Schedulers

- Topics:
  - Unix fair share scheduler
  - Solaris multi-level feedback queue scheduler
  - Linux completely fair scheduler
  - Lottery scheduling
- Learning objectives:
  - Describe different scheduling policies
  - Match different policies to different scenarios/ workloads/environments

4.4 BSD Fair-Share Scheduling

- Policy:
  - Every job gets a “fair share” of the processor
    - If there are N jobs, each gets a 1/N share.
  - If a job uses more than its share, decrease its priority; if it uses less, increase its priority (there is only one queue).
  - The highest priority process is the one that has used the least CPU time recently.

Fair Share Mechanisms

- Maintain per-process usage statistics.
- Apply priorities by discounting actual usage.
  - A high priority process scales its usage so it looks like it's used less of the processor than it has.
  - A low priority does the opposite (makes it look like it's used a lot more).

The Solaris Scheduler

- Policy:
  - One run queue/scheduler per CPU.
  - Multi-level feedback queues on each CPU.
  - How to allocate threads to each CPU?
    - Place on the processor with the shortest queue, but...
    - If memory is NUMA (non-uniform), place thread “close” to where its memory is allocated.
    - If architecture is hype-rthreaded and multicore, do not place threads on the same core if others are idle.

Multi-Level Feedback Queues

- Combines time slices, priority, and prediction.
- Goal
  - I/O jobs should be responsive.
  - Use processor efficiently (avoid extraneous scheduling switches).
- Mechanism:
  - Multiple run queues.
  - Each queue corresponds to a priority.
  - Lower priority queues have longer time slices.

MLFQ Policy

- Jobs start at a high priority with a 1 unit time slice.
- If a job uses its entire time slice:
  - We decrease its priority by 1 and
  - double its time slice
- If a job blocks before it finishes its time slice:
  - We increase its priority by 1 and
  - Halve its time slice

Solaris Multiprocessing

- Move threads only when descheduled:
  - If another processor is idle, why not move the thread instead of making it wait?
- Considerations:
  - Load balancing.
  - Better memory utilization (separate memory hogs).
  - Sometimes lets a CPU go idle:
    - If you move two memory-hog processes together, it can actually decrease performance. A schedule that lets the CPU go idle is called “non-workconserving”. These are relatively rare; most schedulers are “work-preserving”.

The Completely Fair Scheduler (CFS)

- Based on the Linux 2.6.23 scheduler.
- With much credit given to the O(1) scheduler.
- Guiding principles from O(1) scheduler:
  - Processor efficiency: no idle processors if there is work to be done.
  - Processor affinity: leave tasks on processors as much as possible.
  - Fairness: no task should go “too long” without being scheduled.
  - Interactive performance: even under high load, the system must be responsive.
  - Priorities: honor relative importance of tasks.
- Updated for CFS
  - Tasks should get equal share of processor.

High Level Structure

- Each processor does its own scheduling.
- Main idea: maintain balance (fairness) among tasks. If some task is not being given enough time, give it some (and vica versa).
- Each processor maintains:
  - Red/Black tree: contains a "timeline" of future task execution.
    - Keyed by how much time it has used the processor (virtual time = vruntime).
    - Picking a task to run is O(1)
    - Inserting a task when it's done is O(log N) where N is the length of the (per-processor) run queue.
  - Accounts for processor time in nanoseconds (not timeslices).
- Basic algorithm
  - Execute task with lowest virtual time (leftmost node).
  - Run it
  - Update its virtual time
  - Reinsert it

Similar but different

- No traditional priorities
  - Instead, use decay factors.
  - Decay factors determine how quickly the time a task is permitted to execute diminishes.
  - High priority tasks have low decay factors; low priority tasks have high decay factors.
- Group scheduling
  - A collection of tasks can share a virtual time.
  - Example:
    - Allocate 50% of the time to each of two users.
    - Allocate time within that allotment to individual tasks of each user.
- Scheduling classes
  - Allows for multiple scheduling policies.

Multiprocessor Scheduling

- Every CPU runs a migration thread
- `void load_balance()`
  - Attempts to move tasks from one CPU to another.
- When called:
  - Explicitly if runqueues are imbalanced
  - Periodically by timer tick.

Lottery Scheduling

- Allocates the CPU randomly, but proportionally.
- Mechanism:
  - Each job has some number of lottery tickets.
  - At each scheduling interval, pick a random ticket and run that process.
  - On average, a job P with twice as many tickets as job Q will run twice as often.
- Policy: How do you allocate tickets?

Lottery Scheduling

- Allocates the CPU randomly, but proportionally.
- Policy:
  - Each job has some number of lottery tickets.
  - At each scheduling interval, pick a random ticket and run that process.
  - On average, a job P with twice as many tickets as job Q will run twice as often.
- Policy: How do you allocate tickets?
  - The same way you assign priorities!
  - Can augment any priority-based policy.

Possible Ticket Policies

- Give every user the same number of tickets.
- Give every task the same number of tickets.
- Allocate tickets to a group of processes.
- Map a priority level to a number of tickets.
- Give Prof. Seltzer all the tickets.

Behavior of Lottery Scheduling

- **Pros**: What are the virtues of a lottery?
  - **Fair:** Every job has a chance to run.
  - Has some of randomness's robustness against change, strange workloads
  - You can do cool stuff with tickets! For example, you can let processes decise how to allocate their tickets among threads (user-level scheduling).
- **Cons:** What is the cost of a lottery?
  - On optimal workloads (which are hopefully “normal” workloads), you pay a performance tax.

Useful Features

- Currencies provide insulation between users.
  - Different users can use different currencies.
  - OS makes sure each user gets fair allocation.
  - User's tickets get distributed among his/her tasks.
- Ticket exchanges: permit tasks to work together.
  - Applications can give servers tickets to do work on their behalf.
  - Avoid priority inversion by giving your tickets to a process blocking you.
- Compensation tickets: allow applications that voluntarily relinquish the processor to hold tickets in reserve.