UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department
CS 537
Intro to Operating Systems
Arpaci-Dusseau

Advanced Topics in Scheduling

Questions answered in these notes

- How does multi-level feedback scheduling work?
- How can we build a scheduler with proportional sharing?
- How is scheduling performed on multiprocessors?

No reading for this topic

Multi-level Feedback Queues

Goals

- Run short, interactive jobs quickly
- Handle I/O-bound jobs effectively
- Properly deal with CPU-bound jobs too

Tough to handle all these demands with simple, fixed policy

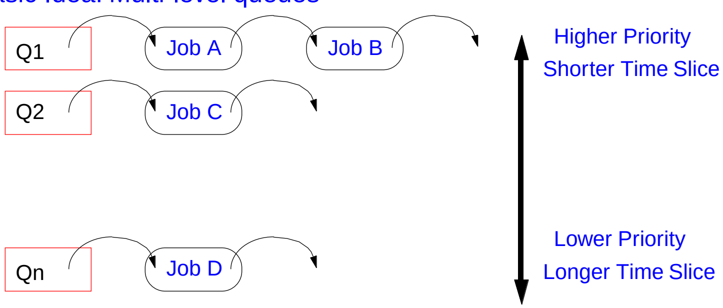
- Scheduler needs to be adaptive
- Goal: Approach STCF behavior, but w/o knowledge of job length

How? Priority-based methods

- Many different ones out there
- Will go over the basic idea

Multi-level queues with feedback

Basic Idea: Multi-level queues



Four behaviors

- If Pri(A) > Pri(B), A will be in higher queue and therefore run
- If Pri(A) = Pri(B), A and B are in same queue, so use round-robin
- Time slices: Shorter at top, longer at bottom
- Key: Process priority may vary over time

Changing Your Priority

How to decide when to change priority?

- Remember goals of supporting interactive, I/O bound, and CPU-bound jobs effectively

Attempt #1

- Job enters system at top queue
- If Job gives up CPU before end of time slice, stays at same level
- If full quantum is used, move down a level

Evaluation

- Assume there is 1 long running job in system
- New job comes in: could be short, interactive, or long and CPU-bound

Potential problems: Gaming the scheduler, starvation

More approaches

Approach #2

- Periodically, move jobs back to top of the queue
- How does this improve the situation?

Approach #3

- Don't reset time-slice when job gives up CPU
- Thus, gaming becomes harder

Real implementations

- BSD Unix: Exponential delay (formula for changing priority)
- SVR4 Unix (Solaris): Table-driven approach, allows admin to tune

Conclude

- Complex, adaptive, works pretty well, but hard to control/understand

Motivation for Lottery Scheduling

Multilevel Feedback Queue: Not appropriate for all workloads

Disadvantages

- Little control provided to applications
- Parameters are difficult to understand
- Difficult to give fixed percentage to one job versus another
- Users running more jobs get more of CPU

Proportional-Share Scheduling

Each client gets resource in proportion to number of *tickets*

- Two clients: Client A has 100 tickets; Client B has 200 tickets
  - Client A: 100/(100 + 200) = 1/3 of resources
  - Client B: 200/(100 + 200) = 2/3 of resources

Potential Clients: Users and/or Processes

Form hierarchy with *currencies*

- Users have fixed number of tickets in **base currency**
- Users distribute tickets to jobs in their **user currency**
- Example
  - User A: 400 tickets to Job 1A and 600 tickets to Job 2A
  - User B: 50 tickets to Job 1B, 30 tickets to Job 2B, 10 tickets to Job 3
- What proportion of resources does each job receive?

Currencies

Convert tickets in user currency to tickets in base currency

- User A: 100 base tickets
- 400 tickets to Job 1A and 600 tickets to Job 2A -- 1000 user tickets

Job 1A: $\frac{400}{\cancel{1000}} = \frac{x}{\cancel{100}} \quad x = 40$

Job 2A: $\frac{600}{\cancel{1000}} = \frac{x}{\cancel{100}} \quad x = 60$

- User B: 200 base tickets
- 50 tickets to Job 1B, 30 to Job 2B, 10 to Job 3 -- 90 user tickets

Job 1B: $\frac{50}{\cancel{90}} = \frac{x}{\cancel{200}} \quad x = 111$

Job 2B: $\frac{30}{\cancel{90}} = \frac{x}{\cancel{200}} \quad x = 67$

Job 3B: $\frac{10}{\cancel{90}} = \frac{x}{\cancel{200}} \quad x = 22$

Lottery Scheduling

Implementation of Proportional-Share Scheduling

- Scheduling decision --> Hold a lottery with each job’s tickets
- Schedule winning job for a time-slice

Clients with more tickets expected to win more frequently

How do you implement lottery?

- Pick a random number, *n*, between 1 and *GLOBAL_TICKETS*
- Scan list of jobs, increment *count* by job’s base tickets
- If *count* >= *n*, this is winning job

1..300
n=214

	1A	2A	1B	2B	3B
	40	60	111	67	22
count	*40*	*100*	*211*	*278*	*300*

Deterministic version: Stride scheduling

Lottery Scheduling Conclusion

Interesting Concept

Advantages

- Apply to resources other than CPU (amount of memory)
- Can specify proportional-share
- Good for multimedia applications
- Provides hierarchical control
- Easy to donate tickets to others (when waiting on locks)
- Simple implementation (conceptually)

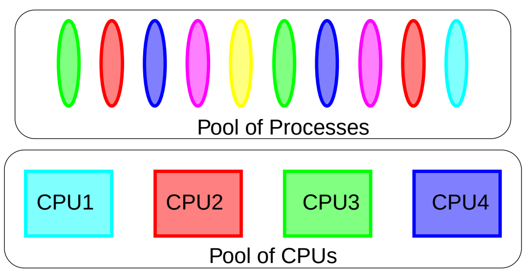
Disadvantages

- Not ready for general-purpose workloads with interactive jobs

Multiprocessor Scheduling Issues

So far in course: Uniprocessor scheduling

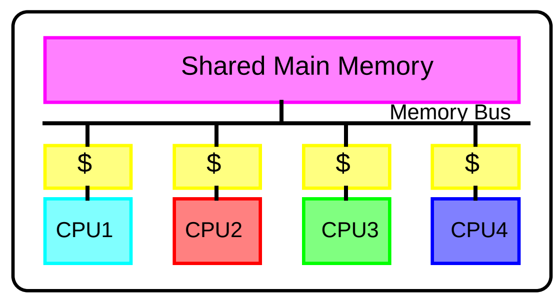
Shared-memory Multiprocessor t



How do we allocate processes to CPUs?

Symmetric Multiprocessor

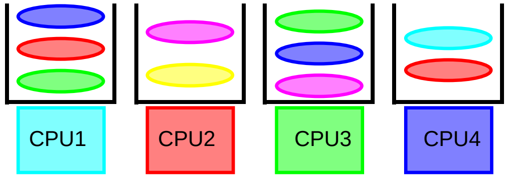
Architectural Overview



- Small number of CPUs
- Same access time to all of main memory
- Cached accesses
- Single copy of operating system

SMP: Local Queues of Processes

When process enters system, pick CPU to always execute



Advantages

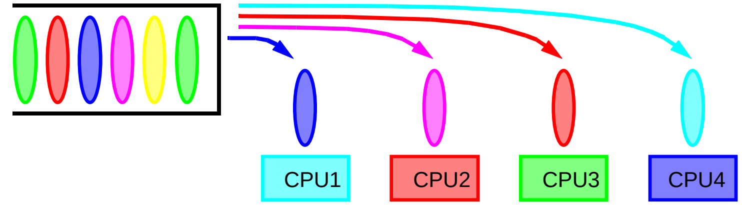
- Very simple to implement
- No contention for shared resources
- Maintains locality of cache references

Disadvantages

- Load-imbalance (some CPUs have more ready processes)
  --> Unfair to processes and lower utilization
- Losing power of SMP to easily share

SMP: Global Queue of Processes

Pick *n* jobs with highest priority in system



Advantages

- Good utilization of CPUs (always busy if a ready job)
- Fair to all processes

Disadvantages

- Contention for global queue (acquire locks!) --> Limits scalability
- No locality maintained in cache or memory subsystem

  More important with Non-Uniform Memory Access (NUMA)

SMP: Hybrid Approach

Combination of Local and Global Queues

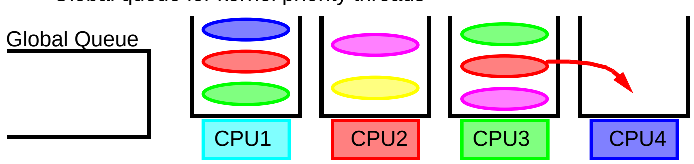
- Low contention for ready queues

  Usually have ready job in local queue

  Peek in other queues occassionally
- Load-Balancing

  If no local jobs, look for highest-priority ready job in remote queues

  Global queue for kernel priority threads



- Processor Affinity

  Add job to local ready queue if run recently (infer cache state still present)

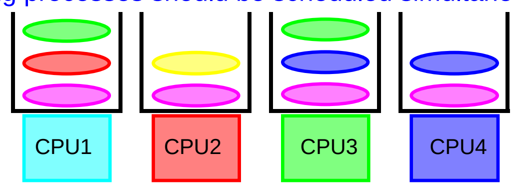
  Else, add job to remote queue with lowest-priority running job

SMP: Coordination Issues

Some processes are related to one another

- Processes in a parallel job
- Any cooperating processes (send + receive, synchronization)

Cooperating processes should be scheduled simultaneously



Dispatcher on each CPU does not act independently

- **Coscheduling (gang-scheduling)**:
  Pick set of processes to run simultaneously
- **Global context-switch** across all CPUs

SMP Scheduling Conclusions

Modifications to uniprocessor scheduling algorithms

Issues

- Avoid contention for shared resource
- Fair and efficient performance for all processes
- Maintain memory locality
- Coordinate communicating processes

Alternatives

- Global ready queue: Contention and no locality
- Local ready queues: Not fair to different processes
- Hybrid: Compromise between global and local queues