**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. 3
- Midterm following Tuesday, Feb. 8
- 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 (< 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 \(p_{\text{estcpu}}\) 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 ±5 based on behavior

**Linux task lists**

<table>
<thead>
<tr>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td>[140]</td>
<td>[140]</td>
</tr>
</tbody>
</table>

- Processes organized into tasks lists at each priority
  - List heads stored in array
- **Keeps one active/expired array pair per CPU**
  - Avoids global lock and helps with affinity
  - SMP load balancer can move procs between CPUs

**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

**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?

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
- 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 procs, $A$ and $B$
- Prob. 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 $= O(\sqrt{n})$

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**

![Graph showing Pass Value vs. Time (quanta) for 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?

- No! Would mean long latency
  - E.g., transfer 2 tickets at time 0
  - Now has same priority as previous ticket
  - But still waits 6 seconds to run
  - Very bad for IPC latency, mutexes, etc.
- Solution: Must scale remaining portion of pass by new # tickets

- 3 tickets
- 2 tickets
- 1 ticket
- \( \text{stride}_1 = 6 \)
Scaling pass value

- Add some global variables
  - global-tickets – # tickets held by all runable processes
  - global-stride – stride/1 / global-tickets
  - global-pass – advances by global-stride each quantum

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

- 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?
  - Letting P0 run for 100 quanta reduces context switches
  - But then starving P0 for 100 quanta increase absolute error

- 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

- Stride \(l = 1.024\)
- Blue = Tickets
- Red = Stride
- Green = Pass values
- Magenta = Quanta

Process weights
- Each proc. \(i\)'s faction of CPU determined by weight \(w_i\)
  - Just like tickets in stride scheduling
  - \(i\) should get \(w_i / \sum w_i\) 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 \(l\))
  - 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 \(E_i < E_j - C / w_i\)
  - \(C\) is real time (\(\gg\) context switch cost), so must divide by \(w_i\)
  - Also, ignore \(C\) if \(j\) just became runable... 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 \(A_i \leftarrow \max(A_j, SVT)\)
- 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 \(A_i \leftarrow \max(A_j, SVT)\) after socket read
- Also 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

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

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 \(E_i\)

 gcc wakes up after I/O
- 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 \( E_i = A_i - (\text{warp}_i ? W_i : 0) \)
  - \( W_i \) 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 \( w_i / \sum w_j \)
- 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

Running warped

- 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 \( A_j = \max(A_j, SVT) = A_j \)
  - Eventually \( A_j > \text{warp}_j \times A_j \)
  - 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

WARPED thread hogging CPU

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

SMART thread properties

- Application interface
  - \( \text{priocntl (idtype, i} \text{dtype, id, int cmd, ...)} \)
  - Set two properties for each thread: priority & share
  - Real-time applications can specify \text{constraints}, where \text{constraint} = (\text{deadline, estimated processing time})
- Importance = (priority, BVFT) value-tuple
  - priority is parameter set by user or administrator
  - BVFT is Biased Virtual Finishing Time (c.f. fair queuing)
  \( \implies \) 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

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)

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 ⇒ picking 3 doesn’t get you much

CFS

- Quantum depends on # of runnable processes, determined by parameters set in /proc/sys/kernel:
  - sched_latency_us: How often processes should run
  - sched_min_granularity_us: Minimum quantum
    - This is BVT’s SVT (new procs get this vruntime)
    - Red-black tree orders probs by vruntime (O(log n))
  - Quantum = max (sched_latency_us, sched_min_granularity_us)
- Keep stats in per-proc sched_entity structure
  - vruntime is basically pass from the stride scheduler
    - Assumes nanosecond-granularity timer, simplifying things
  - This is how lots of Internet services work
- Keep per-runqueue values:
  - min_vruntime is BVT’s SVT (new procs get this vruntime)
  - Red-black tree orders probs by vruntime (O(log n))
  - Always run process with lowest vruntime
- Extensions for hierarchical grouping w. cgroups

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
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”