

Lottery Scheduling

Scheduler Activations

I. Lottery Scheduling

Very general, proportional-share scheduling algorithm.

Problems with traditional schedulers:

- o Priority systems are ad hoc at best: highest priority always wins
- o “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)
- o Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- o Schedulers are complex and difficult to control

Lottery scheduling:

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

How fair is lottery scheduling?

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

Ticket Transfer: how to deal with dependencies

- o Basic idea: if you are blocked on someone else, give them your tickets
- o 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

Lecture 18

- o Very elegant solution to long-standing problem (not the first solution however)

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

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

Currencies: set up an exchange rate with the base currency

- o Enables inflation just within a group
- o 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.

- o Basic idea: if you complete fraction f of the quantum, your tickets are inflated by $1/f$ until the next time you win.
- o 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.
- o What if B alternates between $1/5$ and whole quanta?

Problems:

- o 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
- o 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)
- o 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...
- o 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?
- o 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)

- o Basic idea: make a deterministic version to reduce short-term variability
- o Mark time virtually using "passes" as the unit
- o 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.
- o Very regular: a job with priority p will run every $1/p$ passes.
- o Algorithm (roughly): always pick the job with the lowest pass number. Updates its pass number by adding its stride.
- o 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.

Lecture 18

- o Overall result: it is far more accurate than lottery scheduling and error can be bounded absolutely instead of probabilistically

II. Scheduler Activations

Goal: support fine-grained parallelism in a multiprogrammed environment.

Fine-grained parallelism model discussed: threads in a shared address space (others are also possible).

Threads can be implemented in two different ways:

- o kernel-implemented:
 - Kernel creates and dispatches threads.
 - Expensive: thread context switch involves crossing protection boundary to/from kernel.
 - Inflexible: can't easily customize the scheduling policy.
- o user-level:
 - Create one kernel thread for each processor, just use these like an OS would use processors to run the user-level threads.
 - Implement user-level threads entirely at the user-level in the runtime system: 1) Any user thread can run on any kernel thread. 2) Very fast, both for thread creation and context switch (no kernel calls in either case), and 3) Synchronization between user threads can be handled entirely at user-level. Can do things like spin-wait on locks.
 - Result: much faster thread primitives can support much finer-grained parallelism.

Problem with user-level threads: scheduling decisions being made independently by both kernel and user-level runtime system.

- o If a user thread executes a kernel call then the kernel thread becomes blocked: application loses a processor.
- o Kernel may deschedule a kernel thread at a bad time for the application, e.g. when the user thread being run by that kernel thread is in a critical section.
- o Application may suddenly need fewer threads and runs idle user threads on some of its kernel threads; the kernel doesn't know to deschedule those threads and give the processor to other applications.

Solution: design a protocol for passing scheduling information back and forth between the kernel and the runtime system.

Scheduler activation:

- o Vessel for running user threads (i.e. acts like a kernel thread). Can be thought of as a virtual processor in this respect.

Lecture 18

- o Notifies the user-level runtime system of interesting kernel events.
- o Provides space in the kernel for saving processor context of the currently running user thread when the thread is stopped by the kernel (e.g. for I/O or processor preemption to another application).

Kernel creates a new activation and does an *upcall* for one of the following reasons:

- o New processor available. Runtime picks a user thread to run on it.
- o Existing activation blocked (e.g. for I/O or page fault). Runtime picks another user thread to run on the new activation.
- o Activation unblocked and is now runnable. New activation includes processor context for *two* old activations: the newly unblocked one and the one that was preempted in order to make this notification. Why was it necessary to preempt a second activation?
 - To obtain a processor to run on. See fig 1 in paper, where black spots represent processors.
- o Activation lost its processor (to another application). Similar to unblocked activation case: new activation contains processor contexts for two old activations: the one whose processor was allocated to another application and the one whose processor is being used to run the new activation.

Runtime informs the kernel of the following events: $\text{No. runnable threads} = \text{No. processors} \pm 1$

- o Tells the kernel about transitions from needing another processor to not needing another processor and vice-versa.
- o Don't need to tell kernel about greater disparities between the two because that won't change the kernel's behavior.
- o All other runtime thread operations are strictly user-level.

Result: get the performance of user-level threads with the consistent behavior of kernel threads.

Some details:

- o User-level priority scheduling: may need to pull a lower priority user thread off of another activation. This is done by having the runtime tell the kernel to preempt the processor running the low priority user thread (only the kernel can preempt a processor). The preempted processor is used to do an upcall back to the application.
- o Dealing with preempted activations running in critical sections:
 - Runtime checks during an upcall whether the preempted/unblocked user thread was running in a critical section. Continues the user thread out of the critical section if so. Then puts the user thread on the appropriate queue.
 - Critical sections are detected by keeping a hash table of section begin/end addresses that are computed by placing special assembly instructions around critical sections in the object code and then post-processing the object code.

Lecture 18

3 key features about this paper:

- o Goal is to get user-level threads performance with the scheduling consistency provided by kernel-level threads in a multiprogramming environment.
- o The problem to solve: coordinating two independent thread schedulers: the kernel and the application runtime.
- o Scheduler activations used as a vessel to transmit information between the two as well as to provide virtual processors for running user-level threads.

Some flaws:

- o Authors wave their hands regarding the 5x slower upcall than kernel thread performance.
- o Only one application was tested. How would “ordinary” user-level threads perform relative to scheduler activations on other applications? Does the kernel’s scheduling policy affect the relative performance in any interesting ways?

A lesson: Export your functionality (in this case, threads) out of the kernel for improved performance and flexibility and figure out how to interact with the kernel “just enough” to allow the kernel to do its job as “traffic cop” among competing applications.