## 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 $\leq 2.4$** scheduler had several drawbacks
  - $O(n)$ operations for $n$ processes (e.g., re-calculate “goodness” of all processes. Decaying $p_{extcpu}$ 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

## Linux task lists

```
<table>
<thead>
<tr>
<th>priority</th>
<th>task lists</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[140]</td>
<td></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

## Linux task lists (continued)

- **Length of time quantum depends on priority**
- **Run highest-priority task in active array**
  - Keep track of partial quantum use on sleep
  - Once task uses entire quantum, place it in expired list
  - Swap expired/active pointers when active list empty
  - Adjust priority $\pm 5$ when putting task on expired list
- **Bitmap cache for empty/non-empty state of each list**
- **Next: look at some research schedulers**
  - … then we’ll see what Linux currently does

## 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?

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
  - Inflate \( B \)'s tickets by \( \frac{1}{f} \) until it next wins CPU
  - E.g., if \( B \) always uses half a quantum, it should gets 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
- 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 \( np \), 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 be largish integer (stride for 1 ticket)
  - Really set \( \text{stride} = \text{stride}_t / \text{tickets} \)
**Stride scheduling example**

- 3 tickets
- 2 tickets
- 1 ticket

Stride example:
- \( \text{stride} = 6 \)

**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?

**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 \( A \) at time 0
  - Now \( A \) has same priority as \( B \)
  - 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

- \( \text{stride} = 6 \)
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->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?

- Will completely monopolize CPU when it wakes up with much smaller pass value than everyone else

- 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?

- Letting $P_0$ run for 100 quanta reduces context switches
- But then starving $P_0$ 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

<table>
<thead>
<tr>
<th>Stride1 = 1,024</th>
<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>268</td>
<td>1536</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>512</td>
<td>256</td>
<td>384</td>
</tr>
<tr>
<td>128</td>
<td>512</td>
<td>1024</td>
<td>512</td>
<td>896</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>1024</td>
<td>512</td>
<td>1152</td>
</tr>
<tr>
<td>32</td>
<td>512</td>
<td>1024</td>
<td>512</td>
<td>1122</td>
</tr>
</tbody>
</table>

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
  - \( A_i - \text{actual virtual time consumed by process } i \)
  - effective virtual time \( E_i = A_i - (\text{warp } i \times W_i : 0) \)
  - 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 \( A_i \leftarrow \max(A_i, SVT) \)
- 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 \( A_i \leftarrow \max(A_i, SVT) \) after socket read
- Also note \( A_j \) can never decrease
  - After short sleep, might have \( A_j > SVT \), so \( \max(A_i, SVT) = A_i \)
  - \( i \) never gets more than its fair share of CPU in long run

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

- 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 $E_i = A_i - (warp_i \cdot W_i : 0)$
  - $W_i$ 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 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

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 = (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) $\Rightarrow$ 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
  - \( VFT + \text{Bias} = \text{BVFT} \)
  - 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 cfs)
  - 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 = \( \max \left( \frac{\text{sched_latency_ns}}{\text{runnable}}, \text{sched_min_granularity_ns} \right) \)
- Keep stats in per-proc sched_entity structure
  - vruntime is basically pass from the stride scheduler
  - Assumes nanosecond-granularity timer, simplifying things
- Keeps per-runqueue values:
  - min_vruntime is BVT’s SVT (new proc gets this vruntime)
  - Red-black tree orders proc by vruntime \( O(\log n) \)
  - Always run process with lowest vruntime
- Extensions for hierarchical grouping w. 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 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
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”