Slide 1

Administrivia

- Reminder: Homework 2 written problems due today, programming problems Wednesday.

Slide 2

CPU Scheduling

- We mentioned in the discussion of process states that some transitions (e.g., “running” to “blocked”) don’t really involve any decision-making, but others (“ready” to “running”) do.
- Who/what makes these decisions? “scheduler”. Note that this is how `enter_cr` in the semaphore implementation avoids busy-waiting — in situations in which busy-waiting would be needed, instead it “yields” — moves from running to ready and allows scheduler to pick a new process to run.

Scheduling, Continued

- 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). But there are other choices.
- How to choose? various “scheduling algorithms”.

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?

Slide 5

Scheduler Goals, Continued

- 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).
- Other goals depend on system type.

Slide 6

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 7

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 8

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 9

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 10

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 11

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 12

Priority Scheduling

- Basic ideas:
  - Keep a queue of ready processes, as before.
  - Assign a priority to each process.
  - When a process starts (or, if we also allow for blocking, when it 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 13

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 14

Multiple-Queue Scheduling

- Basic idea — variant on priority scheduling:
  - Divide processes into “priority classes”.
  - When picking a new process, pick one from the highest-priority class with ready processes.
  - Within a class, use some other algorithm to decide (round-robin, e.g.).
  - Optionally, periodically lower processes’ priorities.

Slide 15

Some Other Scheduling Algorithms

- Guaranteed scheduling.

  "Guarantee" each process (of N) 1/N of the CPU cycles; (try to) schedule to make this true.

  Calculate, for each process, fraction of the time it has had the CPU in its lifetime, fraction it "should" have had; choose process for which actual time / entitled time is smallest.

- Lottery scheduling.

  Give each process one or more "lottery tickets" — more or fewer depending on its priority (so to speak); pick one at random to decide who's next.

- Fair-share scheduling.

  Factor in process's owner in deciding which process to pick. I.e., if two "equal" users, schedule processes such that user A's processes get about as much time as those of user B.

Slide 16

Scheduling and Threads

- If system uses both processes and threads, we now possibly have an additional level of scheduling.
- Details depend on whether threads are implemented in user space or kernel space:
  - In user space — runtime system that manages them must do scheduling, and without the benefit of timer interrupts.
  - In kernel space — scheduling done at O/S level, so context switches are more expensive, but timer interrupts are possible, etc.

Slide 17

What Do Real Systems Use?

- Traditional UNIX: two-level approach (upper level to swap processes in/out of memory, lower level for CPU scheduling), using multiple-queue scheduling for CPU scheduling. See chapter 10 for details.
- Linux: facilities for soft real-time scheduling and “timesharing” scheduling, with the latter a mix of priority and round-robin scheduling. See chapter 10 for details. As of kernel version 2.6.23, replaced with “Completely Fair Scheduler”, which sounds like what Tanenbaum calls “guaranteed scheduling”.
- Windows NT/2000/Vista: multiple-queue scheduling of threads, with round-robin for each queue.

Slide 18

One More Scheduling-Related Topic

- A question I used to use as homework:

  Recall that some proposed solutions to the mutual-exclusion problem (e.g., Peterson’s algorithm) involve busy waiting.

  Do such solutions work if priority scheduling is being used and one of the processes involved has higher priority than the other(s)? Why or why not?

  How about if round-robin scheduling is being used? Why or why not?

  Note that a process can be interrupted while in its critical region; if that happens, it is considered to still be in its critical region, and other processes wanting to be in their critical regions are supposed to busy-wait.

Slide 19

One More Scheduling-Related Topic, Continued

- Yes, with priority scheduling, a solution involving busy-waiting can fail ("priority inversion", in text). Not so with round-robin.

Slide 20

Sidebar — Simulating Scheduling Algorithms

- Can be helpful in understanding how these algorithms work to simulate what they do given a particular sequence of inputs.
- Example — batch system with the following jobs.

job ID	running time	arrival time
A	6	0
B	4	0
C	10	0
D	2	2

Asked to compute turnaround times for all jobs using FCFS, what would you do . . .

Slide 21

Minute Essay

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

job ID	running time	arrival time
A	6	0
B	4	0
C	10	0
D	2	2

  Compute turnaround times for all jobs using SJF.

Slide 22

Minute Essay Answer

- Solution:

job ID	start time	stop time	turnaround time (SJF)
A	6	12	12
B	0	4	4
C	12	22	22
D	4	6	4