Goal: want to

- Run shorter "interactive" jobs first
- Handle I/O bound jobs well
- Also handle CPU bound jobs

⇒ Dynamically try to approach 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)

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 \( J \)

long, CPU-bound \( L \)

Algorithm 2:

1. Start at top
2. If give up CPU before end of quantum, stay at same level
3. If use full quantum, move down 1 notch

When good?

1. Long-running job has been in system
2. New job comes along
   - interactive, CPU bound?
     - Will be served first
   - Else, will stay max down, eventually settle

Why bad?

1. Can "game" scheduler
2. Can lead to starvation
3. Job that changes its behavior: not accounted for
ML #2: [let them figure these out]

> periodically, move job back to top queue

⇒ no more starvation, BUT still can get

ML #3:

> don't reset quantum counter upon I/O

⇒ no more gaming (no value to it)

Real Implementations

BSP: "Berkley 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: tickets
Each user may get some #
E.g., 2 clients, A, B

\[
\begin{align*}
A & : \text{100} \\
B & : \text{200}
\end{align*}
\]

\[
\begin{align*}
\frac{100}{100+200} &= \frac{1}{3} \text{ of resources} \\
\frac{200}{(\quad)} &= \frac{2}{3}
\end{align*}
\]

Currencies: (can skip)
Users have fixed # of tick in base currency
Can distribute tick in their own user currency
E.g.,

\[
\begin{align*}
A & : 400 \Rightarrow A_1, \quad 600 \Rightarrow A_2 \\
B & : 50 \Rightarrow B_1, \quad 30 \Rightarrow B_2, \quad 20 \Rightarrow B_3
\end{align*}
\]

\[
\begin{align*}
A_1 &= \frac{400}{1000} \cdot 100 \Rightarrow 40 \\
A_2 &= \frac{60}{1000} \\
B_1 &= \frac{50}{100} \cdot 200 \Rightarrow 100 \\
B_2 &= 60 \\
B_3 &= 40
\end{align*}
\]
Implement: PS \rightarrow Lottery

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

How does that work?

\[
\begin{array}{c|c|c}
A & B & \rightarrow \text{more tickets, more likely to win} \\
100 & 200 & \\
| 1 & 100 | 100 | 299 |
\end{array}
\]

\text{random \# between 0-299 inclusive}

How to implement?

\rightarrow \text{List of jobs}

\rightarrow A_1, 40 \quad A_2, 60 \quad B_1, 100 \quad B_2, 60 \quad B_3, 40

\text{count} = 0
\text{pick} R = \text{random} (1 \rightarrow 300)
\text{count} += f(x)(\text{ptr})
if (count \geq R) 
\begin{tabular}{l}
\text{winner(ptr);}
\end{tabular}

\rightarrow \text{How to order list for shortest traversals?}
> Could also do deterministically
   (think about how)

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

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

Projects:
   Extra office hours
   tomorrow will send out time,
   Monday?
   email: cs537-help@cs
   at any time for help
Multiprocessor Issues

SMP = "symmetric multiprocessor"

\[ \text{[shared memory]} \]

CPU1 CPU2 CPU3

\( P_1, P_2, \ldots, P_n \)

pair of processes

? How to allocate \( P \rightarrow C \) ?

? Hardware

concurrency

\[ \text{Shared main Memory} \]

\[ \text{memory bus} \]

\( e.g. \)

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

> caches
Approach 1: Per CPU queue

Positive:
- Simple
- Run on same processor ⇒ use same cache

Negative:
- Load imbalance could occur (lose natural advantage of SMP)

Approach 2: Global Queue

Positive:
- Allows dynamic load balancing

Disadvantage:
- Single queue: contention! might limit scalability
  - might not be cache sensitive
  - e.g., Po l
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 (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 ctx+ switch => co-scheduling