_DFL);
  - To call signal handler you wrote, called *MySigHandler()*:
    signal(SIGINT, MySigHandler);
- raise(SIGINT) sends signal to CURRENT process.
- kill(pid,SIGINT) sends signal to named process id.
- Signals can be
  - delivered immediately or pending a process's return to RUNNING state
  - synchronous (e.g., running process divides by zero and signal immediately interrupts execution) or asynchronous (e.g., blocked process receives signal that pending I/O operation has completed)
- Example (that prints "Hello world" every time you hit ^C):

```
#include <stdio.h>
#include <signal.h>

void hello() {
  signal(SIGINT,hello);    // needed in some UNIX systems (e.g., linux)
  printf("Hello world\n");
}

main() {
  signal(SIGINT,hello);
  while (1) { ; }
}
```

- Now let's look at thread signals
  - Does API expose thread ids? Are thread ids known across programs?
  - If user-level threads, then OS delivers signal to user process running thread. Up to implementation on what to do (e.g., notify all threads) via its own signal handlers and masks.

        - If kernel-level threads, in POSIX processes specify signal recipient by a process id, not a thread id. Each thread can then specify mask as to whether it wants to receive signal. SEE SLIDE 20.
        - What if many-to-many threading model is used, and signal goes to multithreaded process, but running thread masks signal? In Solaris 7 uses an extra thread to monitor signals: Asynch. Signal LWP (ASLWP). Delivers signal to appropriate thread, even if destination thread is not running.
- Section 4.7.2: How can a thread terminate?
    - 1. Thread completes execution
    - 2. Thread experiences fatal exception (e.g., divide by zero)
    - 3. Thread receives signal to terminate.
  - What does the book mean by "A thread that modifies a value in its process's shared address space may leave data in an inconsistent state if terminated properly" on page 165?
    - Problem with book order of topics: has not discussed use in algorithms of shared memory
    - Imagine a program that adds two matrices. It uses many threads to compute element-wise sum – let's say 8 threads for an 8 processor computer. Let's say program is organized as manager thread that hands out work, and worker threads that each perform computation on one element. If a worker dies, it may have modified a data structure without finishing, and another worker (assigned to do the same element later) may find unstable quantities in memory.
- 4.8 POSIX thread highlights:
  - Portable thread spec – no implementation information; can be done at user or kernel-level
  - Called PThreads
  - Signals are masked by individual threads as described earlier
  - API allows CANCELLATION of a target thread
    - Target chooses asynchronous (immediate) or deferred cancellation or DISABLE cancellation.
- 4.9 LINUX Threads
  - Do NON-PORTABLE clone() not fork() to create a thread
  - OS scheduler treats threads and processes equally; permits high scalability
  - #include <sched.h>
    int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
    _syscall2(int, clone, int, flags, void *, child_stack)
  - Returns PID of child thread
  - Unlike fork, shares
    - memory space
    - table of file descriptor
    - table of signal handlers.
  - "fn" is the function that the clone executes (unlike fork()).
  - "arg" is the arguments passed to function "fn".

- Parent must explicitly chose what memory area child uses as a stack (child_stack)
- Use of "flags"
  - Low bytes of "flags" states what signal parent receives when child dies (or no signal if flags not set).
  - Other bytes say if child shares file descriptors, working director, file descriptor table, whether child is in same thread group as parent, …

- 4.10 Windows XP Threads
  - Allows private data for a thread (Thread Local Storage – TLS)
  - A thread can contain one or more fibers. Thread schedules fibers, Scheduler schedules threads.
  - Fiber can have private data (Fiber Local Storage – FLS)
  - Fibers work as coroutines – they run until they voluntarily allow another fiber in same thread to run. But the entire thread can be preempted (e.g., by expiration of quantum to use processor)
  - XM provides kernel-mode threads to execute functions on behalf of user threads
    - One kernel thread can sleep, do read for user thread, sleep, do read for another user thread, etc., with less cost of creating new kernel thread for each request.
- 4.11 Java Threads
  - Look at class Thread API: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html
  - Look at class Runnable: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Runnable.html
    The "run()" method is almost like a main() method in a C program that you write!
  - How to use #1:
    - Extend class Thread and implement run() method.
    - Create instance of thread and invoke its run() method.

```
class PrimeThread extends Thread {
        long minPrime;
        PrimeThread(long minPrime) {
           this.minPrime = minPrime;
        }

        public void run() {
           // compute primes larger than minPrime
            . . .
        }
    }
…
PrimeThread p = new PrimeThread(143);
p.start();
```

- How to use #2:
  - Create any old class.
  - Have that class implement interface java.lang.Runnable.
  - Create an instance of class java.lang.Thread whose constructor takes a Runnable object.
  - Do Thread::start() to start a thread for Runnable class.

```
class PrimeRun implements Runnable {
        long minPrime;
        PrimeRun(long minPrime) {
           this.minPrime = minPrime;
        }

        public void run() {
           // compute primes larger than minPrime
            . . .
        }
    }

…
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

- To terminate the thread, let the run() method you defined return!
- The JVM (Java Virtual Machine) continues to run until one of these happens:
  - Some Java method calls exit(), and security manager OK's it.
    - WHEN is calling exit() forbidden? Applet!
  - When run() method of each non-daemon thread returns.
- LOOK AT SLIDES 29-32.
- EXPERT QUESTION:
  Why is method Thread.sleep() static? After all, Thread.start() is not.
  - Answer: sleep can be called by any method, anywhere in program, as Thread.sleep(), whereas code can only call start() if it has a reference to the thread. In other words, it makes sense to let any piece of code cause a thread to sleep, while only a special piece of code should start a thread.