Slide 1

Administrivia

- (By e-mail.)

Slide 2

Synchronization Mechanisms — Recap/Review

- "Synchronization mechanisms" — more-abstract ways of coordinating what processes do. A key point is providing *something* that potentially makes a process wait.
- So far — semaphores (as an ADT, how they can be used, how to implement).

Slide 3

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 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 not_full;
condition not_empty;
```

- Procedures:

```
insert(item itm) {
    while (count == N)
        wait(not_full);
    put(itm, B);
    count += 1;
    signal(not_empty);
}
```

```
remove(item &itm) {
    while (count == 0)
        wait(not_empty);
    itm = get(B);
    count -= 1;
    signal(not_full);
}
```

- Does this work?

Slide 6

Bounded-Buffer Monitor, Continued

- Does this work? Yes:
- Atomicity ensured by how monitors work (one procedure at a time).
- Wait/signal on two condition variables ensures that we only get if buffer is not empty and put if it's not full.

  Note: Some published solutions use `if` rather than `while`. In principle, should work, but some implementations generate "spurious wakeups", and they recommend always testing in a loop this way.

Slide 7

Implementing Monitors

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

Slide 8

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.

Slide 9

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 (not discussed this year but you can possibly imagine the key idea — processes that don't share memory can coordinate by sending each other messages, waiting for them to arrive).

  Often built using something lower-level.

Slide 10

Classical IPC Problems — Review

- 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 sometimes a simplified version of something "real".

Slide 11

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".)

Slide 12

Dining Philosophers — Naive Solution

- Naive approach — we have five mutual-exclusion problems to solve (one per fork), so just solve them.
- Does this work? No — deadlock possible.

Slide 13

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? Well, it "works" w.r.t. meeting safety condition and no deadlock, but it's too restrictive.

Slide 14

Dining Philosophers — Dijkstra 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 . . .

Slide 15

Dining Philosophers — Code

- Shared variables as on previous slide.

Pseudocode for philosopher i:

```
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:

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

Slide 16

Dining Philosophers — Dijkstra Solution Works?

- Could there be problems with access to shared `state` variables?

Slide 17

"Solution Works", Continued

- Could there be problems with access to shared `state` variables?
- No (because all accesses are "protected" by mutual-exclusion semaphore).
- Do we guarantee that neighbors don't eat at the same time?

Slide 18

"Solution Works", Continued

- Do we guarantee that neighbors don't eat at the same time?
- Yes:

  Semaphore `self[i]` has value 1 only when it's safe for philosopher `i` to eat — either when it became hungry neither neighbor was eating, or a neighbor that was eating stopped eating and did an "up" on `self[i]`.
- Do we allow non-neighbors to eat at the same time?

Slide 19

"Solution Works", Continued

- Do we allow non-neighbors to eat at the same time?
- Yes.
- Could we deadlock?

Slide 20

"Solution Works", Continued

- Could we deadlock?
- No:

  The "critical region" for semaphore `mutex` is finite, so no deadlock on that.

  We only suspend on `self[i]` if a neighbor is eating — and it will eventually stop and perform an "up".
- Does a hungry philosopher always get to eat eventually?

Slide 21

"Solution Works", Continued

- Does a hungry philosopher always get to eat eventually?
- Sadly, no. Two philosophers can starve the one between them, if things are timed just right (or wrong).
- This *is* fixable . . .

Slide 22

Dining Philosophers — Chandy/Misra Solution

- Original solution allows for scenarios in which one philosopher "starves" because its neighbors alternate eating while it remains hungry.
- Briefly, we could improve this by maintaining a notion of "priority" between neighbors, and only allow a philosopher to eat if (1) neither neighbor is eating, *and* (2) it doesn't have a higher-priority neighbor that's hungry. After a philosopher eats, it lowers its priority relative to its neighbors.

Slide 23

Other Classical Problems

- Readers/writers (in textbook).
- Sleeping barber, drinking philosophers, . . .
- Advice — if you ever have to solve problems like this "for real", read the literature . . .

Slide 24

Minute Essay

- Questions?