Slide 1

Administrivia

- None.

Slide 2

Recap — Processes

- Process abstraction — “program running on virtual CPU” (virtual program counter, virtual registers, etc.).
- Apparent concurrency (in almost all respects identical to real concurrency) provided by interleaving / context switches.
- Context switch — switch between virtual CPUs, triggered by interrupts (I/O, error, system call, timer).
- Process can also be a way of grouping together other resources needed by a running program, e.g., “address space”, list of open files.

  These resources may form part of the “context” that must be saved / restored on a context switch.

Slide 3

Interrupt Handling, Revisited, Part 1

- When an interrupt occurs, the hardware:
  - Saves a little about the current process (program counter at least) in an agreed-upon location, e.g., on stack.
  - Transfers control to fixed location — could be always the same location, or one of several depending on the type of interrupt.

Slide 4

Interrupt Handling, Revisited, Part 2

- Code at fixed location is an “interrupt handler”, which:
  - Saves enough of the CPU’s current state to enable later restart, usually in current process’s process control block.
  - “Handles” the interrupt, but minimally — saves data that could be lost (e.g., in device’s input buffer), marks blocked processes ready if appropriate.
  - Invokes scheduler to decide which process to run next.
  - Restores saved CPU state for this next process (from its process control block), causing it to resume execution.

  Other interrupts may be “disabled” during this processing.

Recap — Process States

- Three basic states for processes – running, ready, blocked.
- Some transitions are obvious, others require decision-making (ready to running); for now, assume existence of “scheduler” to make decisions.

Recap — Threads

- Processes versus threads:
  - Process implements “program on virtual CPU” abstraction, has its own group of resources.
  - Thread implements “program on virtual CPU” abstraction, shares group of resources with (some) other threads.
- Threads are in a way “processes within processes”.
- Compare context switching between processes with context switching between threads within process.
- Two basic approaches to implementing threads — “in user space” and “in kernel space”.

Slide 7

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.

Slide 8

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 9

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 1 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 10

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;`

Slide 11

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 . . . No.

Slide 12

Proposed Solution — Simple Lock Variable

- Shared variables:

```
int lock = 0;
```

  Pseudocode for each process:

```
while (true) {
    while (lock != 0);
    lock = 1;
    do_cr();
    lock = 0;
    do_non_cr();
}
```

- Does it work? reviewing the criteria . . . No.

Slide 13

Proposed Solution — Strict Alternation

- Shared variables:

```
int turn = 0;
```

Pseudocode for process p0:

```
while (true) {
    while (turn != 0);
    do_cr();
    turn = 1;
    do_non_cr();
}
```

Pseudocode for process p1:

```
while (true) {
    while (turn != 1);
    do_cr();
    turn = 0;
    do_non_cr();
}
```

- Does it work? reviewing the criteria . . . No.

Slide 14

Minute Essay

- Do you (think you) see why the various solutions to the mutual exclusion problem so far work / don't work?
- Give an example (other than those discussed) of a situation in which you think a solution to this problem would be needed.