Slide 1

Administrivia

- Homework 1 graded. I'm attaching a sample solution to your graded work. I suggest reviewing it even for questions where you got full credit; many people's answers on the first problem seemed a bit confused.
- Homework 2 on the Web. Written problems due next Monday, programming problems due next Wednesday.

Slide 2

Minute Essay From Last Lecture

- No clear consensus on use of invariants to reason about concurrent algorithms. Not in the textbook, so the lectures notes are your best resource. I mean for this to be a useful supplement but not something you have to master to pass the class.

Slide 3

Homework 1 Essays

- Many people forgot to include this! You do lose a point (half a point this time).
- A couple of people mentioned not being able to find answers in the textbook. That's sort of by design — I mean for you to think about how to apply what's in reading and lectures.
- One person said "never really thought about not having an O/S". "Make the students think" is a good goal though not one I always focus on?
- Several people mentioned `strace` problem — interesting, difficult, could have used some in-class guidance.

Slide 4

Semaphores – Review

- A "synchronization mechanism" — way of controlling interaction among processes in a more abstract way than the first few solutions to the mutual exclusion problem.
- Semaphore as ADT:
  - "Value" — non-negative integer.
  - Two operations, "up" and "down", *both atomic*.
- Allows for nice solution for mutual exclusion, also ability to solve more complex problems (e.g., bounded buffer).

Implementing Semaphores

Slide 5

- We want to define:
  - Data structure to represent a semaphore.
  - Functions `up` and `down`.
- `up` and `down` should work the way we said, and we'd like to do as little busy-waiting as possible.

Implementing Semaphores, Continued

Slide 6

- Idea — represent semaphore as integer plus queue of waiting processes (represented as, e.g., process IDs).
- Then how should this work . . .

Slide 7

Implementing Semaphores, Continued

- Variables — integer `value`, queue of process IDs `queue`.

```
down() {                                        up() {
    bool zero;                                      process p = null;
    enter_cr();                                     enter_cr();
    zero = (value == 0);                            if (empty(queue))
    if (!zero)                                          value += 1;
        value -= 1;                                 else
    else                                                p = dequeue(queue);
        enqueue(current_process, queue);            leave_cr();
    leave_cr();                                     if (p != null)
    if (zero)                                           unblock(p);     // mark p runnable
        block();    // mark current process blocked
}
```

- `enter_cr()`, `leave_cr()`? next slide.

Slide 8

Implementing Semaphores, Continued

- Revised functions to enter, leave critical region:

```
enter_cr:
    TSL registerX, lockVar
    compare registerX with 0
    if equal, jump to ok
    invoke scheduler # thread yields to another thread
    jump to enter_cr
ok:
    return


leave_cr:
    store 0 in lock
    return
```

Slide 9

Sidebar: Shared Memory and Synchronization

- Solutions that rely on variables shared among processes assume that assigning a value to a variable actually changes its value in memory (RAM), more or less right away. Fine as a first approximation, but reality may be more complicated, because of various tricks used to deal with relative slowness of accessing memory:

  Optimizing compilers may keep variables' values in registers, only reading/writing memory when necessary to preserve semantics.

  Hardware may include cache, logically between CPU and memory, such that memory read/write goes to cache rather than RAM. Different CPUs' caches may not be in synch.

Slide 10

Sidebar: Shared Memory and Synchronization, Continued

- So, actual implementations need notion of "memory fence" — point at which all apparent reads/writes have actually been done. Some languages provide standard ways to do this; others (e.g., C!) don't. C's `volatile` ("may be changed by something outside this code") helps some but may not be enough.
- Worth noting, however, that many library functions / constructs include these memory fences as part of their APIs (e.g., Java `synchronized` blocks).

Slide 11

Another Synchronization Mechanism — Monitors

- History — Hoare (1975) and Brinch Hansen (1975).
- Idea — combine synchronization and object-oriented paradigm.
- A monitor consists of
  - Data for a shared object (and initial values).
  - Procedures — only one at a time can run.
- "Condition variable" ADT allows us to wait for specified conditions (e.g., buffer not empty):
  - Value — queue of suspended processes.
  - Operations:
    - ∗ Wait — suspend execution (and release mutual exclusion).
    - ∗ Signal — *if* there are processes suspended, allow *one* to continue. (if not, signal is "lost"). Some choices about whether signalling process continues, or signalled process awakens right away.

Slide 12

Bounded Buffer Problem, Revisited

- Define a `bounded_buffer` monitor with a queue and `insert` and `remove` procedures.
- Shared variables:

```
bounded_buffer B(N);
```

Pseudocode for producers:

```
while (true) {
    item = generate();
    B.insert(item);
}
```

Pseudocode for consumers:

```
while (true) {
    B.remove(item);
    use(item);
}
```

Slide 13

Bounded-Buffer Monitor

- Data:

```
    buffer B(N);  // N constant, buffer empty
    int count = 0;
    condition not_full;
    condition not_empty;
```

- Procedures:

```
insert(item itm) {             remove(item &itm) {
    if (count == N)                if (count == 0)
        wait(not_full);                wait(not_empty);
    put(itm, B);                   itm = get(B);
    count += 1;                    count -= 1;
    signal(not_empty);             signal(not_full);
}                              }
```

- Does this work? (Yes.)

Slide 14

Implementing Monitors

- Requires compiler support, so more difficult to implement than (e.g.) semaphores.
- Java's methods for thread synchronization are based on monitors . . .

Slide 15

Java’s Adaptation of the Monitor Idea

- Data for monitor is instance variables (data for class).
- Procedures for monitor are `synchronized` methods/blocks — mutual exclusion provided by implicit object lock.
- `wait`, `notify`, `notifyAll` methods.
- No condition variables, but above methods provide more or less equivalent functionality.

  Note that the language specs for Java allow spurious wake-ups. So “best practice” is to `wait()` in a loop, re-checking the desired condition. The textbook’s bounded-buffer code doesn’t do this (?!).

Slide 16

Yet Another Synchronization Mechanism — Message Passing

- Previous synchronization mechanisms all involve shared variables; okay in some circumstances but not very feasible in others (e.g., multiple-processor system without shared memory).
- Idea of message passing — each process has a unique ID; two basic operations:
  - Send — specify destination ID, data to send (message).
  - Receive — specify source ID, buffer to hold received data. Usually some way to let source ID be “any”.

Message Passing, Continued

Slide 17

- Exact specifications can vary, but typical assumptions include:
  - Sending a message never blocks a process (more difficult to implement but easier to work with).
  - Receiving a message blocks a process until there is a message to receive.
  - All messages sent are eventually available to receive (can be non-trivial to implement).
  - Messages from process A to process B arrive in the order in which they were sent.

Implementing Message Passing

Slide 18

- On a machine with no physically shared memory (e.g., multicomputer), must send messages across interconnection network.
- On a machine with physically shared memory, can either copy (from address space to address space) or somehow be clever.

Slide 19

Mutual Exclusion, Revisited

- How to solve mutual exclusion problem with message passing?
- Several approaches based on idea of a single “token”; process must “have the token” to enter its critical region.

  (I.e., desired invariant is “only one token in the system, and if a process is in its critical region it has the token.”)
- One such approach — a “master process” that all other processes communicate with; simple but can be a bottleneck.
- Another such approach — ring of “server processes”, one for each “client process”, token circulates.

Slide 20

Mutual Exclusion With Message-Passing (1)

- Idea — have “master process” (centralized control).

  Pseudocode for client process:

```
while (true) {
    send(master, "request");
    receive(master, &msg);
        // assume "token"
    do_cr();
    send(master, "token");
    do_non_cr();
}
```

  Pseudocode for master process:

```
bool have_token = true;
queue waitQ;
while (true) {
    receive(ANY, &msg);
    if (msg == "request") {
        if (have_token) {
            send(msg.sender, "token");
            have_token = false;
        }
        else
            enqueue(sender, waitQ);
    }
    else { // assume "token"
        if (empty(waitQ))
            have_token = true;
        else {
            p = dequeue(waitQ);
            send(p, "token");
        }
    }
}
```

Slide 21

Mutual Exclusion With Message-Passing (2)

- Idea — ring of servers, one for each client.

Pseudocode for client process:

```
while (true) {
    send(my_server, "request");
    receive(my_server, &msg);
        // assume "token"
    do_cr();
    send(my_server, "token");
    do_non_cr();
}
```

Pseudocode for server process:

```
bool need_token = false;
if (my_id == first)
    send(next_server, "token");
while (true) {
    receive(ANY, &msg);
    if (msg == "request")
        need_token = true;
    else { // assume "token"
        if (msg.sender == my_client) {
            need_token = false;
            send(next_server, "token");
        }
        else if (need_token)
            send(my_client, "token");
        else
            send(next_server, "token");
    }
}
```

Slide 22

Synchronization Mechanisms — Recap

- Low-level ways of synchronizing — using shared variables only, using TSL instruction. All seem tedious and inefficient.
- “Synchronization mechanisms” are more-abstract ways of coordinating what processes do. A key point is providing *something* that potentially makes a process wait. Examples include semaphores, monitors, message passing. Often built using something lower-level.

Slide 23

Minute Essay

- Alleged joke (from some random Usenet person):

  A man's P should exceed his V else what's a sema for?

  Do you understand this? (Remember that P is "down" and V is "up".)

Slide 24

Minute Essay Answer

- It's a pun. The idea is roughly that if you never have a situation in which you've attempted more "down" operations than "up" operations, you didn't need a semaphore. (Or that's what I think it means. The author might have had another idea!)