**Administrivia**

- **Project 1 due right now**
  - But since you’re here, extension to 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 tomorrow to go over project 2**
- **Project 2 Due Thursday, Feb. 5**
- **Midterm following Tuesday, Feb. 10**
- **Midterm will be open book, open notes**
  - Feel free to bring textbook, printouts of slides
  - Laptop computers or other electronic devices prohibited

**Linux 2.6 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 procs by $\pm 5$ based on behavior

**Recall Limitations of BSD scheduler**

- Mostly apply to Linux scheduler, too
- Hard to have isolation / prevent interference
  - Priorities are absolute
- Can’t transfer priority (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]**

- Inspired by economics & free markets
- Issue lottery tickets to processes
  - Let $p_i$ have $t_i$ tickets, let $T = \sum t_i$
  - 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
  - Will block until server sends response
  - So temporarily donate tickets to server
- Also avoids priority inversion w. mutexes
  - Reflect true priority of a process
  - Which includes priority of processes waiting for it

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
  - If \( B \) uses \( f \) of quantum, inflate \( B \)'s tickets by \( 1/f \) until it next wins CPU
  - E.g., process that uses half of quantum gets schedules twice as often

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
  - \( \binom{n}{k} p^k (1 - p)^{n - k} \) [\( p \) = faction 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 = \( np(1 - p) \), \( \text{stddev} = O(\sqrt{n}) \)

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: \( c->pass += c->stride \)
- 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 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
- 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 $n$ procs 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?

- No!
  - Would mean long latency
  - E.g., transfer 2 tickets to $c$ at time 0
  - Now $c$ has same priority as $a$
  - 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 runable procs
  - global-stride = stride / global-tickets
  - global-pass = advances by global-stride each quantum

- On ticket transfer:
  - c->tickets = new_tickets; c->stride = stride / 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?
Sleep/wakeup

- Process might use only fraction $f$ of quantum
  - Just increment $c->pass += f \times c->stride$

- What if a process blocks or goes to sleep?
- Could do nothing—what’s wrong with this?
  - Will completely monopolize CPU when it wakes up with much smaller pass value than everyone else

- Could adjust # tickets 0 as on previous slide
  - But 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

- Say we have 101 procs w. allocations 100 : 1 : 1 : \ldots : 1
  - What happens?

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

  - Letting $P_0$ run for 100 quanta reduces context switches
  - But very bad for response time of other procs

Solution: Hierarchical stride scheduling
  - Organize processes into a tree
  - 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

- Stride1 = 1,024
- Blue = Tickets
- Red = Stride
- Green = Pass values
- Magenta = Quanta

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 $i$
  - effective virtual time $E_i = A_i - (\text{warp, } W_i = 0)$
  - 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 $w_i/\sum_j w_j$ 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 stride 1)
  - Pre-compute $m_i = N/w_i$, then set $A_i = t \cdot m_i$

- Example: gcc (weight 2), bigsim (weight 1)
  - 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 $E_j \leq E_i - C/w_i$

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 $A_i$ for all runnable threads $j$
  - When waking $i$ from voluntary sleep, set $A_i = \max(A_i, SVT)$

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

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

Real-time threads

- Also want to support soft real-time threads
  - E.g., mpeg player must run every 10 clock ticks

- Recall $E_i = A_i - (\text{warp} \cdot W_i : 0)$
  - $W_i$ 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 warp 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 $L_i$ time
  - $L_i = 0$ means no limit – okay for small $W_i$ value

Running warped

- mpeg player runs with ~50 warp value
  - Always gets CPU when needed, never misses a frame
warped thread hogging CPU

- mpeg goes into tight loop at time 5
- Exceeds $l_i$ at time 10, so $\text{warp}_i \leftarrow \text{false}$

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 $A_j = \max(A_j, SVT) = A_i$, and eventually $A_j - \text{warp}_j > A_i$
  - At that point thread $i$ will run again, so no starvation

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 = $\langle$ deadline, estimated processing time $\rangle$
  - Importance = $\langle$ priority, BVFT $\rangle$ value-tuple
    - priority is parameter set by user or administrator
    - 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

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 system load balancing

- Large system of independent nodes
- Want: run job on lightly loaded node
  - Querying each node too expensive
- Instead randomly pick one
  - This is how lots of Internet servers work
- Mitzenmacher: Then randomly pick one other one!
  - Send job to shortest run queue
  - Result? Really close to optimal (w. a few assumptions...)
  - Exponential convergence $\Rightarrow$ picking 3 doesn’t get you much

The universality of scheduling

- Used to let $m$ requests share $n$ resources
  - issues same: fairness, prioritizing, optimization
- Disk arm: which read/write request to do next?
  - Opt: close requests = faster
  - Fair: don’t starve far requests
- Memory scheduling: whom to take page from?
  - Opt: 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

<table>
<thead>
<tr>
<th>Space sharing</th>
<th>Time sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>(terminals CPU, disk arm locks)</td>
<td>(seats, professors rooms)</td>
</tr>
<tr>
<td>(disk blocks, pages, cache blocks)</td>
<td></td>
</tr>
</tbody>
</table>

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