

Administrivia

- Reminder: Quiz 2 Wednesday. Topics from parts of chapter 2 up through today.
- Homework 2 on the Web; due in a week. Homework 3 coming soon; to be due before midterm.
- Midterm October 23 (Wednesday).

Slide 1

Minute Essay From Last Lecture

- (Some people got the alleged joke; others didn't.)

Slide 2

Slide 3

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 4

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.

Slide 5

Synchronization Mechanisms — Review/Recap

- Synchronization using only shared variables seems to be 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.

Slide 6

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.

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 7

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.)

Slide 8

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 9

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 10

Yet Another Synchronization Mechanism — Message Passing

Slide 11

- 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 12

- 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 13

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 14

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.

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 15

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 16

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 17

Minute Essay

- Anything about synchronization mechanisms that is particularly unclear or that you want to hear more about?

Slide 18