Priority Scheduling

Azza Abouzied

From Last Class: Scheduling Recap

Hybrids are almost always better

Quick Recap: We looked at single queue RR or FCFS algorithms.

Suppose you have

A: 80ms, B: 80ms, C: 50ms cpu; 160ms io

RR only

CPU utilization: 67%; Disk utilization: 40%

RR + FCFS queue for IO-bound tasks. Priority goes to IO task.

CPU utilization: 100%; Disk utilization: 60%

Terminology

Terms defined differently:

Mean Response time or latency: average time between arrival to completion of requests.

Others use the time to ‘first response,’ which is also messy to define … is it time to be scheduled? That gets us into the murky *wait time* territory, which is also time waiting ready to be run but put in the ready queue. Others use *turnaround time* to mean time to completion.

How long is a context-switch

Generally not long enough to matter but if there are too many context switches then it matters.

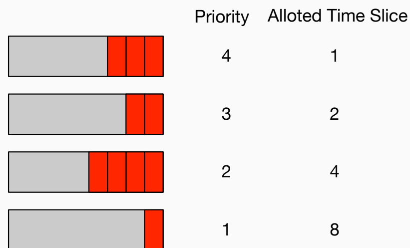
Context switch lasts from 100ns to 30μs. In Linux, the minimum granularity of a time slice can be as low as 1ms. At that rate, the context switch can be roughly 3% overhead!

Priority Scheduling

The Gist

1. Each process gets a priority.
2. Highest runs first. Can preempt the currently running process and take its place if higher priority.
3. Priorities can serve other purposes:
   - Interactive process get high priority
   - CPU-bound get low priority

Approach 1: Multi-level feedback queues



- Task starts at the highest priority.
- On timeout, it moves to the next lower priority queue;
- On interrupt, it moves up.

What does this scheduling algorithm achieve for IO-bound vs. CPU-bound?

Approach 2: Unix/Early Linux O(1) scheduler approach

Integrating user-defined priorities: Nice values (-20 to +20) map to priority map to time slices.

- Low nice $\rightarrow$ high priority $\rightarrow$ large time slice
- High nice $\rightarrow$ low priority $\rightarrow$ small time slice

Think of nice as a process's personality: the nicer it is, the more time it gives to others.

Runqueues further split into multiple queues of different priority.

The time slice alloted to a certain priority is calculated by scaling the priority range to the min, max time slice range

Can dynamically change priorities by boosting priority of a sleeping process or penalizing priority of a CPU-bound process.

Can this lead to problems?

Some problems

1. Not ideal: you want low priority CPU-bound tasks to actually have longer timeslices.

2. Additive scale. Differences in time slices are additive so larger relative time differences at the ends of the priority scale.

3. May not port well as timeslice now depends on timer ticks.

4. What about *sleeper fairness*? Boosting does help but you can game the system.

A definition of fairness?

What is perfect multitasking?

1. Two equal-priority processes get 50% of CPU-time: one runs for 100ms, the second runs for a 100ms.

2. Two equal-priority processes get 50% of CPU-time: both run *simultaneously* taking 50% of the processor.

Why is the second impossible multitasking more fair?

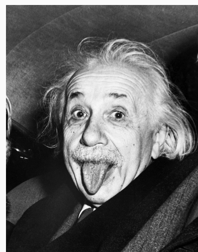
Fair-share scheduling. Suppose there was no overhead to context-switching, then if we switch frequently and fast enough between processes we can get close to perfect multitasking. Map priorities to shares of the processor rather than time slice allotments.

The Completely-Fair Scheduler

Ideal multi-tasking CPU is a non-existent CPU which can divy up its power precisely among tasks and execute them in parallel:

- 2 tasks of equal priority of 100 ms length run simultaneously on the CPU for 200ms each using 50% CPU power.

Problem: On real hardware, we can only run a single task at once.
Solution: Virtualization/Illusion



Virtual Runtime: *"the rate at which time passes depends on your ~~frame of reference~~ priority"*

CFS - a simplified outline

Virtual Run Time vt_i: For each task, the kernel keeps track of the actual time spent on the CPU and then weighs that time by task priority. e.g. $vt_i \mathrel{+}= 1/priority_i \times actualruntime_i$

1. Order tasks by virtual run time vt_i
2. Schedule task i with the least vt_i
3. Run task for some maximum execution time or until it is interrupted or blocks
4. Calculate vt_i
5. Repeat

Simplied CFS - details

What is the maximum execution time?

$$1/N \times targetlatency$$

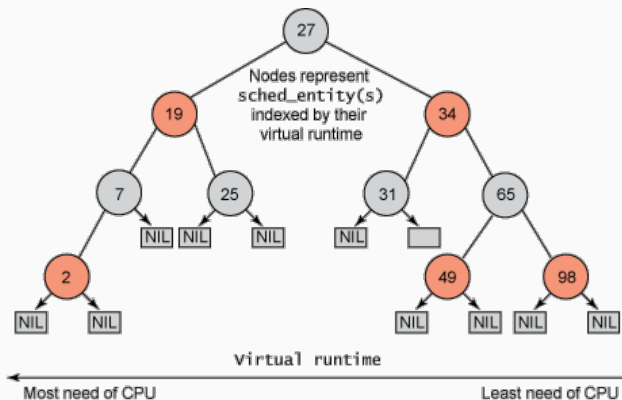
- $O(1)$ schedulers give constant switching rate, variable fairness
- Here, we get variable switching rate, constant fairness

How to quickly find the least vt_i?

- Store tasks in a Red-Black Tree ordered by virtual run time
- Pick the left-most node.

1. Order tasks by virtual run time vt_i
2. Schedule task i with the least vt_i
3. Run task for some maximum execution time or until it is interrupted or blocks
4. Calculate vt_i
5. Repeat

Red-black trees



Simplified CFS - Details

- Why have a notion of max-execution time?
- What happens when there are many tasks? Minimum granularity time. Is that fair?
- How does virtual run time handle sleeper fairness?
- What about new tasks? $min_i vt_i$
- How does it deal with priorities? Geometric scaling

1. Order tasks by virtual run time vt_i
2. Schedule task i with the least vt_i
3. Run task for some maximum execution time or until it is interrupted or blocks
4. Calculate vt_i
5. Repeat

Solution 3: Lottery Scheduling

How it works?

1. Each job gets a set of tickets.
2. Randomly pick a set of tickets
3. Schedule those

How can you use this for?

1. Priority
2. Promoting shorter jobs
3. Allowing Cooperation
4. Ensuring Fairness or Preventing Starvation

Interaction effects

The interplay of synchronization and preemption/scheduling can have the following effects:

1. **Priority Inversion Problems.** Low-priority thread acquires lock needed by high-priority thread.
   Solution: **Priority Inheritance***:* When a thread holds a lock that other threads are waiting on, give that thread the priority of the highest-priority thread waiting to get the lock.

2. **Convoy Effects.** A thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up.

Design Exercise

Design an elevator

- m elevators: start with $m = 1$ then scale. Be careful of concurrency control.
- n floors
- R_t a request at time time t. Requests are either up or down.

What are you optimizing for? How did that influence your policy?

Expand your design: Introduce priority.

Questions?