The scheduling problem:
- Have $k$ jobs ready to run
- Have $n \geq 1$ CPUs that can run them

Which jobs should we assign to which CPU(s)?
Multiprocessor scheduling issues

- Must decide on more than which processes to run
  - Must decide on which CPU to run which process
- Moving between CPUs has costs
  - More cache misses, depending on architecture more TLB misses too
- Affinity scheduling—try to keep process/thread on same CPU

- But also prevent load imbalances
- Do cost-benefit analysis when deciding to migrate
Outline

1. Lottery Scheduling
2. Stride Scheduling
3. Virtual Time Scheduler
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 $\sim \sqrt{n}$
1. Lottery Scheduling
2. Stride Scheduling
3. Virtual Time Scheduler
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 $\approx \frac{1}{\text{tickets}}$ – 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 $\text{stride}_1$ be largish integer (stride for 1 ticket)
  - Really set $\text{stride} = \frac{\text{stride}_1}{\text{tickets}}$
Stride scheduling example

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

- 3 tickets, stride = 2
- 2 tickets, stride = 3
- 1 ticket, stride = 6
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
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
Outline

1. Lottery Scheduling
2. Stride Scheduling
3. Virtual Time Scheduler
Advanced scheduling with virtual time

- Many modern schedulers employ notion of *virtual time*
  - Idea: Equalize virtual CPU time consumed by different processes
  - Higher-priority processes consume virtual time more slowly
- Forms the basis of the current Linux scheduler, **CFS**
- Case study: Borrowed Virtual Time (BVT) [*Duda*]
- BVT runs process with 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 real seconds per virtual second that process $i$ has CPU
- When $i$ consumes $t$ CPU time, track it: $A_i \leftarrow t / w_i$
- 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 wall-clock time (>> context switch cost), so must divide by $w_i$
  - Ignore $C$ if $j$ just became runable...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 real seconds per virtual second that process $i$ has CPU
- When $i$ consumes $t$ CPU time, track it: $A_i += t/w_i$
- 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 wall-clock time (>> context switch cost), so must divide by $w_i$
  - Ignore $C$ if $j$ just became runnable 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
  - Processes run for \( C \) time after lines cross before context switch
Sleep/wakeup

- Must lower priority (increase $A_i$) 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

- Note: Even with SVT $A_i$ 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 time-critical tasks
  - 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 \( w_i / \sum_j w_j \)
- Note \( W_i \) only matters when \( \text{warp}_i \) is true
  - Can set \( \text{warp}_i \) 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
- 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 $\text{warp}_j \leftarrow \text{false}$
Common queries 150 times faster than uncommon
- Have 10-thread pool of threads to handle requests
- Assign $W_i$ a value sufficient to process fast query (e.g., 50)

Example 1: one slow query, small trickle of fast queries
- Fast queries come in, warped by 50, execute immediately
- Slow query runs in background
- Good for turnaround time

Example 2: one slow query, but many fast queries
- At first, only fast queries run
- But SVT is bounded by $A_i$ of slow query thread $i$
- Recall fast query thread $j$ gets $A_j = \max(A_j, SVT) = A_j$; eventually $SVT < A_j$ and a bit later $A_j - \text{warp}_j > A_i$.
- At that point thread $i$ will run again, so no starvation
Real-time scheduling

- Two categories:
  - *Soft real time*—miss deadline and CD will sound funny
  - *Hard real time*—miss deadline and plane will crash

- System must handle periodic and aperiodic events
  - E.g., processes A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if \( \sum \frac{CPU}{period} \leq 1 \) (not counting switch time)

- Variety of scheduling strategies
  - E.g., first deadline first
    (works if schedulable, otherwise fails spectacularly)