Slide 1

Administrivia

- (Review minute essay from last time.)

Slide 2

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 3

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 4

Bounded-Buffer Monitor

- Data:

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

- Procedures:

```
insert(item itm) {
    if (count == N)
        wait(full);
    put(itm, B);
    count += 1;
    signal(empty);
}
```

```
remove(item &itm) {
    if (count == 0)
        wait(empty);
    itm = get(B);
    count -= 1;
    signal(full);
}
```

- Does this work? (Yes.)

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 5

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 6

Slide 7

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”.

Slide 8

Message Passing, Continued

- 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

- 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 9

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 10

Slide 11

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 12

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");
    }
}
```

Synchronization Mechanisms — Recap

- Low-level ways of synchronizing — using shared variables only, using TSL instruction.
- Higher-level mechanisms — semaphores, monitors, message passing. Often built using something lower-level.

Slide 13

Minute Essay

- Can you think of any reason to use message passing on a shared-memory system?

Slide 14

Minute Essay Answer

- One reason might be that the programming model is in in some ways simpler. Another might be a desire to write code that will run on either a shared-memory system or a cluster of machines not sharing memory.

Slide 15