Slide 1

Administrivia

- *Please do not* reboot the machines in HAS 340! People use these remotely, and you may cause someone's program to crash. If you think a reboot is needed, ask a faculty member.

  (If the machines seem to be very slow, odds are it's a background program running. Try another machine.)

- (Review minute essay from last time.)

Slide 2

Implementing Semaphores, Continued

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

```
down() {
    bool zero;
    enter_cr();
    zero = (value == 0);
    if (!zero)
        value -= 1;
    else
        enqueue(current_process, queue);
    leave_cr();
    if (zero)
        block();    // mark current process blocked
}
```

```
up() {
    process p = null;
    enter_cr();
    if (empty(queue))
        value += 1;
    else
        p = dequeue(queue);
    leave_cr();
    if (p != null)
        unblock(p);     // mark p runnable
}
```

- `enter_cr()`, `leave_cr()` mostly like before; see p. 113.

Slide 3

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”).

Slide 4

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 5

Bounded-Buffer Monitor

- Data:

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

- Procedures:

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

Slide 6

Implementing Monitors

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

Slide 7

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.

Slide 9

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.

  (Why would you want to do this? programming model is in some ways simpler, doesn't require memory shared among processes.)

Slide 10

Minute Essay

- Which of the following have you done?
  - Message-passing programming?
  - Multithreaded programming in Java?
  - Other parallel/concurrent/threaded programming? (What?)