Slide 1

Administrivia

- Reminder: Homework 1 written problems due today. Hardcopy, now or in my mailbox by 5pm.
- Reminder: Homework 1 programming problem due Wednesday. Send source code by e-mail, by 11:59pm.

Slide 2

Minute Essay From Last Lecture

- Most people got the intended answer. Most common error was not realizing that it's not really possible to have all 100 processes in the "ready" state — *something* would be running on each CPU.

Slide 3

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 4

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 its own "virtual CPU" (a distinct copy of registers, including program counter).
- Implementation involves data structures similar to process table.
- Advantages / disadvantages (compared to processes)?

Slide 5

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 6

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 7

Implementing Threads, Continued

- Implementing in user space is likely more efficient — fewer system calls, so less overhead.
- 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 8

Adding Multithreading

- As you know 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 — e.g., in implementations of system calls).
  - Deal properly with signals/interrupts.

Slide 9

Sidebar: Signals

- Textbook mentions that one complication of adding support for threads is dealing with “signals”. It may not be clear what those are.
- Signals are a mechanism used by UNIX-family operating systems for one form of interprocess communication, sort of a software equivalent of hardware interrupts.
- Signals can arise from hardware error interrupts (e.g., invalid memory address), from user input (e.g., control-C from console), or from another process (e.g., `kill` command).

Slide 10

Signals, Continued

- O/S delivers signal to process, which can choose to accept it or block it; if it accepts it, it can take a default action (e.g., ignore, or terminate process), or it can provide its own handler.
- If the process contains multiple threads, however . . . Implementation of threads must decide what happens then.

Slide 11

Implementing Threads, Example — Linux

- Early versions of Linux provided no support for kernel-space threading, but there were libraries for user-space threading (e.g., “green threads” for Java).
- More-recent kernels provide support, but in an interesting way — threads in some ways are just processes 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

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 real programs (both applications and O/S) need to do.

Slide 13

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 14

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 mechanisms” (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 15

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 16

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 17

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).

Slide 18

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 . . .

Slide 19

Simple Lock Variable, Continued

- Can easily fail (1).

Slide 20

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 . . .

Slide 21

Strict Alternation, Continued

- (Yes, we're simplifying to only two processes.)
- (1) okay.
- (2) / (3) not okay, since non-critical region need not be finite.

Slide 22

Sidebar: Reasoning about Concurrent Algorithms

- For concurrent algorithms (such as various solutions proposed for mutual exclusion problem), testing is less helpful than for sequential algorithms. (Why?)
- May be helpful, then, to try to think through whether they work. How? Idea of "invariant" may be useful:
  - Loosely speaking — "something about the program that's always true". (If this reminds you of "loop invariants" in CSCI 1323 — good.)
  - Goal is to come up with an invariant that's easy to verify by looking at the code and implies the property you want (here, "no more than one process in its critical region at a time").
  - We will do this quite informally, but it can be done much more formally — mathematical "proof of correctness" of the algorithm.

Slide 23

Sidebar of Sidebar: Reasoning About Loops

- (I probably won't have time to through these slides in much detail in class but will leave them here for anyone interested.)
- Usually want to prove two things: (1) the loop eventually terminates, and (2) it establishes some desired postcondition.
- Proving that it terminates: Define a *metric* that you know decreases by some minimum amount with every trip through the loop, and when it goes below some threshold value, the loop ends.
- Proving that it establishes the postcondition: Use a *loop invariant*.
- (I say "prove" here, since this can be done very rigorously, but in practical situations an informal version is good enough.)

Slide 24

Reasoning About Loops, Continued

- What's a loop invariant? in the context of reasoning about programs, it's a *predicate* (boolean expression using program variables) that
  - is true before the loop starts, and
  - if true before a trip through the loop, with the loop condition true, is also true after the trip through the loop.

  If you can prove that a particular predicate is a loop invariant, then after the loop exits, you know it's still true, and the loop condition is not. With a well-chosen invariant, this is enough to prove useful things.
- (Might be worth noting that compiler writers have a different definition — some computation that can be moved outside the loop.)

Reasoning About Loops, Simple Example

- Loop to compute sum of elements of array `a` of size `n`:

```
i = 0; sum = 0;
while (i != n) {
    sum = sum + a[i];
    i = i + 1;
}
```

  At end, `sum` is sum of elements of `a`.

- Does this work? well, you probably believe it does, but you could prove it using the invariant:

  `sum` is the sum of `a[0]` through `a[i-1]`

Reasoning About Loops, Example

- Euclid's algorithm for computing greatest common divisor of nonnegative integers `a` and `b`:

```
i = a; j = b;
while (j != 0) {
    q = i / j; r = i % j;
    i = j; j = r;
}
```

  At end, `i = gcd(a, b)`.

- Does this work? work through some examples and gain some confidence — or prove using invariant:

  `gcd(i, j) = gcd(a, b)`

  and the math fact `gcd(n, 0) = n`

Slide 27

Strict Alternation, Revisited

- 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();
}
```

- Proposed invariant: "If pn is in its critical region, `turn` has value n, and `turn` is either 0 or 1" (interpreting "in its critical region" as "from just after the `while` to the line after `do_cr()`".

Slide 28

Strict Alternation, Continued

- Proposed invariant again: "If pn is in its critical region, `turn` has value n, and `turn` is either 0 or 1".
- How would this help? would mean that if p0 and p1 are both in their critical regions, `turn` has two different values — impossible. So the first requirement would be met. (Still have to think about the other three.)
- Is it an invariant? check whether true initially and remains true even when one process changes something it mentions. Fairly obvious that it's initially true, so check . . .

Strict Alternation, Continued

Slide 29

- Proposed invariant: "If pn is in its critical region, `turn` has value n, and `turn` is either 0 or 1". True initially. When could it become false?
- When either process enters its critical region. But this happens for pn only when `turn` is n, so invariant stays true (okay).
- When either process leaves its critical region. Also okay.
- When either process changes `turn`. Only happens after process leaves its critical region. So also okay.

Mutual Exclusion, Continued — Preview

Slide 30

- We'll look at one more solution based only on shared variables, but which actually works.
- We'll then look at a solution requiring hardware support.
- We'll then start talking about higher-level "synchronization mechanisms".

Minute Essay

- Did you learn about loop invariants in CSCI 1332 (Discrete Structures)?
- Tell me about your exposure to concurrent programming (multi-threading, message passing, etc. — anything involving multiple threads of control).