Slide 1

Administrivia

- Homework 3 on the Web (written problems now, programming soon). Due Monday.

Slide 2

Dining Philosophers Problem — Review

- Scenario:
  - 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 3

Dining Philosophers — Dijkstra Solution

- Solution uses shared variables to track state of philosophers and semaphores to synchronize:
  - 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 4

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 5

Dining Philosophers — Dijkstra 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?

Slide 6

Dining Philosophers — Dijkstra Solution Works?

- 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? Yes.
- Do we allow non-neighbors to eat at the same time? Yes.
- Could we deadlock? No.
- Does a hungry philosopher always get to eat eventually? Usually. Exception is when two next-to-neighbors (e.g., 1 and 3) seem to conspire to starve their common neighbor (e.g., 2).

Slide 7

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 8

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 9

Review — Processes and Context Switches

- Recall idea behind process abstraction — make every activity we want to manage a “process”, and run them “concurrently”.
- Apparent concurrency provided by interleaving. (Some) true concurrency provided by multiple cores/processors.
- To make this work — process table, ready/running/blocked states, context switches.
- Context switches triggered by interrupts — I/O, timer, system call, etc.
- On interrupts, interrupt handler processes interrupt, and then goes back to some process — but which one?

Slide 10

Which Process To Run Next?

- Deciding what process to run next — scheduler/dispatcher, using “scheduling algorithm”.
- When to make scheduling decisions?
  - When a new process is created.
  - When a running process exits.
  - When a process becomes blocked (I/O, semaphore, etc.).
  - After an interrupt.
- One possible decision — “go back to interrupted process” (e.g., after I/O interrupt).

Slide 11

Scheduler Goals

- Importance of scheduler can vary; extremes are
  - Single-user system — often only one runnable process, complicated decision-making may not be necessary (though still might sometimes be a good idea).
  - Mainframe system — many runnable processes, queue of "batch" jobs waiting, "who's next?" an important question.
  - Servers / workstations somewhere in the middle.
- First step is to be clear on goals — want to make "good decisions", but what does that mean? Typical goals for any system:
  - Fairness — similar processes get similar service.
  - Policy enforcement — "important" processes get better service.
  - Balance — all parts of system (CPU, I/O devices) kept busy (assuming there is work for them).

Slide 12

Aside — Terminology

- Discussion often in term of "jobs" — holdover from mainframe days, means "schedulable piece of work".
- Processes usually alternate between "CPU bursts" and I/O, can be categorized as "compute-bound" ("CPU-bound") or "I/O bound".
- Scheduling can be "preemptive" or "non-preemptive".

Slide 13

Scheduler Goals By System Type

- For batch (non-interactive) systems, possible goals (might conflict):
  - Maximize throughput — jobs per hour.
  - Minimize turnaround time.
  - Maximize CPU utilization.

  Preemptive scheduling may not be needed.
- For interactive systems, possible goals:
  - Minimize response time.
  - Make response time proportional (to user's perception of task difficulty).

  Preemptive scheduling probably needed.
- For real-time systems, possible goals:
  - Meet time constraints/deadlines.
  - Behave predictably.

Slide 14

Scheduling Algorithms

- Many, many scheduling algorithms, ranging from simple to not-so-simple.
- Point of reviewing lots of them? notice how many ways there are to solve the same problem ("who should be next?"), strengths/weaknesses of each.

Slide 15

First Come, First Served (FCFS)

- Basic ideas:
  - Keep a (FIFO) queue of ready processes.
  - When a process starts or becomes unblocked, add it to the end of the queue.
  - Switch when the running process exits or blocks. (I.e., no preemption.)
  - Next process is the one at the head of the queue.
- Points to consider:
  - How difficult is this to understand, implement?
  - What happens if a process is CPU-bound?
  - Would this work for an interactive system?

Slide 16

Shortest Job First (SJF)

- Basic ideas:
  - Assume work is in the form of "jobs" with known running time, no blocking.
  - Keep a queue of these jobs.
  - When a process (job) starts, add it to the queue.
  - Switch when the running process exits (i.e., no preemption).
  - Next process is the one with the shortest running time.
- Points to consider:
  - How difficult is this to understand, implement?
  - What if we don't know running time in advance?
  - What if all jobs are not known at the start?
  - Would this work for an interactive system?
  - What's the key advantage of this algorithm?

Slide 17

Round-Robin Scheduling

- Basic ideas:
  - Keep a queue of ready processes, as before.
  - Define a "time slice" — maximum time a process can run at a time.
  - When a process starts or becomes unblocked, add it to the end of the queue.
  - Switch when the running process uses up its time slice, or it exits or blocks. (I.e., preemption allowed!)
  - Next process is the one at the head of the queue.
- Points to consider:
  - How difficult is this to understand, implement?
  - Would this work for an interactive system?
  - How do you choose the time slice?

Slide 18

Minute Essay

- None — sign in.