Slide 1

Administrivia

- (Quiz 1 solution in hardcopy.)

Slide 2

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 easily 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 3

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 4

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 5

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 6

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 7

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 8

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 9

Interprocess Communication

- Processes almost always need to interact with other processes:
  - "Ordering constraints" – e.g., process B uses as input some data produced by process A.
  - Use of shared resources — files, shared memory locations, etc.
- Use of shared resources can lead to "race conditions" — output depends on details of interleaving.
- Processes must communicate to avoid race conditions and otherwise synchronize.
- "Classical IPC problems" — simplified versions of things you often want to do.

Slide 10

Mutual Exclusion Problem

- In many situations, we want only one process at a time to have access to some shared resource.
- Generic/abstract version — multiple processes, each with a "critical region" ("critical section"):

```
while (true) {
    do_cr();        // must be "finite"
    do_non_cr();    // need not be "finite"
}
```

- Goal is to add something to this code such that:
  1. No more than one process at a time can be "in its critical region".
  2. No process not in its critical region can block another process.
  3. No process waits forever to enter its critical region.
  4. No assumptions are made about how many CPUs, their speeds.

Slide 11

Mutual Exclusion Problem, Continued

- We'll look at various solutions (some correct, some not):
  - Using only hardware features always present (some notion of shared variable).
  - Using optional hardware features.
  - Using "synchronization primitives" (abstractions that help solve this and other problems).
- Recall that a correct solution
  - Must work for more than one CPU.
  - Must work even in the face of unpredictable context switches — whatever we're doing, another process can pull the rug out from under us between "atomic operations" (machine instructions).

Slide 12

Sidebar: Atomic Operations

- "Atomic" operation — indivisible, executes without interference from other processes.
- Which of the following are atomic?
  - `x = 1;`
  - `x = x + 1;`
  - `++x;`
  - `if (x == 0) x = 1;`

  (Or does it depend? On what?)

Slide 13

Proposed Solution — Disable Interrupts

- Pseudocode for each process:

```
while (true) {
    disable_interrupts();
    do_cr();
    enable_interrupts();
    do_non_cr();
}
```

- Does it work? reviewing the criteria . . .

Slide 14

Disable Interrupts, Continued

- (1) okay – context switches take place only in response to interrupts, so yes if one CPU.
- (4) not okay — fails if more than one CPU (unless there is a way to disable interrupts on all CPUs).
- Also, user-level programs shouldn't be able to do this (though might be okay for o/s).
- More next time . . .

Slide 15

Minute Essay

- Tell me about your experience (if any!) with writing programs that involve concurrency — multithreaded, message-passing, communicating over sockets, etc.
- Give an example (other than those discussed) of a situation in which you think a solution to the mutual-exclusion problem would be needed.