CSE 373: Heaps
(applications)

Chapter 6

Resource Management

Many of today's examples will examine how resources can be shared between multiple users

- fairly…
- without wasting the resources…

This is a complex issue and it gets a great deal of study (queuing theory; operating systems)

Q: How well do you need to understand this?

A: Well enough to understand why we might use heaps (*i.e.*, pay attention and "get it," but don't freak)

Application 1: Printer Queue

*(**Note:** Most printer queues are actually FIFO)*

The idea:

- you submit your document to the printer queue
- eventually, it gets its turn and prints out
- afterwards, the job is dropped (not needed again)

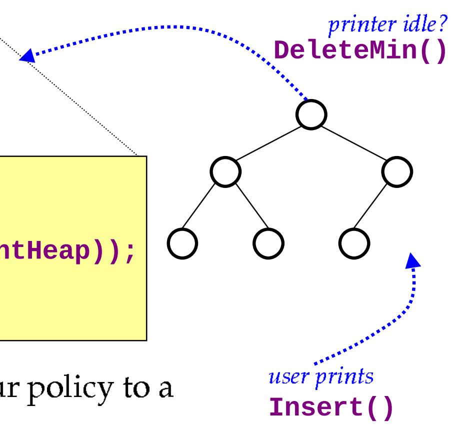
Imaginary Queue Policy:

- shorter print jobs should always print before longer ones (we'll measure length by # pages)
- jobs with equal length should print in FIFO order

This looks like a job for a Heap!

printer

```
while (1) {
  if (!printing) {
    Print(DeleteMin(PrintHeap));
  }
}
```



Q: How can we convert our policy to a numerical priority?

Printer Priority Scheme

Proposal: Priority = 1000 × (# *pages*) + counter

1) AJ prints a 2-page document
2) Brad prints a 30-page document
3) Sun Liang prints a 1-page document
4) AJ prints a 1-page document
5) Sun Liang prints a 1-page document

Problems with this scheme?

Why do people use FIFO?

Application 2: NQS

NQS: Network Queuing System

How it works:

- A supercomputer has 256 processors that can be used to run programs
- To use the computer, you must submit your program to NQS
- When submitting a program, you request a number of processors & an amount of time (*e.g.*, 8 processors for 1 hour)
- Eventually, NQS will assign your job to a set of processors according to your request
- if your job doesn't complete in the time you requested, it is killed so that other programs may use the processors
- if it does complete in time, it's dropped

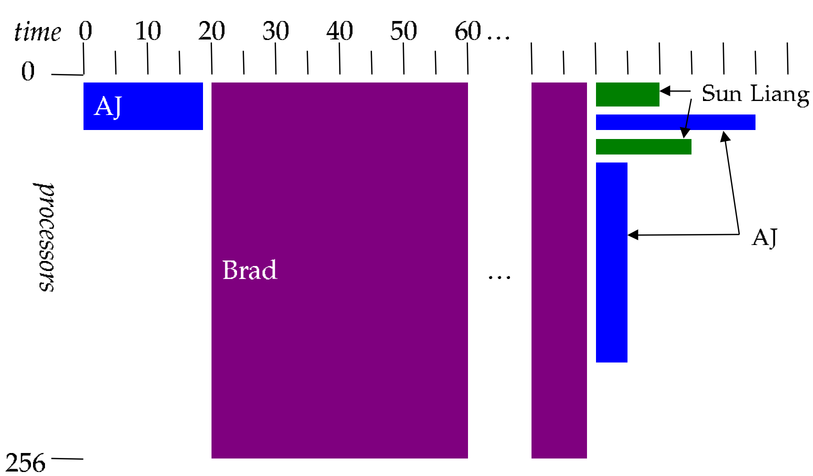
FIFO Queue Approach

For simplicity, we could just use a FIFO queue…

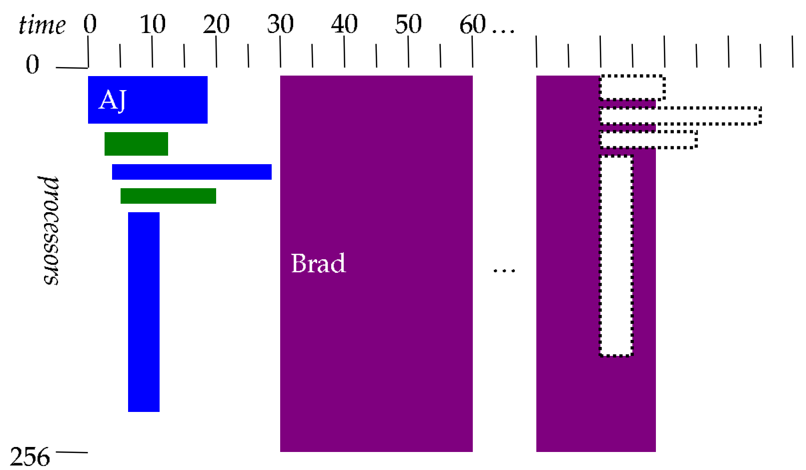
1) AJ submits a 32-processor, 20-minute job
2) Brad submits a 256-processor, 8-hour job
3) Sun Liang submits a 16-processor, 10-minute job
4) AJ submits an 8-processor, 25 minute job
5) Sun Liang submits an 8-processor, 15-minute job
6) AJ submits a 128-processor, 5 minute job

What's the problem with using FIFO in this example?

FIFO visualization



A Better Schedule



"Smallest Job First" Heap

Proposal: priority = job size

job size = # *processors* × *requested time* (in minutes)

What's the problem with always running the "smallest" job first?

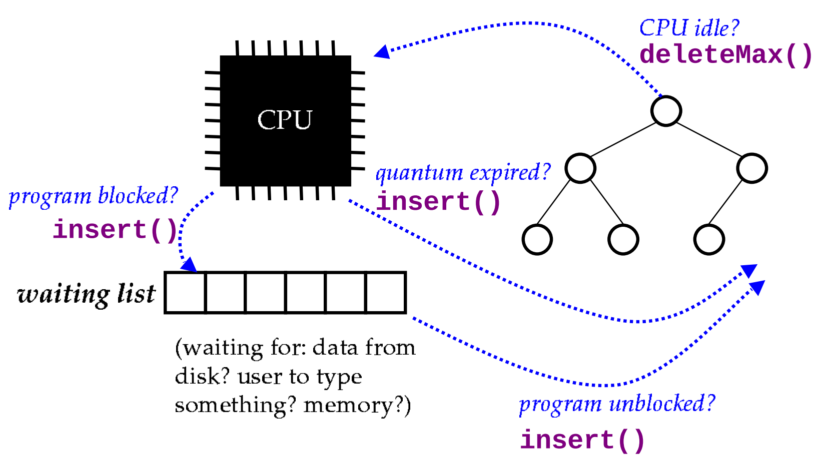
What's the solution?

Application 3: Sharing a CPU

The idea:

– all computers have a CPU (*e.g.*, Pentium-II) that can only run one program at a time
– to make it *seem* like many programs are running at once, the CPU takes turns running each for a short time (a *quantum*)
– some programs are more important than others and have higher priority
– programs waiting to be run can be kept on a heap
– CPU uses `deleteMax()` to find the most important program
– if the program blocks while running, it's put on a waiting list
– otherwise, once the quantum is up, it is re-`insert()`ed

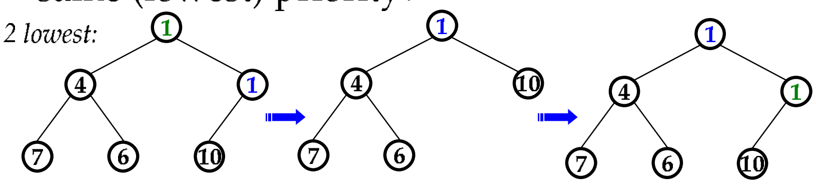
Shared CPU: Diagram



Shared CPU: Priorities

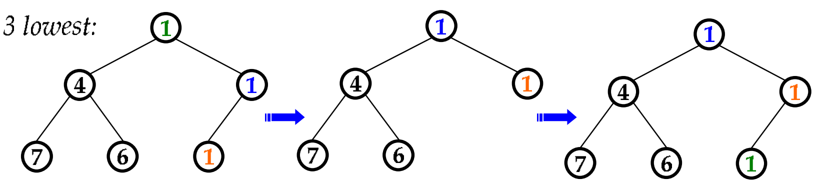
- User may select an initial priority
- Operating System may adjust priority if program hogs CPU or never gets to run
- What if we have multiple programs with the same (lowest) priority?

2 lowest:

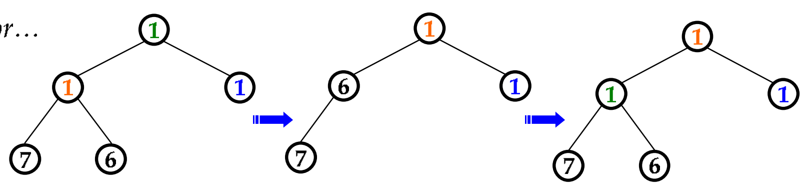


3 Programs, Same Priority

3 lowest:



or…



what next?

CPU Scheduling

- Tough problem
  - fairness vs. priority
  - must avoid *starving* processes
- Could modify priority based on lots of stuff:
  - how much a program has run vs. waited
  - how long it's been in the queue
  - etc.
- But…it's unclear (to me) whether a Priority Queue is really the best way to go…

Application 4: Simulations

Q: Let's say we've designed a policy for any one of these applications. How could we evaluate it?

A: Could simulate an artificial workload:

- set up a time scale (seconds, milliseconds, etc.)
- keep track of *events*. For example:
  - a job is submitted to NQS
  - a job starts running
  - a job finishes running
- submission time and running time are set (randomly?) by the workload
- starting and finishing times depend on our scheduling policy

Simulations: Continued

- Could simulate time tick-by-tick:

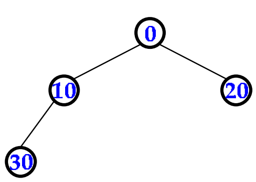
```
while (1) {
  time++;
  for (job=0;job<numjobs;job++) {
    CheckForEvent(job,time);
  }
}
```

  - *inefficient*, since there are more ticks than events
- Instead, keep a priority queue of events where events are prioritized by time
  - `deleteMin()` will give us the next event

Simulations

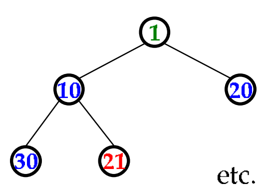
sample workload:

- at t=0, AJ submits 32-proc, 20-min job
- at t=10, Brad submits a 256-proc, 8-hour job
- at t=20, Sun Liang submits a 16-proc, 10-min job
- at t=30, AJ submits an 8-proc, 25-min job



next event is AJ's submission;

policy says we should run AJ's job at *t*=1



etc.