**Administrivia**

- Project 1 due Thursday 4:15pm
  - Show up to lecture for free extension to midnight
  - SCPD can just watch lecture before midnight
- 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 Thursday, Feb. 4
- Midterm following Tuesday, Feb. 9
- Midterm will be open book, open notes
  - Feel free to bring textbook, printouts of slides
  - Laptop computers or other electronic devices prohibited

**Errata**

- Fair queuing *isn’t* SJF scheduling for networks
- SJF has three limitations
  - Can’t see the future
  - Optimizes response time but not turnaround time
  - Not fair
- The network setting does address the first two
- But SJF still unfair
  - Would starve long packets on the network
  - So obviously not same as fair queuing
- Today’s lecture will discuss CPU schedulers like FQ

**Linux 2.6 (< 2.6.23) Scheduler**

- Linux ≤ 2.4 scheduler had several drawbacks
  - $O(n)$ operations for $n$ processes (e.g., re-calculate “goodness” of all processes. Decaying $p_{estcpu}$ in BSD similarly $O(n)$.)
  - On SMPs: No affinity (bad for cache), global run-queue lock
- Linux 2.6 goal: Be $O(1)$ for all operations
- 140 Priority levels
  - 1–100 for real-time tasks (configured by administrator)
  - 101–140 for user tasks (depend on nice & behavior)
- Also keeps per-process 4-entry “load estimator”
  - How much CPU consumed in each of the last 4 seconds
  - Adjusts priority of user proc by ±5 based on behavior

**Linux task lists**

- Keeps one active/expired array pair per CPU
  - Avoids global lock and helps with affinity
  - SMP load balancer can move procs between CPUs
- Run highest-priority task in active array
  - After task uses quantum, place it in expired list
  - Swap expired/active pointers when active list empty
- Bitmap cache for empty/non-empty state of each list

**Recall Limitations of BSD scheduler**

- Mostly apply to Linux scheduler, 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

**Lottery scheduling [Waldspurger’94]**

- Inspired by economics & free markets
- Issue lottery tickets to processes
  - 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 lottery tickets not used up, more like season tickets
- Control avg. 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
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?

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
    - Inflates 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 $O(\sqrt{n_a})$ for $n_a$ allocations
    - E.g., process A should have had 1/3 of CPU yet after 1 minute has had only 19 seconds
    - Absolute error – absolute value of A’s error (1 sec)
    - Relative error – A’s error considering only 2 procs, A and B
- Prob. of getting $k$ of $n$ quanta is binomial distribution
  - $\binom{n}{k} p^k (1-p)^{n-k}$
  - $p = \text{fraction tickets owned, } \binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - 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 $= O(\sqrt{n})$

Stride scheduling [Waldspurger’95]

- Idea: Apply ideas from weighted fair queuing
  - Deterministically achieve similar goals to lottery scheduling
- For each process, track:
  - tickets – priority assigned by administrator
  - stride – roughly inverse of tickets
  - pass – roughly how much CPU time used
- 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 be largish integer (stride for 1 ticket)
  - Really set stride = stride1/tickets
- Latest linux scheduler (CFS) roughly reinvented idea
**Stride scheduling example**

- 3 tickets
- 2 tickets
- 1 ticket

\[ \text{stride}_1 = 6 \]

**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?

**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?

- Want to transfer tickets like lottery
- Just recompute stride on transfer?
- No! Would mean long latency
  - E.g., transfer 2 tickets to \( \Delta \) at time 0
  - Now \( \Delta \) has same priority as \( \Delta \)
  - But still waits 6 seconds to run
  - Very bad for IPC latency, mutexes, etc.
- Solution: Must scale remaining portion of pass by new # tickets
Scaling pass value

- Add some global variables
  - global-tickets – # tickets held by all runnable processes
  - global-stride – stride / global-tickets
  - global-pass – advances by global-stride each quantum

- On ticket transfer:
  ```c
  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?

- Could just revoke tickets while sleeping
  - Use negative ticket transfer to reduce # tickets to 0
  - But code on previous slide would require division by 0

- 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
  - What happens?

- Consider 101 procs w. allocations 100 : 1 : 1 : . . . : 1
  - Cycle where high priority $P_0$ gets CPU for 100 quanta
  - Then $P_1 \ldots P_{99}$ get one quanta each

- Another scheduler might give $P_0, P_1, P_0, P_2, P_0, \ldots$
  - Which is better?

- Solution: Hierarchical stride scheduling
  - Organize processes into a tree, schedule at each level
  - Internal nodes have more tickets, so smaller strides
  - Greatly improves response time
  - Now for $n$ procs, absolute error is $O(\log n)$, instead of $O(n)$
Hierarchical stride example

<table>
<thead>
<tr>
<th>Blue = Tickets</th>
<th>Red = Stride</th>
<th>Green = Pass values</th>
<th>Magenta = Quanta</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>2</td>
<td>512</td>
<td>2</td>
</tr>
<tr>
<td>161</td>
<td>10</td>
<td>512</td>
<td>168</td>
</tr>
<tr>
<td>29x623</td>
<td>29x631</td>
<td>29x623</td>
<td>29x631</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 7</td>
<td>26x719</td>
<td>26x727</td>
<td>26x711</td>
</tr>
<tr>
<td>Blue = Tickets</td>
<td>Red = Stride</td>
<td>Green = Pass values</td>
<td>Magenta = Quanta</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 7</td>
<td>26x719</td>
<td>26x727</td>
<td>26x711</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 7</td>
<td>26x719</td>
<td>26x727</td>
<td>26x711</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 7</td>
<td>26x719</td>
<td>26x727</td>
<td>26x711</td>
</tr>
</tbody>
</table>

BVT [Duda]

- **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
  - \( A_i \) - actual virtual time consumed by process \( i \)
  - \( \text{effective virtual time} E_i = A_i - (\text{warp}_i ? W_i : 0) \)
  - Special warp factor allows borrowing against future CPU time
    … hence name of algorithm

Process weights

- Each proc. \( i \)’s faction of CPU determined by weight \( w_i \)
  - Just like tickets in stride scheduling
  - \( i \) should get \( w_i / \sum w_j \) faction of CPU
- When \( i \) consumes \( t \) CPU time, charge it by \( A_i += t / w_i \)
  - As with stride, pick some large \( N \) (like stride1)
  - 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 (>> context switch cost), so must divide by \( w_i \)
  - Also, ignore \( C \) if \( j \) just became runnable… why?

Sleep/wakeup

- As with stride, must lower priority after wakeup
  - Otherwise process w. very low \( A_i \) would starve everyone
- Bound lag with Scheduler Virtual Time (SVT)
  - SVT is minimum \( A_j \) for all runnable threads \( j \)
  - When waking \( i \) from voluntary sleep, set \( A_i += \max(A_j, SVT) \)
- Note voluntary/involuntary sleep distinction
  - E.g., Don’t reset \( A_j \) to SVT after page fault
  - Faulting thread needs a chance to catch up
  - But do set \( A_i += \max(A_j, SVT) \) after socket read
- Also note \( A_j \) can never decrease
  - After short sleep, might have \( A_j > SVT \), so \( \max(A_j, SVT) = A_j \)
  - \( i \) never gets more than its fair share of CPU in long run

GCC wakes up after I/O

- GCC’s \( A_i \) gets reset to SVT on wakeup
  - Otherwise, would be at lower (blue) line and starve bigsim
Real-time threads

- Also want to support soft real-time threads
  - E.g., mpeg player must run every 10 clock ticks
- Recall \( E_i = A_i - (\text{warp}_i \cdot W_i : 0) \)
  - \( W_i \) is warp factor – gives thread precedence
  - Just give mpeg player \( i \) large \( W_i \) factor
  - Will get CPU whenever it is runnable
  - But long term CPU share won’t exceed \( w_i / \sum_j w_j \)
- But \( W_i \) only matters when \( \text{warp}_i \) is true
  - Can set it with a syscall, or have it set in signal handler
  - Also gets cleared if \( i \) keeps using CPU for \( L_i \) time
  - \( L_i \) limit gets reset every \( U_i \) time
  - \( L_i = 0 \) means no limit – okay for small \( W_i \) value

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

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 = \( \langle \text{deadline}, \text{estimated processing time} \rangle \)
- Importance = \( \langle \text{priority}, \text{BVFT} \rangle \) value-tuple
  - \text{priority} is parameter set by user or administrator
  - \text{BVFT} is Biased Virtual Finishing Time (c.f. fair queuing) \( \Rightarrow \) when quantum would end if process scheduled now
- To compare the importance of two threads
  - Priority takes absolute precedence
  - If same priority, earlier BVFT more important

Running warped

- mpeg player runs with \(-50\) warp value
  - Always gets CPU when needed, never misses a frame
**BVFT high-level overview**

- Each task has weighted “virtual time” as in 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

**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

**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 querying 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 (w. 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 = 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

**How to allocate resources**

- Space sharing (sometimes): split up. When to stop?
- Time-sharing (always): how long do you give out piece?
  - Pre-emptable (CPU, memory) vs. non-pre-emptable (locks, files, terminals)

**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 ad hoc over the years. 1960s papers very math heavy, now mostly “tweak and see”