Advanced Topics in Computer Systems, CS262A

Prof. Eric Brewer

Lottery Scheduling

I. Lottery Scheduling

Very general, proportional-share scheduling algorithm.

Problems with traditional schedulers:

- Priority systems are ad hoc at best: highest priority always wins
- "Fair share" implemented by adjusting priorities with a feedback loop to achieve fairness over the (very) long term (highest priority still wins all the time, but now the Unix priorities are always changing)
- Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- Schedulers are complex and difficult to control

Lottery scheduling:

- Priority determined by the number of tickets each process has: priority is the relative percentage of all of the tickets competing for this resource.
- Scheduler picks winning ticket randomly, gives owner the resource
- Tickets can be used for a wide variety of different resources (uniform) and are machine independent (abstract)

How fair is lottery scheduling?

- If client has probability p of winning, then the expected number of wins (from the binomial distribution) is np.
- Variance of binomial distribution: $\sigma^2 = np(1-p)$
- Accuracy improves with $\sqrt{n}$
- Geometric distribution used to find tries until first win
- Big picture answer: mostly accurate, but short-term inaccuracies are possible; see Stride scheduling below.

Ticket Transfer: how to deal with dependencies

- Basic idea: if you are blocked on someone else, give them your tickets
- Example: client-server
  - Server has no tickets of its own
  - Clients give server all of their tickets during RPC
  - Server's priority is the sum of the priorities of all of its active clients
  - Server can use lottery scheduling to give preferential service to high-priority clients
- Very elegant solution to long-standing problem (not the first solution however)

Scheduling

Ticket inflation: make up your own tickets (print your own money)

- Only works among mutually trusting clients
- Presumably works best if inflation is temporary
- Allows clients to adjust their priority dynamically with zero communication

Currencies: set up an exchange rate with the base currency

- Enables inflation just within a group
- Simplifies mini-lotteries, such as for a mutex

Compensation tickets: what happens if a thread is I/O bound and regular blocks before its quantum expires? Without adjustment, this implies that thread gets less than its share of the processor.

- Basic idea: if you complete fraction f of the quantum, your tickets are inflated by 1/f until the next time you win.
- Example: if B on average uses 1/5 of a quantum, its tickets will be inflated 5x and it will win 5 times as often and get its correct share overall.
- What if B alternates between 1/5 and whole quantums?

Problems:

- Not as fair as we'd like: mutex comes out 1.8:1 instead of 2:1, while multimedia apps come out 1.92:1.50:1 instead of 3:2:1
- Practice midterm question: are these differences statistically significant? (probably are, which would imply that the lottery is biased or that there is a secondary force affecting the relative priority)
- Multimedia app: biased due to X server assuming uniform priority instead of using tickets. Conclusion: to really work, tickets must be used everywhere. Every queue is an implicit scheduling decision... Every spinlock ignores priority...
- Can we force it to be unfair? Is there a way to use compensation tickets to get more time, e.g., quit early to get compensation tickets and then run for the full time next time?
- What about kernel cycles? If a process uses a lot of cycles indirectly, such as through the ethernet driver, does it get higher priority implicitly? (probably)

Stride Scheduling: follow on to lottery scheduling (not in paper)

- Basic idea: make a deterministic version to reduce short-term variability
- Mark time virtually using "passes" as the unit
- A process has a stride, which is the number of passes between executions. Strides are inversely proportional to the number of tickets, so high priority jobs have low strides and thus run often.
- Very regular: a job with priority p will run every 1/p passes.
- Algorithm (roughly): always pick the job with the lowest pass number. Updates its pass number by adding its stride.
- Similar mechanism to compensation tickets: if a job uses only fraction f, update its pass number by $f \times stride$ instead of just using the stride.
- Overall result: it is far more accurate than lottery scheduling and error can be bounded

Scheduling

absolutely instead of probabilistically