EECS 262a
Advanced Topics in Computer Systems
Lecture 11

Lottery Scheduling / SEDA
February 29th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

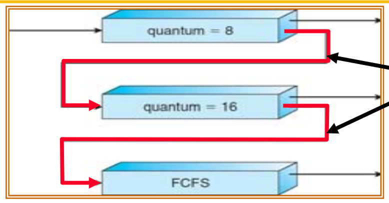
- Lottery Scheduling: Flexible Proportional-Share Resource Management
  Carl A. Waldspurger and William E. Weihl. Appears in *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 1994
- SEDA: An Architecture for WellConditioned, Scalable Internet Services
  Matt Welsh, David Culler, and Eric Brewer. Appears in *Proceedings of the 18th Symposium on Operating Systems Principles* (SOSP), 2001

- Thoughts?

Scheduling Review

- Scheduling: selecting a waiting process and allocating a resource (e.g., CPU time) to it
- First Come First Serve (FCFS/FIFO) Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time (quantum) when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)

Multi-Level Feedback Scheduling



Long-Running Compute tasks demoted to low priority

- A scheduling method for exploiting past behavior
  - First used in Cambridge Time Sharing System (CTSS)
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g., foreground – Round Robin, background – First Come First Serve
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc.)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Lottery Scheduling

- Very general, proportional-share scheduling algorithm
- Problems with traditional schedulers:
  - Priority systems are ad hoc at best: highest priority always wins, starvation risk
  - "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 with hard to understand behaviors

Lottery Scheduling

- Give each job some number of lottery tickets
- On each time slice, randomly pick a winning ticket and give owner the resource
- On average, resource fraction (CPU time) is proportional to number of tickets given to each job
- Tickets can be used for a wide variety of different resources (uniform) and are machine independent (abstract)

How to Assign Tickets?

- Priority determined by the number of tickets each process has:
  - Priority is the relative percentage of all of the tickets competing for this resource
- To avoid starvation, every job gets at least one ticket (everyone makes progress)

Lottery Scheduling Example

- Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

How Fair is Lottery Scheduling?

- If client has probability $p = (t/T)$ of winning, then the expected number of wins (from the binomial distribution) is np
  - *Probabilistically fair*
  - Variance of binomial distribution: $\sigma^2 = np(1 - p)$
  - Accuracy improves with $\sqrt{n}$
- Geometric distribution yields number of tries until first win
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses
- Big picture answer: mostly accurate, but short-term inaccuracies are possible
  - See (hidden) Stride Scheduling lecture slides for follow-on solution

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)

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
  - Also isolates from other groups
- 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

Lottery Scheduling 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 (e.g., X server)

Lottery Scheduling Problems

- 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

Stride Scheduling Algorithm

- 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* × *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

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

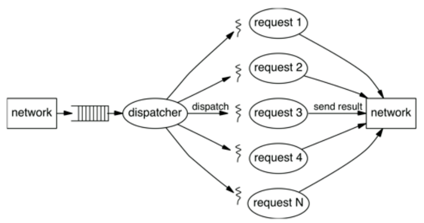
BREAK

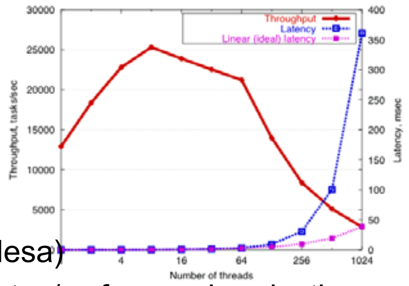
Project proposals due Friday at AOE– should have:

1. Motivation and problem domain
2. Description of what you are going to do and what is new about it
3. How you are going to do the evaluation (methodology, base case, …)
4. List of resources you need
5. List of ALL participants (with at least two people from CS262A)

Projects are supposed to be advancing the state of the art in some way, *not just redoing something that others have done*

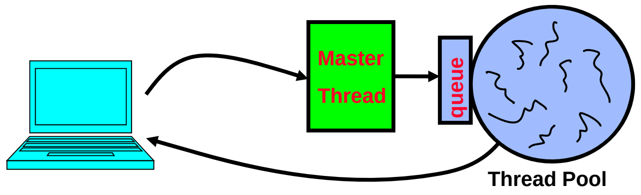
SEDA – Threads (Background)





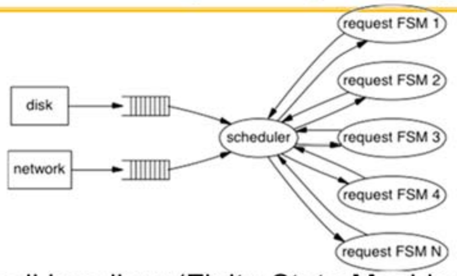
- The standard model (similar to Mesa)
- Concurrent threads with locks/mutex/… for synchronization
- Blocking calls
- May hold locks for long time
- Problems with more than 100 or so threads due to OS overhead (why?)
  – See SEDA graph of throughput vs. number of threads
- Strong support from OS, libraries, debuggers, ...

SEDA – Thread Pools (Background)



- **Problem with previous threads: Unbounded Threads**
  - **When a website becomes too popular throughput sinks and latency spikes**
- **Instead, allocate a bounded "pool" of threads (maximum level of multiprogramming) – crude form of load conditioning**
  - **Popular with some web servers**
- **But can't differentiate between short-running (static content) and long-running (dynamic content) threads**
  - **Also, highly unfair to clients stuck waiting for a thread – see Fig 12(a)**

SEDA – Events (Background)



- Lots of small handlers (Finite State Machines) that run to completion
  - No blocking allowed
- Basic model is event arrives and runs a handler
  - State is global or part of handler (not much in between)
- An event loop runs at the core waiting for arrivals, then calls handler
- No context switch, just procedure call

SEDA – Events (Background)

- Threads exist, but just run event loop ➔ handler ➔ event loop ➔ …
  - Stack trace is not useful for debugging!
  - Typically one thread per CPU (any more doesn't add anything since threads don't block)
  - Sometimes have extra threads for things that may block; e.g. OSs that only support synchronous disk reads
- Natural fit with finite-state machines (FSMs)
  - Arrows are handlers that change states
  - Blocking calls are split into two states (before and after the call)
- Allows very high concurrency
  - Multiplex 10,000 FSMs over a small number of threads
  - Approach used in telephony systems and some web servers

Cooperative Task Scheduling I

- **preemptive**: tasks may be interrupted at any time
  - Must use locks/mutex to get atomicity
  - May get preempted while holding a lock – others must wait until you are rescheduled – risk of priority inversion
  - Might want to differentiate short and long atomic sections (short should finish up work)

Cooperative Task Scheduling II

- **serial**: tasks run to completion
  - Basic event handlers, which are atomic
  - Not allowed to block
  - What if they run too long? (not much to do about that, could kill them; implies might be better for friendly systems)
  - Hard to support multiprocessors

Cooperative Task Scheduling III

- **cooperative**: tasks are not preempted, but do yield the processor
  - Can use stacks and make calls, but still interleaved
  - Yield points are not atomic: limits what you can do in an atomic section
  - Better with compiler help: is a call a yield point or not?
  - Hard to support multiprocessors
- Note: **preemption** is OK if it can't affect the current atomic section – easy way to achieve this is data partitioning! Only threads that access the shared state are a problem!
  - Can preempt for system routines
  - Can preempt to switch to a different process (with its own set of threads), but assumes processes don't share state

Implementing Split-Phase Actions –Threads

- Threads: Not too bad – just block until the action completes (synchronous)
- Assumes other threads run in the meantime
- Ties up considerable memory (full stack)
- Easy memory management: stack allocation/deallocation matches natural lifetime

Implementing Split-Phase Actions – Events

- Events: ***hard***
- Must store live state in a continuation (on the heap usually)
- Handler lifetime is too short, so need to explicitly allocate and deallocate later
- Scoping is bad too: need a multi-handler scope, which usually implies global scope
- Rips the function into two functions: before and after
- Debugging is ***hard***
- Evolution is ***hard***:
  - Adding a yielding call implies more ripping to do
  - Converting a non-yielding call into a yielding call is worse – every call site needs to be ripped and those sites may become yielding which cascades the problem

Atomic Split-Phase Actions (*Really Hard*)

- Threads – pessimistic: acquire lock and then block
- Threads – optimistic: read state, block, try write, retry if fail (and re-block!)
- Events – pessimistic: acquire lock, store state in continuation; later reply completes and releases lock
  - Seems hard to debug, what if event never comes? or comes more than once?
- Events – optimistic: read state, store in continuation ; apply write, retry if fail
- Basic problem: exclusive access can last a long time – hard to make progress
- General question: when can we move the lock to one side or the other?

One Strategy

- Structure as a sequence of actions that may or may not block (like cache reads)
- Acquire lock, walk through sequence, if block then release and start over
- If get all the way through then action was short (and atomic)
- This seems hard to automate!
  - Compiler would need to know that some actions mostly won't block or won't block the second time... and then also know that something can be retried without multiple side effects...
- Main conclusion (for me): compilers are the key to concurrency in the future

SEDA

- Problems to solve:
  1. Need a better way to achieve concurrency than just threads
  2. Need to provide graceful degradation
  3. Need to enable feedback loops that adapt to changing conditions
- The Internet:
  - Makes the concurrency higher
  - Requires high availability
  - Ensures that load will exceed the target range

Graceful Degradation

- Want throughput to increase linearly and then stay flat as you exceed capacity
- Want response time to be low until saturation and then linearly increase
- Want fairness in the presence of overload
- Almost no systems provide this
- Key is to drop work early or to queue it for later (threads have implicit queues on locks, sockets, etc.)
- Virtualization makes it harder to know where you stand!

Problems with Threads I (claimed)

- Threads limits are too small in practice (about 100)
  - Some of this is due to linear searches in internal data structures, or limits on kernel memory allocation
- Claims about locks, overhead, TLB and cache misses are harder to understand – don't seem to be fundamental over events
  - Do events use less memory? probably some but not 50% less
  - Do events have fewer misses? only if working set is smaller
  - Is it bad to waste VM for stacks? only with tons of threads

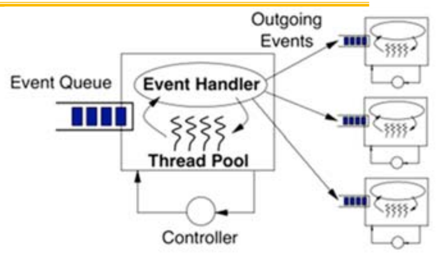
Problems with Threads II (claimed)

- Is there a fragmentation issue with stacks (lots of partially full pages)? probably to some degree (each stack needs at least one page
  - If so, we can move to a model with non-contiguous stack frames (leads to a different kind of fragmentation)
  - If so, we could allocate subpages (like FFS did for fragments), and thread a stack through subpages (but this needs compiler support and must recompile all libraries)
- Queues are implicit, which makes it hard to control or even identify the bottlenecks
- Key insight in SEDA:
  - No user allocated threads: programmer defines what can be concurrent and SEDA manages the threads
  - Otherwise no way to control the overall number or distribution of threads

Problems with Event-based Approach

- Debugging
- Legacy code
- Stack ripping

SEDA



- Use threads within a stage, events between stages
- Stages have explicit queues and explicit concurrency
- Threads (in a stage) can block, just not too often
- SEDA will add and remove threads from a stage as needed
- Simplifies modularity: queues decouple stages in terms of performance at some cost to latency
- Threads never cross stages, but events can be pass by value or pass by reference
- Stage scheduling affects locality – better to run one stage for a while than to follow an event through multiple stages
  - This should make up for the extra latency of crossing stages

Feedback Loops

- Works for any measurable property that has smooth behavior (usually continuous as well)
  – Property typically needs to be monotonic in the control area (else get lost in local minima/maxima)
- Within a stage: batching controller decides how many events to process at one time
  – Balance high throughput of large batches with lower latency of small batches – look for point where the throughput drops off
- Thread pool controller: find the minimum number of threads that keeps queue length low
- Global thread allocation based on priorities or queue lengths...
- Performance is very good, degrades more gracefully, and is more fair!
  – But huge dropped requests to maintain response time goal
  – However, can't really do any better than this...

Events versus Threads Revisited

- How does SEDA do split-phase actions?
- Intra-stage:
  – Threads can just block
  – Multiple threads within a stage, so shared state must be protected – common case is that each event is mostly independent (think HTTP requests)
- Inter-stage:
  – Rip action into two stages
  – Usually one-way: no return (equivalent to tail recursion) – this means that the continuation is just the contents of the event for the next stage
  – Loops in stages are harder: have to manually pass around the state
  – Atomicity is tricky too: how do you hold locks across multiple stages? generally try to avoid, but otherwise need one stage to lock and a later one to unlock

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?