Scheduling 2:
Classic Policies (Con’t), Case Studies,
Realtime, Starvation

February 27th, 2024
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: SRTF Example continued:

| Disk Utilization | ~90% but lots of wakeups! |
| Disk Utilization: | 9/201 ~ 4.5% |

Disk Utilization: 90%

SRTF

predicted burst length

\[ \tau_n = \alpha \tau_{n-1} + (1-\alpha) \tau_{n-1} \]

with \(0 < \alpha < 1\)

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: hard to predict job’s runtime even for non-malicious users
- Bottom line, can’t really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can’t do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn’t help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts: Let \(t_{n-1}, t_{n-2}, t_{n-3}, \ldots\) be previous CPU burst lengths. Estimate next burst \(\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)\)
  - Function \(f\) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance: exponential averaging
    - \[ \tau_n = \alpha \tau_{n-1} + (1-\alpha) \tau_{n-1} \] with \(0 < \alpha < 1\)
How to Handle Simultaneous Mix of Diff Types of Apps?

- Consider mix of interactive and high throughput apps:
  - How to best schedule them?
  - How to recognize one from the other?
  - Do you trust app to say that it is “interactive”?
  - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst Time (observed) useful to decide which application gets CPU time?
- Short Bursts → Interactivity → High Priority?
- Assumptions encoded into many schedulers:
  - Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  - Apps that compute a lot should get low(er?) priority, since they won’t notice intermittent bursts from interactive apps
- Hard to characterize apps:
  - What about apps that sleep for a long time, but then compute for a long time?
  - Or, what about apps that must run under all circumstances (say periodically)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    » serve all from highest priority, then next priority, etc.
  - Time slice:
    » each queue gets a certain amount of CPU time
    » e.g., 70% to highest, 20% next, 10% lowest

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)

Countermeasure:

- user action that can foil intent of the OS designers
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job’s priority high
  - Of course, if everyone did this, wouldn’t work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    » Put in printf’s, ran much faster!
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

Linux O(1) Scheduler

- Lots of ad-hoc heuristics
  - Try to boost priority of I/O-bound tasks
  - Try to boost priority of starved tasks

O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    » p->sleep_avg = sleep_time – run_time
    » Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    » Earned when a task sleeps for a “long” time
    » Spend when a task runs for a “long” time
    » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
  - However, “interactive tasks” get special dispensation
    » To try to maintain interactivity
    » Placed back into active queue, unless some other task has been starved for too long…
- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    » SCHED_FIFO: preempts other tasks, no timeslice limit
    » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same 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) but can be task groups (also Linux)
- Note: 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)
Administrivia

- Midterm 1 results: Mean: 52.4, StdDev: 15.0, Min: 9.6, Max: 93.2!
- Project 1 due tomorrow (Wednesday, 2/28)
  - Code and final report
- Also due Tomorrow: Peer evaluations
  - These are a required mechanism for evaluating group dynamics
  - Project scores are a zero-sum game
    - In the normal/best case, all partners get the same grade
    - In groups with issues, we may take points from non-participating group members and give them to participating group members!
- Homework 3:
  - Due Tuesday 3/5
  - Can be done in Rust (if you want!)

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, branch prediction
  - Example for O(1) scheduler: 1 set of queues/core with background rebalancing

Recall: Spinlocks for multiprocessing

- Spinlock implementation:
  ```c
  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!)

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)
  - Multiple phases of parallel and serial execution
- Additionally: 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

- 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. \(b_{i+1} = D_i + P_i\) for each task!)
  - The scheduler always schedules the active task with the closest absolute 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. \(b_{i+1} = D_i + P_i\) for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline
**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:
  \[
  \sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1
  \]

**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 \((N-1)\times Q\) 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
Priority 2
Priority 1

Job 3
Job 1
Job 2

Blocked on Acquire
Release()

One Solution: Priority Donation/Inheritance

- Job 1 completes critical section and releases lock
- Job 3 acquires lock, runs again
- How does the scheduler know?

Priority 3
Priority 2
Priority 1

Job 1
Job 3
Job 2

Acquire()

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
- Computers Per Person
  - 1:10^6
  - 1:10^3
  - 1:1
  - 10^5:1

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

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)
Lottery Scheduling

- Simple Idea:
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example (Cont.)

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

<table>
<thead>
<tr>
<th># short jobs</th>
<th>% of CPU each short jobs gets</th>
<th>% of CPU each long jobs gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>0/2</td>
<td>N/A</td>
<td>50%</td>
</tr>
<tr>
<td>2/0</td>
<td>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>10/1</td>
<td>9.9%</td>
<td>0.99%</td>
</tr>
<tr>
<td>1/10</td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

- What if too many short jobs to give reasonable response time?
  - If load average is 100, hard to make progress
  - One approach: log some user out

Lottery Scheduling: Simple Mechanism

- $N_{\text{ticket}} = \sum N_i$
- Pick a number $d$ in $1 \ldots N_{\text{ticket}}$ 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%, $U =$ finish time of first / 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, …

**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?
- **Priority Inversion**
  - A higher-priority task is prevented from running by a lower-priority task
  - Often caused by locks and through the intervention of a middle-priority task
- **Proportional Share Scheduling**
  - 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)