Scheduling

OS Lecture 11

UdS/TUKL WS 2015

Scheduling

What is “scheduling” and why is it necessary?

- To **share** a *serially reusable* resource among multiple processes.
  - processors, I/O links, bandwidth, memory…
- **Schedule**: determine the order in which use is granted. The *policy*, not the low-level mechanics of stopping and resuming (→ *dispatching*).
- Arises naturally when resources are *virtualized*.

Scheduling Goals

- **efficient use**: don't waste available capacity (processor time, bandwidth, …)
- **low overheads**: don't waste too much resource capacity on resource management
- **timeliness**: users typically have *some* expectations regarding timing. Examples:
  - minimize response times
  - provably meet (hard) deadlines
  - provide "smooth" user interface (soft deadlines)
  - meet customer service level agreements (SLAs)…

Types of Scheduling Problems

- **preemptive** vs. **non-preemptive**: a matter of timescale and cost
- **uni**- vs. **multiprocessor** scheduling: implicit or explicit load balancing needed for latter
- **identical** vs. **uniform** vs. **heterogenous** multiprocessors
- different **objectives**: make span, average response time, minimize (average/max) lateness, minimize (average/max) tardiness, …

FIFO: First-In First-Out / FCFS: First-Come First-Served

Policy: run jobs in order of arrival until they complete (or block).

POSIX: Available as `SCHED_FIFO` policy.

- **Advantages**: trivial implementation, minimal overheads (doubly linked list).
- **Disadvantage**: long-running jobs can dominate the resource, starvation.

RR: Round Robin

Policy: allocate resource in fixed-length *time slices*, preempt at end of time slice.

POSIX: Available as `SCHED_RR` policy.

- UNIX time-slice length used to be 100ms; nowadays 10ms is more appropriate.
- **Advantages**: avoids starvation; ensures fairness.
- **Disadvantages**: more preemptions; increased average response times; with many ready processes, bad responsiveness (system feels "sluggish" to user).

Example: Average Response Times

Three processes *A*, *B*, *C* arrive at times 0, 1, 2, and each requires 50ms to finish. What is the *average* response time with FIFO and RR (time slice: 1ms)?

- **FIFO**: *A* terminates at time 50, *B* terminates at time 100, *C* terminates at time 150: → **100ms**
- **RR**: *A* terminates at time 148, *B* terminates at time 149, *C* terminates at time 150: → **149ms**
- What if *C* requires only 10ms, whereas *A* requires 90ms?

Example: I/O-bound vs. CPU-bound process

Process *A*: compute 1ms, blocking I/O for 10ms, repeat…
Process *B*: infinite compute loop, no I/O

What happens with FIFO? What happens to I/O utilization when using **RR with 100ms** time slices?

- FIFO: *B* takes over processor, *A* starves.
- RR-100: I/O utilization drops to ~10% because *B* prevents *A* from issuing new I/O commands
- What happens if we use a shorter time slice? How to pick the right time slice length?

Shortest Remaining Time

AKA: “Shortest Time to Completion First” (STCF)

Policy: always schedule job which requires the least time to complete (or block).

» **Advantages**: optimal with regard to average response times, favors *interactive* processes.

» **Disadvantage**: requires knowledge of the future…

Locally, Past Behavior ≈ Future Behavior

How can we anticipate whether or not a process is going to hog the processor?

- **Observation**: program execution may move through *different phases*, but *in the short term*, I/O-bound processes stay I/O-bound, and CPU-bound processes stay CPU-bound.
- **Idea**: track execution and blocking times to *adaptively* predict future resource usage.
- This can be used to approximate STCF.

MLFQ: Multi-Level Feedback Queues

Idea: initially assume that a job will finish quickly, and demote long-running jobs.

- Have multiple *priority levels*, with one RR queue per priority
- time slice: high prio = short, low prio = long
- New jobs start at the highest priority
- If job does not finish before time slice ends, then *lower priority by one* and *double time slice length*.
  → "exponential queue"
- Problem: How can a *bursty process* recover priority?

4.4BSD Scheduler

Idea: adaptive like MLFQ, but use a constant time slice length determine priority based on recent CPU usage and allow fine-tuning with *nice* values.

» 128 priorities (0-127, 0-49 reserved for kernel)

» time slice length: 100ms
→ unchanged in 30 years!

» for each process, estimate recent CPU usage

» user can set *nice* value (range: -20 to +20)

4.4BSD Scheduler: Priority

The priority of a running process is recalculated every four clock ticks (40ms).

Given a usage estimate $\boldsymbol{p\_estcpu}$ and a nice value $\boldsymbol{p\_nice}$, the priority is set to:

$$\boldsymbol{p\_usrprio} \leftarrow \mathbf{50} + \frac{\boldsymbol{p\_estcpu}}{\mathbf{4}} + \mathbf{2} \times \boldsymbol{p\_nice}$$

($\boldsymbol{p\_usrprio}$ is capped to $\boldsymbol{p\_usrprio} \in [\mathbf{50}, \mathbf{127}]$.)

4.4BSD: Usage Tracking

On **every clock tick** (every 10ms), the variable p_estcpu of the *running* process is incremented.

Once per second, the accumulated usage of each ready process is *aged* (→ *exponentially weighted moving average*):

$$p\_estcpu \leftarrow \frac{2 \times load}{2 \times load + 1} \times p\_estcpu + p\_nice$$

$$load \leftarrow \frac{59}{60} \times load + \frac{1}{60} \times number\_of\_ready\_threads$$

4.4BSD: Waking Processes

When a process is blocked, it does not take part in "aging" → its CPU usage is not "forgotten."

Solution: "fixup" p_estcpu of waking processes.

$$p\_estcpu \leftarrow \left(\frac{2 \times load}{2 \times load + 1}\right)^{p\_slptime} \times p\_estcpu$$

Where $p\_slptime$ is the time the process was blocked (in seconds).

4.4BSD Scheduler: Issues

Can you think of some potential problems with the 4.4BSD design?

- What about `make`-like tasks that spawn many compute-intensive, but *short-running processes*?
- What if userspace processes have access to *accurate high-resolution timers*?

Generalized Processor Sharing

What is a “fair” share if some processes are more important than others (but none should starve)?

Generalized Processor Sharing

What is a "fair" share if some processes are more important than others (but none should starve)?

Proportional Share Fairness: given n competing processes and a **weight** w_i for each process, the *fair share* of process i over an interval of length Δ is:

$$share_i(\Delta) = \Delta \times \frac{w_i}{\sum_{j=1}^{n} w_j}$$

Lottery Scheduling

Idea: approximate prop-share fairness *stochastically*.

- Have some number of **lottery tickets** (*token abstraction*)
- Give each process a number of tickets *proportional* to its weight
- At beginning of each time slice, **randomly draw** a *winning ticket* and **schedule the winner**.
- Neat concept, but not widely adopted for processor scheduling: takes a relatively long time to converge.

STFQ: Start-Time Fair Queuing (1/2)

Idea: FIFO in principle, but make time “run slower” for heavy-weight processes.

- Track for each process a *virtual time*.
- Always schedule the process with the *earliest virtual time* (FIFO).
- When process is scheduled, *advance virtual time* at a rate *proportional to its weight*.

STFQ: Start-Time Fair Queuing (2/2)

Let T_i denote *virtual time* of process i.

Initially, $T_i \leftarrow realTime$ (time of process creation).

After running for Δ time units, T_i is advanced:

$$T_i \leftarrow T_i + \Delta \times \frac{\sum_{j=1}^{n} w_j}{w_i}$$

» What happens if a process blocks?

Fair Scheduling: Further Reading

There exist many, many more fairness-based schedulers:

Weighted Fair Queuing (WFQ), Virtual Time Round Robin (VTRR), Group Ratio Round Robin (GR^3), …

The Linux "Completely Fair Scheduler" (CFS) is also based on fairness. However, it is certainly not "completely fair" and in fact quite difficult to analyze.

In contrast, **provable lag bounds** are known for WFQ, VTRR, GR^3, etc.