Adminstrivia

- 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
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)

```
Flow 1
Flow 2
Flow 3
Flow 4
```

Round-robin service
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?
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

- **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 $P_i^\alpha$ = length of packet $i$ in flow $\alpha$
  - Let $S_i^\alpha$ = virtual time router starts transmitting packet $i$ of flow $\alpha$
  - Transmission will finish at virtual time $F_i^\alpha = S_i^\alpha + P_i^\alpha$

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

- **Thus:** $F_i^\alpha = \max(F_{i-1}^\alpha, A_i^\alpha) + P_i^\alpha$
FQ Algorithm (cont)

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

  ![Diagram](image)

- **For weighted fair queuing**, assign each flow $\alpha$ a weight $w^\alpha$
  
  - Lets flow $\alpha$ transmit $w^\alpha$ bits per virtual clock tick
  
  - Now $F^\alpha_i = \max(F^\alpha_{i-1}, A^\alpha_i) + P^\alpha_i / w^\alpha$

- **Virtual time is a key technique in many schedulers**
  
  - Increases monotonically but non-uniformly with real time
  
  - Saves us from recomputing values (e.g., $F^\alpha_i$) 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
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, $T = \sum_i t_i$
  - 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
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?
  - Consider case of 1,000 equally important processes
  - With priority, no difference between 1 and 1,000 donations
  - With tickets, recipient amasses more and more tickets
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** $\sim \sqrt{n_a}$ 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 processes, $A$ and $B$
- Probability 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 $\sim \sqrt{n}$
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** $\approx 1/tickets$ – speed of virtual time while process has CPU
  - **pass** – cumulative virtual CPU time used by process

- **Schedule process $c$ with lowest pass**

- **Then increase:** $c->pass += c->stride$

- **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

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

- \(\triangle\) 3 tickets, stride=2
- \(\circ\) 2 tickets, stride = 3
- \(\square\) 1 ticket, 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?
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 recomputate stride on transfer?

3 tickets

\[
\text{stride} = 6
\]

1 ticket
2 tickets

\[
\frac{16}{38}
\]
• Want to transfer tickets like lottery
• Just recompute stride on transfer?
• **No!** Would mean long latency
  - E.g., transfer 2 tickets to at time 0
  - Now has same priority as
  - 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 – 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
  ```

```
new_stride
stride
global_pass pass
global_pass new_pass
remain
remain'
done
```
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?
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 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: $c->remain = c->pass - \text{global_pass}$
  - On wakeup: $c->pass = \text{global_pass} + c->remain$
  - Slightly weird if global-tickets varies greatly
Stride error revisited

- Consider 101 procs w. allocations 100 : 1 : 1 : . . . : 1
  - What happens?

- Another scheduler might give $P_0, P_1, P_0, P_2, P_0, \ldots$

- Which is better?

- 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 $O(n)$
Stride error revisited

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

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

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

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

Stride 1 = 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* $E_i = A_i - (\text{warp}_i ? W_i : 0)$
  - Special warp factor allows borrowing against future CPU time
    … hence name of algorithm
Process weights

- Each process $i$’s faction of CPU determined by weight $w_i$
  - $i$ should get $w_i / \sum_j w_j$ 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$_1$
  - 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 $w_i / \sum_{j} w_j$ 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$_1$)
  - 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 runable to avoid affecting response time
• 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 $E_i$
Sleep/wakeup

- As with stride, must lower priority after wakeup
  - Otherwise process with 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 $A_j$ 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 $A_i$ 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
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** \( E_i = A_i - (\text{warp}_i \ ? \ W_i : 0) \)
  - \( W_i \) 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 \( \frac{w_i}{\sum_j w_j} \)

- **Note** \( W_i \) 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
• 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 $A_j = max(A_j, SVT) = A_j$, and eventually $A_j - warp_j > A_i$.
  - 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 
    \[ \text{constraint} = \langle \text{deadline}, \text{estimated processing time} \rangle \]

- **Importance** = \( \langle \text{priority}, \text{BVFT} \rangle \) 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
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
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]
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

- Keeps per-runqueue values:
  - min_vruntime is BVT’s SVT (new procs get this vruntime)
  - Red-black tree orders procs 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 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 = 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”