Administrivia

- None.

Minute Essay From Last Lecture

- Tell me one thing you’ve learned from the textbook.
  There was a range of answers, from “operating systems are complicated” and “an operating system is like a circus” to “it’s made of paper” and “textbook?”
Message Passing — Review

- 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”.

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:

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

Pseudocode for master process:

```c
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");
        }
}
```

Mutual Exclusion With Message-Passing (2)

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

Pseudocode for client process:

```c
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:

```c
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.

Classical IPC Problems

- Literature (and textbooks) on operating systems talk about “classical problems” of interprocess communication.
- Idea — each is an abstract/simplified version of problems o/s designers actually need to solve. Also a good way to compare ease-of-use of various synchronization mechanisms.
- Examples so far — mutual exclusion, bounded buffer.
- Other examples sometimes described in silly anthropomorphic terms, but underlying problem is a simplified version of something “real”.
Dining Philosophers Problem

- Scenario (originally proposed by Dijkstra, 1972):
  - Five philosophers sitting around a table, each alternating between thinking and eating.
  - Between every pair of philosophers, a fork; philosopher must have two forks to eat.
  - So, neighbors can’t eat at the same time, but non-neighbors can.
- Why is this interesting or important? It’s a simple example of something more complex than mutual exclusion — multiple shared resources (forks), processes (philosophers) must obtain two resources together. (Why five? smallest number that’s “interesting”.)

Dining Philosophers — Naive Solution

- Naive approach — we have five mutual-exclusion problems to solve (one per fork), so just solve them.
- Does this work?
Dining Philosophers — Simple Solution

- Another approach — just use a solution to the mutual exclusion problem to let only one philosopher at a time eat.
- Does this work?

Dining Philosophers — Solution

- Another approach — use shared variables to track state of philosophers and semaphores to synchronize.
- i.e., variables are
  - Array of five state variables (states[5]), possible values thinking, hungry, eating. Initially all thinking.
  - Semaphore mutex, initial value 1, to enforce one-at-a-time access to states.
  - Array of five semaphores self[5], initial values 0, to allow us to make philosophers wait.
- And then the code is somewhat complex …
Dining Philosophers — Code

- Shared variables as on previous slide.

Pseudocode for philosopher \( i \):

```java
while (true) {
    think();
    down(mutex);
    state[i] = hungry;
    test(i);
    up(mutex);
    down(self[i]);
    eat();
    down(mutex);
    state[i] = thinking;
    test(right(i));
    test(left(i));
    up(mutex);
}
```

Pseudocode for function:

```java
void test(i) {
    if ((state[left(i)] != eating) &&
        state[right(i)] != eating &&
        state[i] == hungry) {
        state[i] = eating;
        up(self[i]);
    }
}
```

Dining Philosophers — Solution Works?

- Could there be problems with access to shared `state` variables?
- Do we guarantee that neighbors don’t eat at the same time?
- Do we allow non-neighbors to eat at the same time?
- Could we deadlock?
- Does a hungry philosopher always get to eat eventually?
Other Classical Problems

- Readers/writers.
- Sleeping barber.
- And others . . .
- Advice — if you ever have to solve problems like this “for real”, read the literature . . .

Minute Essay

- Which of the synchronization mechanisms we’ve talked about (semaphores, monitors, message passing) do you think you would prefer to use? Why?
- Anything about processes or synchronization you want to hear more about? particularly unclear?