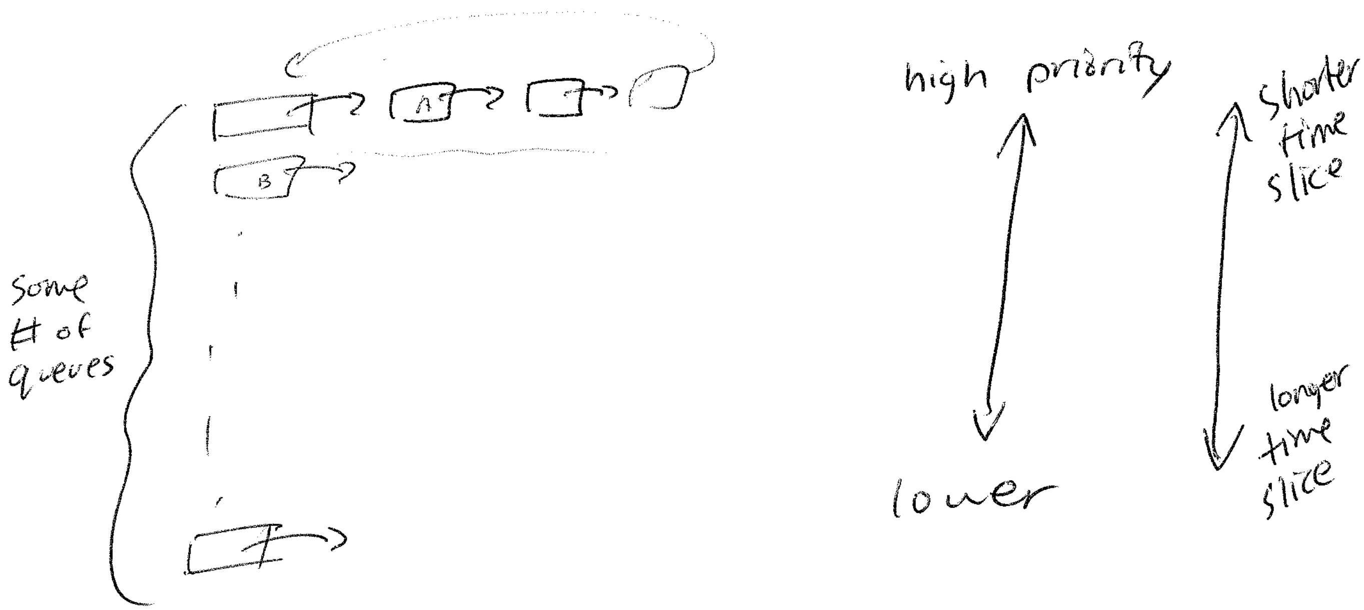
> Goal: want to
  > run shorter "interactive" jobs first
  > handle I/O bound jobs well
  > also handle CPU bound jobs

⇒ dynamically try to approx. STCF

> how: priority-based methods
  > many out there
  > Basic idea



1) if P(A) > P(B), A will run
2) Among jobs A, B s.t. P(A) = P(B), use round-robin (or similar)

Fall '04 do last ⟹ [3) Processes priority may vary over time]

4) Time-slice may vary per queue

> **Key**: How to decide when priority should change?

Goal: short, interactive ↑
long, CPU-bound ↓

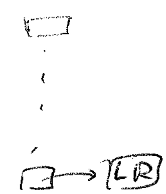
Attempt # 1:

> Start @ top
> if give up CPU before end of quantum, stay at ~~top~~ same level
> if use full quantum, move down 1 notch

when good?

> 1 long-running job has been in system
> new job comes along
interactive, CPU bound?

- interactive → will be serviced **first**!
- CPU bound → ok, will slowly move down, eventually settle



why bad?

1) can "game" scheduler

2) can lead to **starvation**

Fall '04
3) job that changes it's behavior: not accounted for

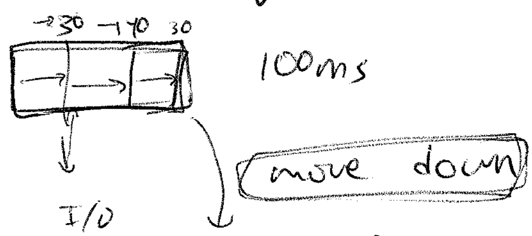
ML #2: [let them figure these out]

> periodically, move ~~[illegible]~~ job back to top queue

⇒ no more starvation, BUT still can game

ML #3:

> don't reset quantum counter upon I/O



⇒ no more gaming (no value to it)

Real Implementations

BSD: "Berkeley Unix"

⇒ formula for priority

"exponential decay"

SVR4: "AT&T Unix"

table-driven approach; sys admin can tune

NT?:

Conclude

- Complex but adaptive scheduler
- works well for many workloads
- How to control what's going on?

Lottery:

Problems w/ multi-level feedback

- Little control to apps (fixed % of CPU)
- Hard to understand parameters
- Per job, not per user

Proportional-share scheduling

New ~~mechanism~~ concept: tickets

each user (or process) may get some #

e.g. 2 clients, A, B

CPU

A: 100   100/(100+200) = 1/3 of resources

B: 200   200/( ) = 2/3

Currencies: (can skip)

Users have fixed # of tix in base currency

Can distribute tix in their own user currency

E.g.

A: 400 ⇒ A_1 600 ⇒ A_2 [100]

B: 50 ⇒ B_1, 30 ⇒ B_2 20 ⇒ B_3 [200]

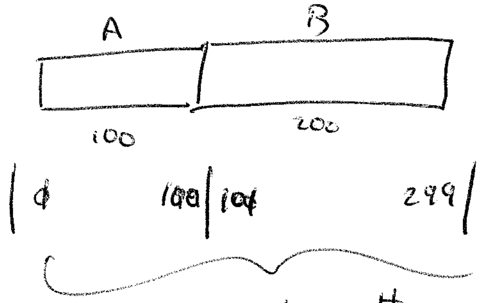
$A_1 = \frac{400}{1000} \cdot 100 \Rightarrow 40$ $A_2 \Rightarrow 60$

$B_1 \quad \frac{50}{100} \cdot 200 \Rightarrow 100$ $B_2 = 60$ $B_3 = 40$

Implement : PS ⇒ Lottery

Scheduling decision : hold a **lottery**!
Winner gets time-slice

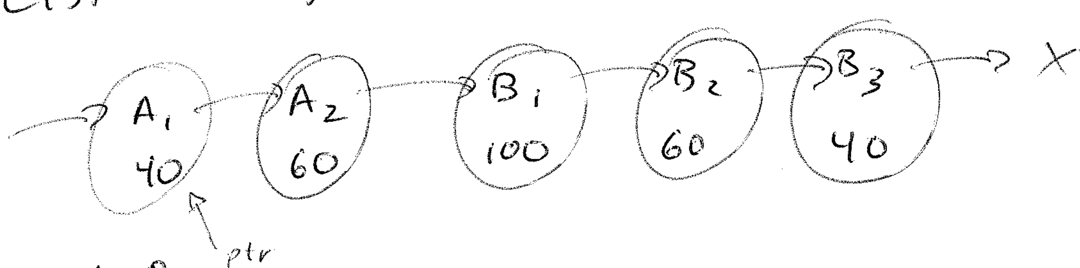
How does that work?



⇒ more tickets, more likely to win

random # between 0 → 299 inclusive

How to implement?
> List of jobs



count = 0
pick R = random (0 → 300)

```
count += tix(ptr)

if (count >= R) {
    winner(ptr);
}
```

> How to order list for shortest traversals?

> Could also do <u>deterministically</u>
  (think about <u>how</u>)

> <u>Positives</u>
  Could apply to many resources
  Proportional share w/ user <u>control</u>
  Simple to implement

> <u>Negatives</u>
  Not general purpose (<u>I/O</u>, <u>interactive</u>)

Projects:
  Extra office hours
    tomorrow: will send out time,

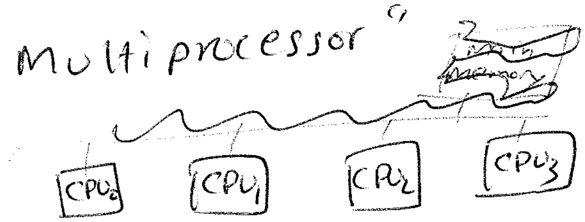
Monday?

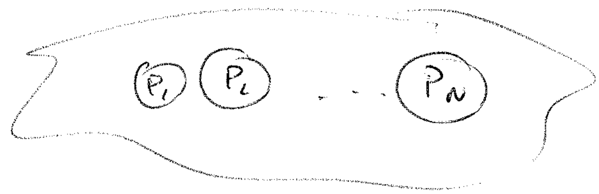
email: cs537-help@cs
  at any time for help

Multiprocessor Issues

M1

SMP ⇒ "Symmetric Multiprocessor"
[shared memory]



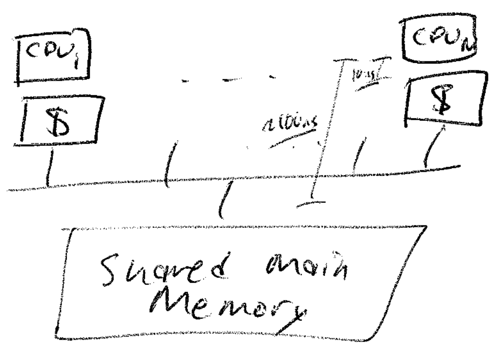


pool of processes

> How to allocate P ⇒ C ?

> Hardware

e.g.

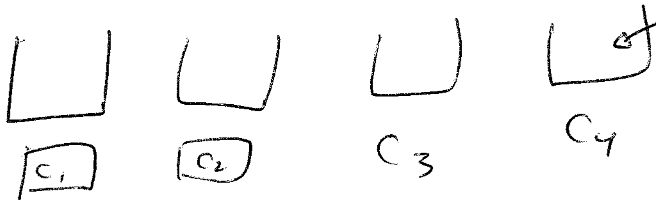


memory bus

Usually
- > Small # of CPUs
- > Uniform access time to memory
- > Single OS

> caches

Approach 1: Per CPU queue



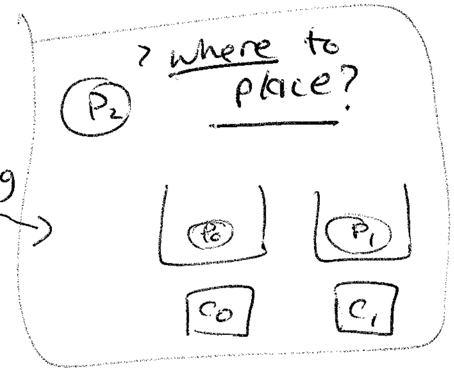
> Process enters, pick queue and stay there forever

Positive:

> simple
> Run on same processor ⇒ use same cache

Negative:

> Load imbalance could occur (lose natural advantage of smp) — e.g.

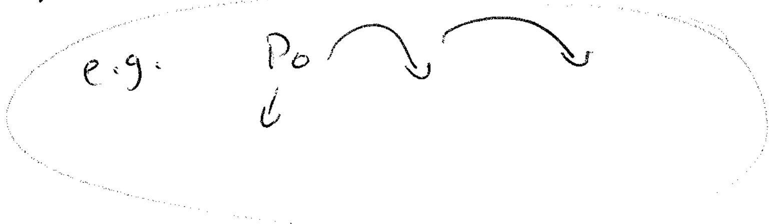


Approach 2: [Global Queue] ⟸ do this first (Fall '04)

> generalize MLFQ
  highest n jobs run

Positive
> allows dynamic load balancing

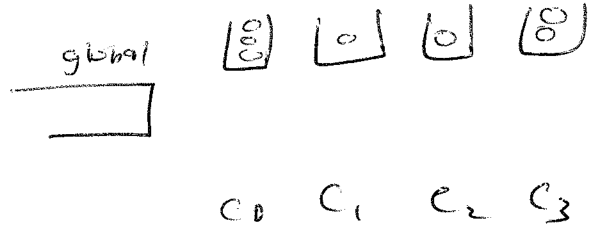
Disadvantage
> single queue: contention! might limit scalability
> might not be cache sensitive



M3

Approach : Hybrid

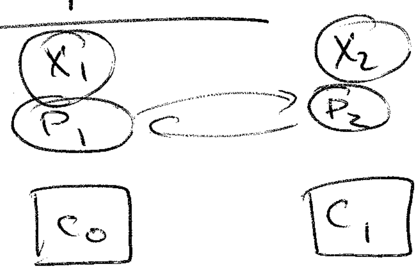
> Both local + global queues
  > usually have jobs in local queue
  > peek in other queues occasionally



> Load balance
  > if none in local queue, look elsewhere in ready queues (job stealing)
  > use global queue for kernel priority threads

> Processor Affinity
  > Try to keep Process on same CPU

Parallel processes



> might want to comm. w/ one another

> Coordinate CPU switches
  global ctxt switch => co-scheduling