Slide 1

Administrivia

- Reminder: Homework 1 due today. Submit written problems in hardcopy form by 5pm, programming problem by e-mail by 11:59pm. (Programming problem accepted without penalty through 11:59pm tomorrow. A bit more about it today.)
- Reminders/requests about homework:

  All homework is considered pledged work. Write "pledged" on hardcopy work, and include it in comments for programming assignments.

  For work submitted by e-mail, please do include the name or number of the course in the subject line of your message, plus something about which assignment it is, to help me get it into the correct folder for grading.

Slide 2

Minute Essay From Last Time

- "CPU" is kind of ambiguous these days. Usually I use it to mean "processing element" (processor/core).
- Maximum of one running process per CPU (in this sense). Minimum number depends on whether there are ready processes. (And if you're pedantic, maybe there is some small interval during a context switch in which there is no process running on the CPU involved.)

Slide 3

Homework 1 Programming Problem — Hints

- The idea is to write a very simple shell based on the sort-of-pseudocode in the textbook.
- To do this, you have to solve a couple of problems:
  - Figure out how to use system-call library functions `fork` and `execve`. Their `man` pages are a good source of details.
  - Deal with string processing in C (or C++). `fgets` is good for reading a whole line of input, or just read input a character at a time and look for newlines.

Slide 4

C Programming Advice

- I strongly recommend always compiling with flags to get extra warnings. There are lots of them, but you can get a lot of mileage just from `-Wall`. Add `-pedantic` to flag nonstandard usage.
- If you want to write "new" C (including C++-style comments), add `-std=c99`.
- If typing all of these gets tedious, consider using a simple makefile. Create a file called `Makefile` containing the following:

  ```
  CFLAGS = -Wall ....
  ```

  and then compile `hello.c` to `hello` by typing `make hello`.

Slide 5

Processes Versus Threads

- So far I've used "process" in an abstract/general way.
- In typical implementations, though, "process" is more specific — something that has its own address space, list of open files, etc. Often these are called "heavyweight processes".
  - Advantages — such processes don't interfere with each other.
  - Disadvantages — they can't share data, switching between them is expensive ("a lot of state" to save/restore).
- For some applications, might be nice to have something that implements the abstract process idea but allows sharing data and faster context switching — "threads".

Slide 6

Threads

- So, threads are another way to implement the process abstraction.
- Typically, a thread is "owned" by a (heavyweight) process, and all threads owned by a process share some of its state — address space, list of open files.
- However, each thread has a "virtual CPU" (a distinct copy of registers, including program counter).
- Implementation involves data structures similar to process table.
- Advantages / disadvantages (compared to processes)?

Slide 7

Threads, Continued

- Advantages: threads can share data (same address space), switching from thread to thread is fairly fast.
- Disadvantages: sharing data has its hazards (more about this later).

Slide 8

Implementing Threads

- Two basic approaches — “in user space” and “in kernel space” Various hybrid schemes also possible.
- Basic idea of “in user space” — operating system thinks it’s managing single-threaded processes, all the work of managing multiple threads happens via library calls within each process.
- Basic idea of “in kernel space” — operating system is involved in managing threads, the work of managing multiple threads happens via system calls (rather than user-level library calls).
- How do they compare?. . .

Slide 9

Implementing Threads, Continued

- Implementing in user space is likely more efficient — fewer system calls.
- Implementing in kernel space avoids some problems, though:
  - If a thread blocks, it may do so in a way that blocks the whole process.
  - Preemptive multitasking is difficult/impossible without help from the kernel, as is using multiple CPUs.

Slide 10

Adding Multithreading

- If you've written multithreaded applications — moving from single-threaded to multithreaded not trivial:
  - Figure out how to split up computation among threads.
  - Coordinate threads' actions (including dealing properly with shared variables).
- Similar problems in adding multithreading to systems-level programs:
  - Deal properly with shared variables (including ones that may be hidden).
  - Deal properly with signals/interrupts.

Slide 11

Implementing Threads, Example — Linux

- Early versions of Linux provided no support for kernel-space threading, but there were libraries for the user-space version.
- More-recent kernels provide support, but in an interesting way — threads in some ways are just processes with with some different flags allowing them to share memory, etc.

  Adding support for threads complicates process creation — the basic mechanism (`fork`) duplicates an existing process, and if that process is multithreaded, things can be interesting. Some details in chapter 10, or read the POSIX standard for `fork`.

Slide 12

Minute Essay

- Tell me about your experience (if any!) with writing programs that involve concurrency — multithreaded, message-passing, communicating over sockets, etc.