Administrivia

- Project 1 due Friday noon
- If you need longer, email cs140-staff.
  - Put “extension” in the subject
  - Tell us where you are, and how much longer you need.
  - We will give short extensions to people who don’t abuse this

- Section Friday to go over project 2
- Project 2 Due Friday, Feb. 7 at noon
- Midterm following Monday, Feb. 10
- Midterm will be open book, open notes
  - Feel free to bring textbook, printouts of slides
  - Laptop computers or other electronic devices prohibited

Packet scheduling

- Differences from CPU scheduling
  - No preemption or yielding—must send whole packets
    - Thus, can’t send one bit at a time
  - But know how many bits are in each packet
    - Can see the future and know how long packet needs link

- What scheduling algorithm does this suggest?

FQ Algorithm

- Goal: Finish sending packets in same order as (impossible) BR

- Imagine a virtual clock with one tick per round of BR
  - Slows down as number of non-empty queues increases
  - Let $p_i^a = \text{length of packet } i \text{ in flow } a$
  - Let $S_i^a = \text{virtual time router starts transmitting packet } i \text{ of flow } a$
  - Transmission will finish at virtual time $F_i^a = S_i^a + p_i^a$

  - What is $S_i^a$?
    - If $i$ arrived before router finished sending packet $i - 1$ of same flow, then immediately after last bit of $i - 1$ (i.e., $S_i^a = F_{i-1}^a$)
    - If no current packets for this flow, then start transmitting as soon as packet arrives—call the arrival time $A_i^a$
  - Thus: $F_i^a = \max(F_{i-1}^a, A_i^a) + p_i^a$

Fair Queuing (FQ) [Demers]

- Digression: packet scheduling problem
  - Which network packet should router send next over a link?
  - Problem inspired some algorithms we will see today
  - Plus good to reinforce concepts in a different domain…

- For ideal fairness, would use bit-by-bit round-robin (BR)
  - Or send more bits from more important flows (flow importance can be expressed by assigning numeric weights)

Packet scheduling

- Differences from CPU scheduling
  - No preemption or yielding—must send whole packets
    - Thus, can’t send one bit at a time
  - But know how many bits are in each packet
    - Can see the future and know how long packet needs link

- What scheduling algorithm does this suggest? SJF

- Recall limitations of SJF from last lecture:
  - Can’t see the future
    - solved by packet length
  - Optimizes response time, not turnaround time
    - but these are the same when sending whole packets
  - Not fair

FQ Algorithm (cont)

- Algorithm: Send packet with lowest virtual finishing time $F$

  - For weighted fair queuing, assign each flow $a$ a weight $w^a$
    - Lets flow $a$ transmit $w^a$ bits per virtual clock tick
    - Now $F_i^a = \max(F_{i-1}^a, A_i^a) + p_i^a / w^a$

- Virtual time is a key technique in many schedulers
  - Increases monotonically but non-uniformly with real time
  - Saves us from recomputing values (e.g., $F_i^a$) as jobs come and go
  - Accommodates priority by slowing down for important jobs
Recall Limitations of BSD scheduler

- Mostly apply to < 2.6.23 Linux schedulers, too
- Hard to have isolation / prevent interference
  - Priorities are absolute
- Can’t donate CPU (e.g., to server on RPC)
- No flexible control
  - E.g., In monte carlo simulations, error is $1/\sqrt{N}$ after $N$ trials
  - Want to get quick estimate from new computation
  - Leave a bunch running for a while to get more accurate results
- Multimedia applications
  - Often fall back to degraded quality levels depending on resources
  - Want to control quality of different streams

Grace under load change

- Adding/deleting jobs affects all proportionally
- Example
  - 4 jobs, 1 ticket each, each job 1/4 of CPU
  - Delete one job, each remaining one gets 1/3 of CPU

- A little bit like priority scheduling
  - More tickets means higher priority
- But with even one ticket, won’t starve
  - Don’t have to worry about absolute priority problem
    (e.g., where adding one high-priority job starves everyone)

Lottery ticket transfer

- Can transfer tickets to other processes
- Perfect for IPC (Inter-Process Communication)
  - Client sends request to server
  - Client will block until server sends response
  - So temporarily donate tickets to server
- Also avoids priority inversion
- How do ticket donation and priority donation differ?

Lottery scheduling [Waldspurger’94]

- Inspired by economics & free markets
- Issue lottery tickets to processes
  - By analogy with FQ, #tickets expresses a process’s weight
  - Let $p_i$ have $t_i$ tickets
  - Let $T$ be total # of tickets, $T = \sum t_i$
  - Chance of winning next quantum is $t_i/T$.
  - Note tickets not used up by lottery (more like season tickets)
- Control expected proportion of CPU for each process
- Can also group processes hierarchically for control
  - Subdivide lottery tickets allocated to a particular process
  - Modeled as currencies, funded through other currencies

Compensation tickets

- What if process only uses fraction $f$ of quantum?
  - Say $A$ and $B$ have same number of lottery tickets
  - Proc. A uses full quantum, proc. B uses $f$ fraction
  - Each wins the lottery as often
  - $B$ gets fraction $f$ of $B$'s CPU time. No fair!
- Solution: Compensation tickets
  - Say $B$ uses fraction $f$ of quantum
  - Inflate $B$’s tickets by $1/f$ until it next wins CPU
  - E.g., if $B$ always uses half a quantum, it should get scheduled twice as often on average
  - Helps maximize I/O utilization
    (remember matrix multiply vs. grep from last lecture)
Limitations of lottery scheduling

- Unpredictable latencies
- Expected errors \( \sim \sqrt{na} \) for \( na \) allocations
  - E.g., process \( A \) should have had 1/3 of CPU yet after 1 minute has had only 19 seconds
- Useful to distinguish two types of error:
  - Absolute error – absolute value of \( A \)'s error (1 sec)
  - Relative error – \( A \)'s error considering only 2 processes, \( A \) and \( B \)
- Probability of getting \( k \) of \( n \) quanta is binomial distribution
  \[ \binom{n}{k} p^k (1 - p)^{n-k} \]
  - For large \( n \), binomial distribution approximately normal
  - Expected value is \( p \), Variance for a single allocation:
    \[ p(1-p)^2 + (1-p)p^2 = p(1-p)(1-p+p) = p(1-p) \]
  - Variance for \( n \) allocations = \( np(1-p) \), stddev \( \sim \sqrt{n} \)

Stride scheduling [Waldspurger'95]

- Idea: Apply ideas from weighted fair queuing
  - Deterministically achieve similar goals to lottery scheduling
- For each process, track:
  - Tickets – priority (weight) assigned by administrator
  - Stride \( \approx 1/\text{tickets} \) – speed of virtual time while process has CPU
  - Pass – cumulative virtual CPU time used by process
- Schedule process \( c \) with lowest pass
  - Then increase: \( c->\text{pass} + c->\text{stride} \)
- Note, can’t use floating point in the kernel
  - Saving FP regs too expensive, so make stride & pass integers
  - Let stride\(_1\) be largish integer (stride for 1 ticket)
  - Really set stride = stride\(_1\)/tickets

Stride scheduling example

Stride vs. lottery

- Stride offers many advantages of lottery scheduling
  - Good control over resource allocation
  - Can transfer tickets to avoid priority inversion
  - Use inflation/currencies for users to control their CPU fraction
- What are stride’s absolute & relative error?

Stride vs. lottery

- Stride offers many advantages of lottery scheduling
  - Good control over resource allocation
  - Can transfer tickets to avoid priority inversion
  - Use inflation/currencies for users to control their CPU fraction
- What are stride’s absolute & relative error?
- Stride Relative error always \( \leq 1 \) quantum
  - E.g., say \( A, B \) have same number of tickets
  - \( B \) has had CPU for one more time quantum than \( A \)
  - \( B \) will have larger pass, so \( A \) will get scheduled first
- Stride absolute error \( \leq n \) quanta if \( n \) processes in system
  - E.g., 100 processes each with 1 ticket
  - After 99 quanta, one of them still will not have gotten CPU

Simulation results

- Can clearly see \( \sqrt{n} \) factor for lottery
- Stride doing much better
Stride ticket transfer

- Want to transfer tickets like lottery
- Just recompute stride on transfer?

Scaling pass value

- Add some global variables
  - global-tickets – # tickets held by all runnable processes
  - global-stride – stride1 / global-tickets
  - global-pass – advances by global-stride each quantum
- On ticket transfer:
  - c->tickets = new_tickets;
  - c->stride = stride1 / c->tickets
  - int remain = c->pass - global_pass
  - remain *= new_stride / old_stride
  - c->pass = global_pass + remain

Sleep/wakeup

- Process might use only fraction \( f \) of quantum
  - Just increment c->pass += f * c->stride
- What if a process blocks or goes to sleep?
- Could do nothing—what’s wrong with this?

Stride error revisited

- Consider 101 procs w. allocations 100 : 1 : 1 : ... : 1
  - What happens?
- Process might use only fraction \( f \) of quantum
  - Just increment c->pass += f * c->stride
- What if a process blocks or goes to sleep?
- Could do nothing—what’s wrong with this?

Solution: Hierarchical stride scheduling

- Internal nodes have more tickets, so smaller strides
- Organize processes into a tree, schedule at each level
- Letting
  - Which is better?

Instead, keep advancing at global-pass rate

- On sleep: c->remain = c->pass - global_pass
- On wakeup: c->pass = global_pass + c->remain
- Slightly weird if global-tickets varies greatly
Stride error revisited

- Consider 101 procs w. allocations 100 : 1 : 1 : ... : 1
  - Cycle where high priority P0 gets CPU for 100 quanta
  - Then P1 ... P100 get one quanta each
- Another scheduler might give P0, P1, P0, P2, P0, ...
  - Which is better?

Greatly improves response time
Now for n procs, absolute error is $O(\log n)$, instead of $O(n)$

Solution: Hierarchical stride scheduling

- Borrowed Virtual Time (BVT)
  - Algorithm proposed by Duda & Cheriton in 1999
  - Goals:
    - Support mix of soft real-time and best-effort tasks
    - Simple to use (avoid 1,000,000 knobs to tweak)
    - Should be easy, efficient to implement
  - Idea: Run process w. lowest effective virtual time
    - $A_i = \text{actual virtual time} \times \text{effective virtual time}$
    - $\text{effective virtual time} = A_i - (\text{warp}_i \times W_i : 0)$
    - Special warp factor allows borrowing against future CPU time
    - Name of algorithm

BVT [Duda]

Process weights

- Each process $i$'s faction of CPU determined by weight $w_i$
  - $i$ should get $w_i / \sum w_j$ faction of CPU
  - So $w_i$ is seconds per virtual time tick while $i$ has CPU
- When $i$ consumes $t$ CPU time, track it: $A_i += t / w_i$
  - As with stride, pick some large $N$ (like stride_t)
  - Pre-compute $m_i = N / w_i$, then set $A_i += t \cdot m_i$
- Example: gcc (weight 2), bigsim (weight 1)
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, ...
  - Lots of context switches, not so good for performance
- Add in context switch allowance, $C$
  - Only switch from $i$ to $j$ if $E_j \leq E_i - C / w_i$
  - $C$ is real time ($\gg$ context switch cost), so must divide by $w_i$
  - Ignore $C$ if $j$ just became runnable... why?

Process weights

- Each process $i$'s faction of CPU determined by weight $w_i$
  - $i$ should get $w_i / \sum w_j$ faction of CPU
  - So $w_i$ is seconds per virtual time tick while $i$ has CPU
- When $i$ consumes $t$ CPU time, track it: $A_i += t / w_i$
  - As with stride, pick some large $N$ (like stride_t)
  - Pre-compute $m_i = N / w_i$, then set $A_i += t \cdot m_i$
- Example: gcc (weight 2), bigsim (weight 1)
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, ...
  - Lots of context switches, not so good for performance
- Add in context switch allowance, $C$
  - Only switch from $i$ to $j$ if $E_j \leq E_i - C / w_i$
  - $C$ is real time ($\gg$ context switch cost), so must divide by $w_i$
  - Ignore $C$ if $j$ just became runnable to avoid affecting response time
**BVT example**

- gcc has weight 2, bigsim weight 1, C = 2, no I/O
  - bigsim consumes virtual time at twice the rate of gcc
  - Procs always run for C time after exceeding other’s E

- As with stride, must lower priority after wakeup
  - Otherwise process with very low Ai would starve everyone

- Bound lag with Scheduler Virtual Time (SVT)
  - SVT is minimum Ai for all runnable threads j
  - When waking i from voluntary sleep, set Ai ← max(Ai, SVT)

- Note voluntary/involuntary sleep distinction
  - E.g., Don’t reset Ai to SVT after page fault
  - Faulting thread needs a chance to catch up
  - But do set Ai ← max(Ai, SVT) after socket read

- Also note Ai can never decrease
  - After short sleep, might have Ai > SVT, so max(Ai, SVT) = Ai
  - i never gets more than its fair share of CPU in long run

- gcc wakes up after I/O
  - Otherwise, would be at lower (blue) line and starve bigsim

- gcc’s Ai gets reset to SVT on wakeup
  - Otherwise, would be at lower (blue) line and starve bigsim

- mpeg player runs with −50 warp value
  - Always gets CPU when needed, never misses a frame
Google example

- Common queries 150 times faster than uncommon
  - Have 10-thread pool of threads to handle requests
  - Assign $W_i$ value sufficient to process fast query (say 50)

- Say 1 slow query, small trickle of fast queries
  - Fast queries come in, warped by 50, execute immediately
  - Slow query runs in background

- Say 1 slow query, but many fast queries
  - At first, only fast queries run
    - But SVT is bounded by $A_i$, of slow query thread $i$
  - Eventually Fast query thread $j$ gets $A_j = \max(A_j, SVT) = A_j$, and eventually $A_i - \text{warp}_j > A_i$.
  - At that point thread $i$ will run again, so no starvation

SMART Algorithm

- If most important ready task (ready task with best value-tuple) is conventional (not real-time), run it

- Consider all real-time tasks with better value-tuples than the best ready conventional task

- For each such RT task, starting from the best value-tuple
  - Can you run it without missing deadlines of tasks w. better value-tuples?
    - Yes? Add to schedulable set
    - Run task with earliest deadline in schedulable set

- Send signal to tasks that won’t meet their deadlines

SMART thread properties

- Application interface
  - priocntl (idtype t idtype, id t id, int cmd, ...);
  - Set two properties for each thread: priority & share
  - Real-time applications can specify constraints, where $constraint = \langle \text{deadline, estimated processing time} \rangle$

- Importance = (priority, BVFT) value-tuple
  - priority is parameter set by user or administrator
  - BVFT is Biased Virtual Finishing Time
    (like fair queuing, plus bias explained on next slide)

- To compare the importance of two threads
  - Priority takes absolute precedence
  - If same priority, earlier BVFT more important

SMART [Nieh]

- Proposed by Nieh & Lam in 1997

- Goals:
  - Support soft real-time constraints
  - Coexistence w. conventional workloads
  - User preferences (e.g., watching video while waiting for a compile means video lower priority; compiling in background during a video conference is the opposite)

- Key idea: Separate importance from urgency
  - Figure out which processes are important enough to run
  - Run whichever of these is most urgent

BVFT high-level overview

- Each task has weighted virtual time like BVT

- But system keeps a queue for each priority
  - BVT’s SVT is roughly replaced by queue virtual time
  - Try to maintain fairness within each queue
  - While across queues priority is absolute

- Bias factor is kind of like negative warp
  - VFT + Bias = BVFT
  - High bias means process can tolerate short-term unfairness
  - Though in long run proportion of CPU will still be fair
  - Any user interaction sets bias to 0
  - Real-time tasks have 0 bias

Current Linux

- Linux currently has “pluggable” scheduling [LWN]

- Global linked list of struct sched_class
  - Each sched_class has function ptrs implementing a scheduler
    - E.g., enqueue_task, pick_next_task, task_woken, ...
    - Each process’s task_struct has pointer to its sched_class

- Schedulers are in strict hierarchy
  - If sched_class.highest has runnable process, gets CPU
    - Otherwise, sched_class.highest->next, etc.

- Not easy to plug in schedulers w/o changing source
  - E.g., existing schedulers have dedicated fields in task_struct

- Default kernel has two schedulers:
  - Real-time (highest priority, not used unless set with chrt)
  - Completely Fair Scheduler [CFS]
CFS

- Quantum depends on # of runnable processes, determined by parameters set in /proc/sys/kernel:
  - `sched_latency_ns`: How often processes should run
  - `sched_min_granularity_ns`: Minimum quantum
  - Quantum \( = \max\left(\frac{\text{sched_latency}}{\#\text{runnable}}, \text{sched_min_granularity}\right)\)

- Keep stats in per-`proc` `sched_entity` structure:
  - `vruntime` is basically pass from the stride scheduler
  - Assumes nanosecond-granularity timer, simplifying things

- Extensions for hierarchical grouping with `cgroups`

Distributed scheduling

- Say you have a large system of independent nodes

- You want to run a job on a lightly loaded node
  - Unlike single-node scheduler, don’t know all machines’ loads
  - Too expensive to query each node for its load

- Instead, pick node at random
  - This is how lots of Internet services work

- Mitzenmacher: Then randomly pick one other one!
  - Send job to less loaded of two randomly sampled nodes
  - Result? Really close to optimal (with a few assumptions)
    - Exponential convergence \( \Rightarrow \) picking 3 doesn’t get you much

The universality of scheduling

- General problem: Let \( m \) requests share \( n \) resources
  - Always same issues: fairness, prioritizing, optimization

- Disk arm: which read/write request to do next?
  - Optimal: close requests \( \Rightarrow \) faster
  - Fair: don’t starve far requests

- Memory scheduling: whom to take page from?
  - Optimal: past=future? take from least-recently-used
  - Fair: equal share of memory

- Printer: what job to print?
  - People = fairness paramount: uses FIFO rather than SJF
  - Use “admission control” to combat long jobs

Postscript

- In principle, scheduling decisions can be arbitrary & shouldn’t affect program’s results
  - Good, since rare that “the best” schedule can be calculated

- In practice, schedule does affect correctness
  - Soft real time (e.g., mpeg or other multimedia) common
  - Or after 10s of seconds, users will give up on web server

- Unfortunately, algorithms strongly affect system throughput, turnaround time, and response time

- The best schemes are adaptive. To do absolutely best we’d have to predict the future.
  - Most current algorithms tend to give the highest priority to the processes that need the least CPU time
  - Scheduling has gotten increasingly \textit{ad hoc} over the years. 1960s papers very math heavy, now mostly "tweak and see"