

Slide 1

Administrivia

- Homework 2 to be on Web later today; due next Monday.

Slide 2

Semaphores — Recap

- Semaphore ADT:
 - Value — non-negative integer.
 - Two operations, up and down; both atomic.
- Last time — solutions using semaphores for mutual exclusion problem, bounded buffer problem.

Implementing Semaphores

Slide 3

- 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 4

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

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
}

```

Slide 5

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

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 6

Slide 7

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 8

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: <pre>while (true) { item = generate(); B.insert(item); }</pre>	Pseudocode for consumers: <pre>while (true) { B.remove(item); use(item); }</pre>
--	--

Bounded-Buffer Monitor

- Data:

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

Slide 9

- 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);
}                          }
```

- 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 10

Slide 11

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 12

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 13

- 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 14

- 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.

Minute Essay

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

Slide 15

Minute Essay Answer

- One reason might be that the programming model is 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 16