CS162
Operating Systems and
Systems Programming
Lecture 12

Scheduling 3:
Case Studies (Con’t), Realtime,
Starvation, Deadlock

February 28th, 2023
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Multi-Level Feedback Scheduling

- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    » Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    » e.g. foreground – RR, background – FCFS
    » Sometimes multiple RR priorities with quantum increasing exponentially
      (highest: 1ms, next: 2ms, next: 4ms, etc)

- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)
Recall: Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower priority value ⇒ higher priority (for realtime values)
  - Highest priority value ⇒ Lower priority (for nice values)
  - All algorithms O(1)
    » Timeslices/priorities/interactivity credits all computed when job finishes time slice
    » 140-bit bit mask indicates presence or absence of job at given priority level

- Two separate priority queues: “active” and “expired”
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped

- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into “Timeslice Granularity” chunks – round robin through priority
So, Does the OS Schedule Processes or Threads?

- Many textbooks use the “old model”—one thread per process
- Usually it's really: **threads** (e.g., in Linux)

- One point to notice: switching threads vs. switching processes incurs different costs:
  - Switch threads: Save/restore registers
  - Switch processes: Change active address space too!
    » Expensive
    » Disrupts caching

- Recall, However: Simultaneous Multithreading (or “Hyperthreading”)
  - Different threads interleaved on a cycle-by-cycle basis and can be in different processes (have different address spaces)
Multi-Core Scheduling

- Algorithmically, not a huge difference from single-core scheduling

- Implementation-wise, helpful to have per-core scheduling data structures
  - Cache coherence

- Affinity scheduling: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  - Cache reuse
Recall: Spinlocks for multiprocessing

- Spinlock implementation:

  ```
  int value = 0; // Free
  Acquire() {
    while (test&set(&value)) {}; // spin while busy
  }
  Release() {
    value = 0; // atomic store
  }
  ```

- Spinlock doesn’t put the calling thread to sleep—it just busy waits
  - When might this be preferable?
    - Waiting for limited number of threads at a barrier in a multiprocessing (multicore) program
    - Wait time at barrier would be greatly increased if threads must be woken inside kernel

- Every `test&set()` is a write, which makes value ping-pong around between core-local caches (using lots of memory!)
  - So – really want to use `test&test&set()`!

- As we discussed in Lecture 8, the extra read eliminates the ping-ponging issues:

  ```
  // Implementation of test&test&set():
  Acquire() {
    do {
      while(value); // wait until might be free
    } while (test&set(&value)); // exit if acquire lock
  }
  ```
Gang Scheduling and Parallel Applications

• When multiple threads work together on a multi-core system, try to schedule them together
  – Makes spin-waiting more efficient (inefficient to spin-wait for a thread that’s suspended)

• Alternative: OS informs a parallel program how many processors its threads are scheduled on (*Scheduler Activations*)
  – Application adapts to number of cores that it has scheduled
  – “Space sharing” with other parallel programs can be more efficient, because parallel speedup is often sublinear with the number of cores
Real-Time Scheduling

• Goal: Predictability of Performance!
  – We need to predict with confidence worst case response times for systems!
  – In RTS, performance guarantees are:
    » Task- and/or class centric and often ensured a priori
  – In conventional systems, performance is:
    » System/throughput oriented with post-processing (… wait and see …)
  – Real-time is about enforcing predictability, and does not equal fast computing!!!

• Hard real-time: for time-critical safety-oriented systems
  – Meet all deadlines (if at all possible)
  – Ideally: determine in advance if this is possible
    – Earliest Deadline First (EDF), Least Laxity First (LLF), Rate-Monotonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)

• Soft real-time: for multimedia
  – Attempt to meet deadlines with high probability
    – Constant Bandwidth Server (CBS)
Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (\(=\)release) times
- Tasks have deadlines (\(D\)) and known computation times (\(C\))
- Example Setup:
Example: Round-Robin Scheduling Doesn’t Work

Time

Missed deadline!!
Earliest Deadline First (EDF)

- Tasks periodic with period $P$ and computation $C$ in each period: $(P_i, C_i)$ for each task $i$
- Preemptive priority-based dynamic scheduling:
  - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. $D_{i}^{t+1} = D_{i}^{t} + P_i$ for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline

\[
\begin{align*}
T_1 &= (4,1) \\
T_2 &= (5,2) \\
T_3 &= (7,2)
\end{align*}
\]
EDF Feasibility Testing

• Even EDF won’t work if you have too many tasks
• For $n$ tasks with computation time $C$ and deadline $D$, a feasible schedule exists if:

$$\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1$$
• Midterm I results: Mean: 47.3, StdDev: 16.8, Min: 3.4, Max: 87.7
  – Yes, probably was too long!
  – Sorry about that!
• Project 1 Extension:
  – Wednesday March 1st
• Homework 3:
  – Due Tuesday 3/7
  – Can be done in Rust (if you want)
Ensuring Progress

• Starvation: thread fails to make progress for an indefinite period of time

• Starvation ≠ Deadlock because starvation could resolve under right circumstances
  – Deadlocks are unresolvable, cyclic requests for resources

• Causes of starvation:
  – Scheduling policy never runs a particular thread on the CPU
  – Threads wait for each other or are spinning in a way that will never be resolved

• Let’s explore what sorts of problems we might encounter and how to avoid them…
Strawman: Non-Work-Conserving Scheduler

- A *work-conserving* scheduler is one that does not leave the CPU idle when there is work to do.

- A non-work-conserving scheduler could trivially lead to starvation.

- In this class, we’ll assume that the scheduler is work-conserving (unless stated otherwise).
Strawman: Last-Come, First-Served (LCFS)

- Stack (LIFO) as a scheduling data structure
  - Late arrivals get fast service
  - Early ones wait – extremely unfair
  - In the worst case – starvation

- When would this occur?
  - When arrival rate (offered load) exceeds service rate (delivered load)
  - Queue builds up faster than it drains

- Queue can build in FIFO too, but “serviced in the order received”…
Is FCFS Prone to Starvation?

- If a task never yields (e.g., goes into an infinite loop), then other tasks don’t get to run
- Problem with all non-preemptive schedulers…
  - And early personal OSes such as original MacOS, Windows 3.1, etc
Is Round Robin (RR) Prone to Starvation?

• Each of $N$ processes gets $\sim 1/N$ of CPU (in window)
  – With quantum length $Q$ ms, process waits at most $(N-1)*Q$ ms to run again
  – So a process can’t be kept waiting indefinitely

• So RR is fair in terms of waiting time
  – Not necessarily in terms of throughput… (if you give up your time slot early, you don’t get the time back!)
Is Priority Scheduling Prone to Starvation?

- Recall: Priority Scheduler always runs the thread with highest priority
  - Low priority thread might never run!
  - Starvation…

- But there are more serious problems as well…
  - Priority inversion: even high priority threads might become starved
Priority Inversion

• At this point, which job does the scheduler choose?
• Job 3 (Highest priority)
Priority Inversion

- Job 3 attempts to acquire lock held by Job 1
Priority Inversion

- At this point, which job does the scheduler choose?
- Job 2 (Medium Priority)
- Priority Inversion
Priority Inversion

- Where high priority task is blocked waiting on low priority task
- Low priority one **must** run for high priority to make progress
- Medium priority task can starve a high priority one

- When else might priority lead to starvation or “live lock”?
One Solution: Priority Donation/Inheritance

- Job 3 temporarily grants Job 1 its “high priority” to run on its behalf
One Solution: Priority Donation/Inheritance

- Job 3 temporarily grants Job 1 its “high priority” to run on its behalf

Priority 3
<table>
<thead>
<tr>
<th>Priority 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority 1</td>
</tr>
</tbody>
</table>

Job 3
<table>
<thead>
<tr>
<th>Blocked on Acquire</th>
</tr>
</thead>
</table>

Job 1
| Release() |

Job 2
| Locked |

• Job 3 temporarily grants Job 1 its “high priority” to run on its behalf
One Solution: Priority Donation/Inheritance

- Job 1 completes critical section and releases lock
- Job 3 acquires lock, runs again
- How does the scheduler know?
Case Study: Martian Pathfinder Rover

- July 4, 1997 – Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover
  - Novel delivery mechanism: inside air-filled balloons bounced to stop on the surface from orbit!
- And then…a few days into mission…:
  - Multiple system resets occur to realtime OS (VxWorks)
  - System would reboot randomly, losing valuable time and progress
- Problem? Priority Inversion!
  - Low priority task grabs mutex trying to communicate with high priority task:
    - Realtime watchdog detected lack of forward progress and invoked reset to safe state
      » High-priority data distribution task was supposed to complete with regular deadline
- Solution: Turn priority donation back on and upload fixes!
- Original developers turned off priority donation (also called priority inheritance)
  - Worried about performance costs of donating priority!
Are SRTF and MLFQ Prone to Starvation?

- In SRTF, long jobs are starved in favor of short ones
  - Same fundamental problem as priority scheduling
- MLFQ is an approximation of SRTF, so it suffers from the same problem
Cause for Starvation: Priorities?

• The policies we’ve studied so far:
  – Always prefer to give the CPU to a prioritized job
  – Non-prioritized jobs may never get to run

• But priorities were a means, not an end

• Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  – Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  – Give the interactive ones enough CPU to respond to an input and wait (on those slow humans)
  – Let the CPU bound ones grind away without too much disturbance
Recall: Changing Landscape…

Bell’s Law: New computer class every 10 years

Number crunching, Data Storage, Massive Inet Services, ML, …

Productivity, Interactive

Streaming from/to the physical world

The Internet of Things!

Computers Per Person

1:10^6

1:10^3

1:1

10^3:1
Changing Landscape of Scheduling

• Priority-based scheduling rooted in “time-sharing”
  – Allocating precious, limited resources across a diverse workload
    » CPU bound, vs interactive, vs I/O bound
• 80’s brought about personal computers, workstations, and servers on networks
  – Different machines of different types for different purposes
  – Shift to fairness and avoiding extremes (starvation)
• 90’s emergence of the web, rise of internet-based services, the data-center-is-the-computer
  – Server consolidation, massive clustered services, huge flashcrowds
  – It’s about predictability, 95th percentile performance guarantees
DOES PRIORITIZING SOME JOBS NECESSARILY STARVE THOSE THAT AREN’T PRIORITIZED?
Key Idea: Proportional-Share Scheduling

• The policies we’ve studied so far:
  – *Always prefer to give the CPU to a prioritized job*
  – Non-prioritized jobs may never get to run

• Instead, we can share the CPU *proportionally*
  – Give each job a share of the CPU according to its priority
  – Low-priority jobs get to run less often
  – But all jobs can at least make progress (no starvation)
Recall: Lottery Scheduling

- Given a set of jobs (the mix), provide each with a share of a resource
  - e.g., 50% of the CPU for Job A, 30% for Job B, and 20% for Job C
- Idea: Give out tickets according to the proportion each should receive,
- Every quantum (tick): draw one at random, schedule that job (thread) to run
Lottery Scheduling: Simple Mechanism

- $N_{ticket} = \sum N_i$
- Pick a number $d$ in $1 \ldots N_{ticket}$ as the random “dart”
- Jobs record their $N_i$ of allocated tickets
- Order them by $N_i$
- Select the first $j$ such that $\sum N_i$ up to $j$ exceeds $d$. 
Unfairness

- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%,
  \[ U = \frac{\text{finish time of first}}{\text{finish time of last}} \]
- As a function of run time
Stride Scheduling

• Achieve proportional share scheduling without resorting to randomness, and overcome the “law of small numbers” problem.

• “Stride” of each job is $\frac{\text{big}\#W}{N_i}$
  – The larger your share of tickets, the smaller your stride
  – Ex: $W = 10,000$, $A=100$ tickets, $B=50$, $C=250$
  – A stride: 100, B: 200, C: 40

• Each job has a “pass” counter

• Scheduler: pick job with lowest pass, runs it, add its stride to its pass

• Low-stride jobs (lots of tickets) run more often
  – Job with twice the tickets gets to run twice as often

• Some messiness of counter wrap-around, new jobs, …
Linux Completely Fair Scheduler (CFS)

- Goal: Each process gets an equal share of CPU
  - $N$ threads “simultaneously” execute on $\frac{1}{N}$ of CPU
  - The model is somewhat like simultaneous multithreading – each thread gets $\frac{1}{N}$ of the cycles

- In general, can’t do this with real hardware
  - OS needs to give out full CPU in time slices
  - Thus, we must use something to keep the threads roughly in sync with one another

Model: “Perfectly” subdivided CPU:
Linux Completely Fair Scheduler (CFS)

• Basic Idea: track CPU time per thread and schedule threads to match up average rate of execution

• Scheduling Decision:
  – “Repair” illusion of complete fairness
  – Choose thread with minimum CPU time
  – Closely related to Fair Queueing

• Use a heap-like scheduling queue for this…
  – $O(\log N)$ to add/remove threads, where $N$ is number of threads

• Sleeping threads don’t advance their CPU time, so they get a boost when they wake up again…
  – Get interactivity automatically!
Linux CFS: Responsiveness/Starvation Freedom

In addition to fairness, we want **low response time** and starvation freedom

– Make sure that everyone gets to run at least a bit!

**Constraint 1: Target Latency**

– Period of time over which every process gets service

– Quanta = Target_Latency / n

**Target Latency: 20 ms, 4 Processes**

– Each process gets 5ms time slice

**Target Latency: 20 ms, 200 Processes**

– Each process gets 0.1ms time slice (!!!)

– Recall Round-Robin: large context switching overhead if slice gets to small
Linux CFS: Throughput

• Goal: Throughput
  – Avoid excessive overhead
• Constraint 2: Minimum Granularity
  – Minimum length of any time slice

• Target Latency 20 ms, Minimum Granularity 1 ms, 200 processes
  – Each process gets 1 ms time slice
Aside: Priority in Unix – Being Nice

• The industrial operating systems of the 60s and 70’s provided priority to enforced desired usage policies.
  – When it was being developed at Berkeley, instead it provided ways to “be nice”.
• nice values range from -20 to 19
  – Negative values are “not nice”
  – If you wanted to let your friends get more time, you would nice up your job
• Scheduler puts higher nice-value tasks (lower priority) to sleep more …
  – In O(1) scheduler, this translated fairly directly to priority (and time slice)
• How does this idea translate to CFS?
  – Change the rate of CPU cycles given to threads to change relative priority
Linux CFS: Proportional Shares

• What if we want to give more CPU to some and less to others in CFS (proportional share)?
  – Allow different threads to have different rates of execution (cycles/time)
• Use weights! Key Idea: Assign a weight $w_i$ to each process $i$ to compute the switching quanta $Q_i$
  – Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$
  – Weighted Share: $Q_i = \left(\frac{w_i}{\sum_p w_p}\right) \cdot \text{Target Latency}$
• Reuse nice value to reflect share, rather than priority,
  – Remember that lower nice value $\Rightarrow$ higher priority
  – CFS uses nice values to scale weights exponentially: Weight=$1024/(1.25)^{\text{nice}}$
    » Two CPU tasks separated by nice value of 5 $\Rightarrow$
      Task with lower nice value has 3 times the weight, since $(1.25)^5 \approx 3$
• So, we use “Virtual Runtime” instead of CPU time
Example: Linux CFS: Proportional Shares

• Target Latency = 20ms
• Minimum Granularity = 1ms
• Example: Two CPU-Bound Threads
  – Thread A has weight 1
  – Thread B has weight 4
• Time slice for A? 4 ms
• Time slice for B? 16 ms
**Linux CFS: Proportional Shares**

- Track a thread's *virtual* runtime rather than its true physical runtime
  - Higher weight: Virtual runtime increases more slowly
  - Lower weight: Virtual runtime increases more quickly
- Scheduler's Decisions are based on Virtual CPU Time
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - O(log N) time to perform insertions/deletions
    » Cache the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).
Choosing the Right Scheduler

<table>
<thead>
<tr>
<th>I Care About:</th>
<th>Then Choose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Throughput</td>
<td>FCFS</td>
</tr>
<tr>
<td>Avg. Response Time</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>I/O Throughput</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>Fairness (CPU Time)</td>
<td>Linux CFS</td>
</tr>
<tr>
<td>Fairness – Wait Time to Get CPU</td>
<td>Round Robin</td>
</tr>
<tr>
<td>Meeting Deadlines</td>
<td>EDF</td>
</tr>
<tr>
<td>Favoring Important Tasks</td>
<td>Priority</td>
</tr>
</tbody>
</table>
A Final Word On Scheduling

• When do the details of the scheduling policy and fairness really matter?
  – When there aren’t enough resources to go around

• When should you simply buy a faster computer?
  – (Or network link, or expanded highway, or …)
  – One approach: Buy it when it will pay for itself in improved response time
    » Perhaps you’re paying for worse response time in reduced productivity, customer angst, etc…
    » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization ⇒ 100%

• An interesting implication of this curve:
  – Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  – Argues for buying a faster X when hit “knee” of curve
Deadlock: A Deadly type of Starvation

• Starvation: thread waits indefinitely
  – Example, low-priority thread waiting for resources constantly in use by high-priority threads

• Deadlock: circular waiting for resources
  – Thread A owns Res 1 and is waiting for Res 2
  – Thread B owns Res 2 and is waiting for Res 1

• Deadlock ⇒ Starvation but not vice versa
  – Starvation can end (but doesn’t have to)
  – Deadlock can’t end without external intervention
Example: Single-Lane Bridge Crossing

CA 140 to Yosemite National Park
Bridge Crossing Example

• Each segment of road can be viewed as a resource
  – Car must own the segment under them
  – Must acquire segment that they are moving into
• For bridge: must acquire both halves
  – Traffic only in one direction at a time

• Deadlock: Shown above when two cars in opposite directions meet in middle
  – Each acquires one segment and needs next
  – Deadlock resolved if one car backs up (preempt resources and rollback)
    » Several cars may have to be backed up
• Starvation (not Deadlock):
  – East-going traffic really fast ⇒ no one gets to go west
Conclusion

- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Realtime Schedulers such as EDF
  - Guaranteed behavior by meeting deadlines
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
- Linux CFS Scheduler: Fair fraction of CPU
  - Approximates an “ideal” multitasking processor
  - Practical example of “Fair Queueing”
- Deadlock: circular waiting for resources
  - A form of starvation (indefinite stalling) that will never resolve