

Administrivia

- Exam 1 was scheduled for October 12. Okay to move? More info by e-mail, if all agree to postpone.
- Lecture notes online. (If I forget to post them, please feel free to remind me!)

Slide 1

Dining Philosophers, Continued

- Solution from last time — okay? (Not quite, but very close.)

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 2

Slide 3

Dining Philosophers, Improved Version

- 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 4

Review — Processes and Context Switches

- Recall idea behind process abstraction — make every activity we want to manage a “process”, and run them “concurrently”.
(Try `ps -A f` on a Linux system.)
- Each process has a “virtual CPU” (registers, program counter, etc.) and is running some program.
(“Heavyweight processes” have other resources too — address space, files, etc. “Lightweight processes” (threads) share.)
Sometimes program must wait — for I/O, because of synchronization mechanism, etc.
- Apparent concurrency provided by interleaving.

Review — Processes and Context Switches

Slide 5

- 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?

Which Process To Run Next?

Slide 6

- 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 7

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 8

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 9

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 10

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 11

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 12

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 13

Priority Scheduling

- Basic ideas:
 - Keep a queue of ready processes, as before.
 - Assign a priority to each process.
 - When a process starts or becomes unblocked, add it to the end of the queue.
 - Switch when the running process exits or blocks, or possibly when a process starts. (I.e., preemption may be allowed.)
 - Next process is the one with the highest priority.
- Points to consider:
 - What happens to low-priority processes? (So, maybe we should change priorities sometimes?)
 - How do we decide priorities? (external considerations versus internal characteristics)

Slide 14

Shortest Remaining Time Next

- Basic idea — variant on SJF:
 - Assume that for each process (job), we know how much longer it will take.
 - Keep a queue of ready processes, as before; add to it as before.
 - Switch when the running process exits *or* a new process starts. (I.e., preemption allowed — requires recomputing time left for preempted process.)
 - Next process is the one with the shortest time left.
- Points to consider:
 - How does this compare with SJF?

Slide 15

Three-Level Scheduling

- Basic idea — break up problem of scheduling (batch) work into three parts:
 - Admissions scheduling — choose from input queue which jobs to “let into the system” (create processes for).
 - Memory scheduling — choose from among processes in system which to keep in memory, which to “swap out” to disk.
 - CPU scheduling — choose from among processes in memory which to actually run.
- Points to consider:
 - Are there advantages to limiting how many processes, how many in memory? What criteria could we use?
 - Are there advantages to the explicit three-level scheme?
 - Would this (or a variant) work for interactive systems?
 - Do all three schedulers have to be efficient?

Slide 16

Minute Essay

- Suppose you have a batch system with the following jobs.

job ID	running time	arrival time
A	10	0
B	6	0
C	20	10
D	6	10

Compute turnaround times for all jobs using first FCFS and then SJF.