ADVANCED OPERATING SYSTEMS

Lottery Scheduling, Finding Bugs

Lottery Scheduling: Flexible Proportional-Share Resource Management (1994)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

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 Unix priorities are always changing)
  - Priority inversion: high-priority jobs can be blocked behind low-priority jobs
  - Schedulers are complex and difficult to control

How fair is lottery scheduling?

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 $\sqrt{n}$
- Number of lotteries needed for a client's first win has a geometric distribution
- Big picture answer: mostly accurate, but short-term inaccuracies are possible (see stride scheduling

Properties

- Probabilistically fair
- Starvation problem "solved"
  - Any client with a non-zero number of tickets will eventually win a lottery
- Responsive
  - Any changes to relative ticket allocations are immediately reflected in the next allocation decision

What is ticket transfer, why have it?

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 priority inversion problem (not the first solution however)

What is ticket inflation, why have it?

Ticket inflation

- Idea: 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

- Idea: set up an exchange rate with the base currency
  - Enables inflation just within a group
  - Simplifies mini-lotteries, such as for a mutex

Question

- What happens if a thread is I/O bound and regularly blocks before its quantum expires?

Question

- What happens if a thread is I/O bound and regularly blocks before its quantum expires?
  - Without adjustment, this implies that thread gets less than its share of the processor
  - How do they deal with this?

Question

- What happens if a thread is I/O bound and regularly blocks before its quantum expires?
  - Without adjustment, this implies that thread gets less than its share of the processor
  - How do they deal with this?
    - Compensation tickets: 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 with lottery scheduling

- 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
  - Are these statistically significant?

Problems with lottery scheduling

- 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
  - Are these statistically significant?
  - Probably are, which would imply that the lottery is biased or that there is a secondary force affecting the relative priority

Why didn't multimedia app achieve 3:2:1?

Why didn't multimedia app achieve 3:2:1?

- Biased results 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

Problems with lottery scheduling (2)

- 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)

Evaluation

- Are you happy with the evaluation? Workloads chosen?

Follow-on to lottery scheduling

- Stride scheduling
  - 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

Stride scheduling

- Algorithm (roughly): always pick the job with the lowest pass number. Update 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 x stride*) instead of just using the stride
- Overall result: it is far more accurate than lottery scheduling and error can be bounded absolutely instead of probabilistically

Rotating Staircase Deadline Scheduler (RDSL)

- "A starvation free, strict fairness O(1) scalable design with interactivity as good as the above restrictions can provide. There is no interactivity estimator, no sleep/run measurements and only simple fixed accounting. The design has strict enough a design and accounting that task behavior can be modelled and maximum scheduling latencies can be predicted by the virtual deadline mechanisms that manages run queues. The prime concern in this design is to maintain fairness at all costs determined by nice level, yet to maintain as good interactivity as can be allowed within the constraints of strict fairness."

  --- Con Kolivas, 2007

RDSL

- One parameter: the round-robin interval (RR_INTERVAL)
- One input: a thread priority
- Priority defines the levels at which each task can run
  - High priority tasks: **more** levels, **more** chances to run
  - Low priority tasks: **fewer** levels, **fewer** chances to run
- Tasks can run more at most a fixed amount of time per level
- Each level can also run for at most a fixed amount of time – ensures **bounded latency**

RDSL

- To begin a scheduling epoch:
  - Put all threads in a queue determined by their priority
  - Then:
    - 1. Run threads from the highest non-empty queue round-robin
    - 2. If a thread **blocks** or **yields**, it remains at that level
    - 3. If a thread runs out of its quota, it moves to the **next level down**
    - 4. If the level runs out of *its* quota, **all threads move to the next level down.**
    - 5. Continue until all quotas are exhausted or no threads are runnable, then restart another epoch

OS Scheduler

- Never-ending, evolving task for kernel developers
- RSDL → BFS → MuQSS
- …

Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code (2001)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code (2001)

- What kind of paper is this?
  - Describes a technique – a new way of tackling an old problem
    - Old problem: figure out what rules code is supposed to obey
    - New approach: "crowd source it" – look in the code for things that allude to rules and then use the statistics of their appearance to decide on right and wrong
  - The system described embodies the technique, but the contribution is in the technique, not the system

Overview

- Source code encapsulates programmer beliefs about a system
- Examples
  - Dereferencing a pointer indicates a belief that the pointer is non-null
  - Lock/unlock pairs indicate that all Locks followed by unlocks

Overview

- If you can extract beliefs automatically, then you can find bugs when code contradicts belief
  - MUST: Beliefs are directly implied by the code
  - MAY: beliefs are probabilistically implied
- Finds hundreds of bugs in operating systems
- Key insight is that we're going to try to find incorrect behavior without even knowing the correct behavior!

Methodology

- Construct belief sets based upon the code
- Contradictions in belief sets are errors (e.g., a belief that a value is NULL and then a belief that a value is non-NULL, via a dereference)
- Rules are expressed as templates
- Templates contain slots where you fill in variables, function, etc.
- A particular assignment is a slot instance
- Each slot instance has a particular belief set

Details

- If you can propagate beliefs across different functions/modules, then you greatly enhance the power of the checker
- Paper a bit fuzzy on exactly what you need to specify manually to write an effective checker
  - Do I need to figure out a big set of possible beliefs?

Static analysis

- Paper makes case for static analysis:
  - 1) scalable: once fixed cost of writing an analysis pass is paid, analysis is automatic
  - 2. precise: can say exactly what file and line led to an error
  - 3. immediate: does not require executing code

Evaluation

- Demonstrate the efficacy of this approach in real kernels
- Find real bugs
- False positive rate “OK”