Operating Systems

07. Process Scheduling

Paul Krzyzanowski
Rutgers University
Spring 2015

Running more than one process

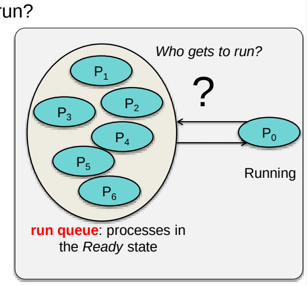
- Batch systems
  – Run one job. When it finishes, run the next one, …
- Cooperative multitasking
  – Run a process until it makes a system call
    ⇒ transfers control to the OS
  – OS can then decide to context switch and run another process
- Preemptive multitasking
  – OS programs a timer to generate an interrupt
  – Interrupt gives control back to the OS
    ⇒ decides whether to context switch

Process Scheduler

We have multiple tasks *ready* to run.
Which one should get to run?

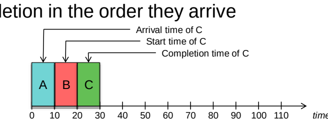
Scheduling algorithm:
– **Policy**: Makes the decision of who gets to run

Dispatcher:
– **Mechanism** to do the context switch



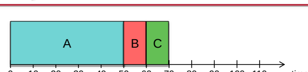
First Come, First Served (FCFS)

- Run jobs to completion in the order they arrive
- Sounds fair?



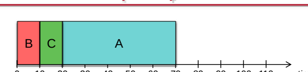
- Turnaround time: Time to complete a job since submitting it
- Turnaround time = $T_{completion} - T_{arrival}$
- Assume A, B, & C arrive at around the same time
  $T_{turnaround}(A) = 10$
  $T_{turnaround}(B) = 20$
  $T_{turnaround}(C) = 30$
  $T_{turnaround}(average) = (10+20+30) \div 3 = 20$

First Come, First Served



- What if A was a long-running job?
  $T_{turnaround}(A) = 50$
  $T_{turnaround}(B) = 60$
  $T_{turnaround}(C) = 70$
  $T_{turnaround}(average) = (50+60+70) \div 3 = 60$

Shortest Job First (SJF)

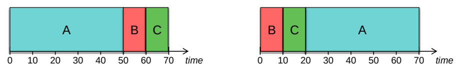


- Let shortest jobs run first ⇒ optimizes turnaround time
  $T_{turnaround}(B) = 10$
  $T_{turnaround}(C) = 20$
  $T_{turnaround}(A) = 70$
  $T_{turnaround}(average) = (10+20+70) \div 3 = 33.333$ vs. 60
- 1.8x better than FCFS! (in this example)
- But if *B* and *C* arrive a bit after *A*, we're still out of luck

© 2014-2015 Paul Krzyzanowski

Response time

- FCFS and SJF: non-preemptive schedulers
  - One job hold up all others!
- Let's consider response time
  - Response time = delay before a job starts to run
  - *Response time* = $T_{arrival} - T_{run}$

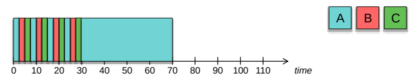


Average response time =
= (0 + 50 + 60)÷3 = 36.67

Average response time =
= (0 + 10 + 20)÷3 = 10

Round Robin

- Let's add preemption
  - Let a job run for some time (time slice = quantum)
  - Then context switch and give someone else a turn



- If quantum = 2.5:
  - average response time = (0+2.5+5) ÷ 3 = 2.5 ⇒ Great!
  - average turnaround time = (70+27.5+30) ÷ 3 = 42.5
    - worse than SJF (33.3) but better than worst-case FCFS (60)
    - In general, Round Robin is not good for turnaround time

Time slice (quantum) length

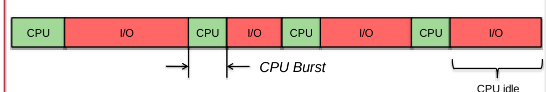
- Short quantum: increases overhead % of context switching
- Long quantum: reduces interactivity
  - Tasks are allowed to run longer before a context switch is forced
  - Amortizes overhead of context switch
- No perfect answer
  - Servers: higher emphasis on efficiency
    - Use a longer quantum to reduce overhead of context switches
    - But still need interactivity to schedule I/O and provide decent response
  - Interactive systems: higher emphasis on fast user response
    - Use a shorter quantum to have more context switches
- But…
  - Interactive and I/O-bound tasks rarely will use up their time slice

What about I/O?

We ignored I/O so far

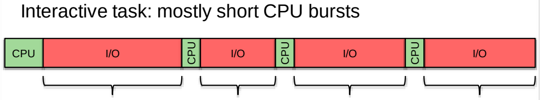
Most tasks fall into one of two categories:

1. Large # of short CPU bursts between I/O requests
2. Small # of long CPU bursts between I/O requests



Task Behavior

Interactive task: mostly short CPU bursts

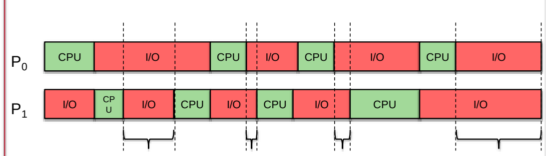


Compute task: mostly long CPU bursts

Task Scheduling With I/O

Goal:

- Maximize use of CPU & improve throughput
- Let another task run when the current one is waiting on I/O

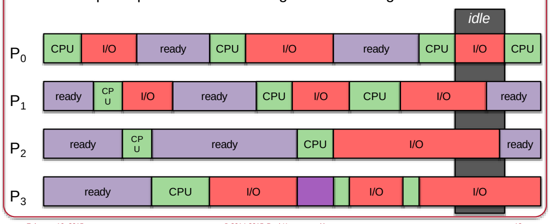


Think of each CPU burst as an individual job that needs to be scheduled

© 2014-2015 Paul Krzyzanowski

Process Scheduling With a Mix of Processes

Improve CPU utilization (increase chance of CPU being busy)
- Some processes will use long stretches of CPU time
  - Preempt them periodically and let another process run
- More processes than CPUs: keep them in the *ready* list
- Perhaps all processes are waiting on I/O: nothing to run!



When does the scheduler make decisions?

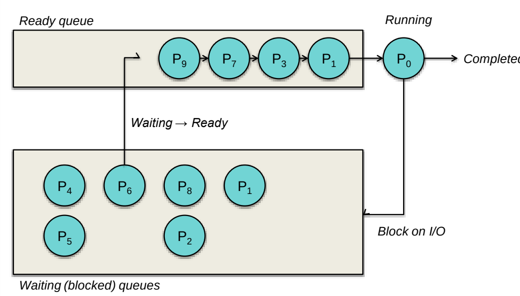
Four events may cause the scheduler to get called:
1. Current process goes from *running* to *blocked* state
2. Current process terminates
3. Interrupt gives the scheduler a chance to move a process from *running* to *ready: scheduler decides it's time for someone else to run*
4. Current process goes from *blocked* to *ready*
   I/O is complete (including blocking events, such as semaphores)
   This does not necessarily mean the currently running process will change

- Preemptive scheduler
- Cooperative (non-preemptive) scheduler
  - CPU cannot be taken away unless a system call takes place or process exits
- Run-to-completion scheduler (old batch systems)

Scheduling algorithm goals

Be fair	(to processes? To users?)
Be efficient	Keep CPU busy … and don't spend a lot of time deciding!
Maximize throughput	Get as many processes to complete as quickly as possible
Minimize response time	Minimize time users must wait
Be predictable	Tasks should take about the same time to run & responsiveness should be similar when run multiple times
Minimize overhead	
Maximize resource use	Try to keep devices busy!
Avoid starvation	
Enforce priorities	
Degrade gracefully	

First-Come, First-Served (FCFS)

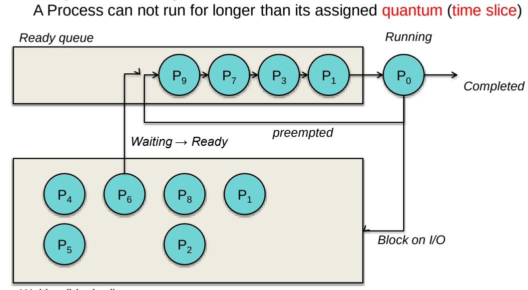


First-Come, First-Served (FCFS)

- Non-preemptive
- A process with a long CPU burst will hold up other processes
  - I/O bound tasks may have completed I/O and are ready to run: poor device utilization
  - Poor average response time

Round-Robin Scheduling

Preemptive Scheduling:
A Process can not run for longer than its assigned quantum (time slice)



Round-Robin Scheduling

- Behavior depends on the quantum
  - Long quantum makes this similar to FCFS
  - Short quantum increases interactivity but increases the overhead % of context switching
- Advantages
  - Every process gets an equal share of the CPU
  - Easy to implement
  - Easy to compute average response time: *f*(# processes on list)
- Disadvantage
  - Giving every process an equal share isn't necessarily good
  - Highly interactive processes will get scheduled the same as CPU-bound processes

Shortest Remaining Time First Scheduling

- Sort tasks by anticipated CPU burst time
- Schedule shortest ones first
- Optimize average response time

Burst time	2	2	10	3	8	Total time = 25
Process queue	*E*	*D*	*C*	*B*	*A*	
Total run time	25	23	21	11	8	Mean time = 17.6

last ← *first*

Burst time	10	8	3	2	2	Total time = 25
Process queue	*C*	*A*	*B*	*D*	*E*	
Total run time	25	15	7	4	2	Mean time = 10.6

Mean completion time for a process falls by almost 40%!

Shortest Remaining Time First Scheduling

- Biggest problem: *we're optimizing with data we don't have!*
- All we can do is estimate
- Exponential average – estimate of next CPU burst:

$$e_{n+1} = \alpha t_n + (1 - \alpha)e_n$$

average of all previous CPU bursts
time of current CPU burst

α is a weight factor to balance the weight of the last burst period vs. historic periods ($0 \le \alpha \le 1$)

If $\alpha = 0$: $e_{n+1} = e_n$ (recent history has no effect)
If $\alpha = 1$: $e_{n+1} = \alpha t_n$ (use only the last burst time)

Shortest Remaining Time First Scheduling

- Advantage
  - Short-burst tasks run fast
- Disadvantages
  - Long-burst (CPU intensive) tasks get a long mean waiting time
    - Starvation risk!
  - Need to rely on ability to estimate CPU burst length

Priority Scheduling

Round Robin assumes all processes are equally important

- Not true
  - Interactive tasks need high priority for good response
  - We might want non-interactive tasks to get the CPU less frequently: *this goal led us to SRTF*
  - Some tasks might be time critical
  - Users may have different status (e.g., administrator)
- Priority scheduling algorithm:
  - Each process has a priority number assigned to it
  - Pick the process with the highest priority
  - Processes with the same priority are scheduled round-robin

Priority Scheduling – Assigning Priorities

- Priority assignments:
  - Internal: time limits, memory requirements, I/O:CPU ratio, …
  - External: assigned by administrators
- Static & dynamic priorities
  - Static priority: priority never changes
  - Dynamic priority: scheduler changes the priority during execution
    - Increase priority if it's I/O bound for better interactive performance or to increase device utilization
    - Decrease a priority to let lower-priority processes run
    - Example: use priorities to drive SJF/SRTF scheduling

© 2014-2015 Paul Krzyzanowski

Priority Scheduling – Problems

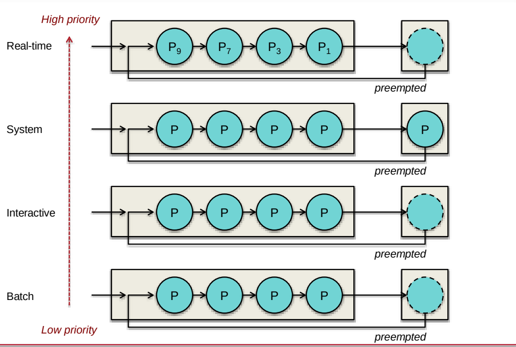
- Priority Inversion
  - A low-priority thread may not get scheduled, thereby preventing a high-priority thread that is holding a resource from making progress
- Starvation
  - A low priority thread may *never* get scheduled if there is always a high-priority thread ready to run

Multilevel Queues

Does each task need to have a unique priority level?

- Priority classes: a ready queues for each priority level
  - Each priority class gets its own queue
  - Processes are permanently assigned to a specific queue
  - Examples: System processes, interactive processes, slow interactive processes, background non-interactive processes
- Implementation
  - Priority scheduler with queues per priority level
  - Each queue may have a different scheduling algorithm (usually round-robin)
  - Quantum may be increased at each lower priority level
    - Lower-priority processes tend to be compute bound

Multilevel Queues

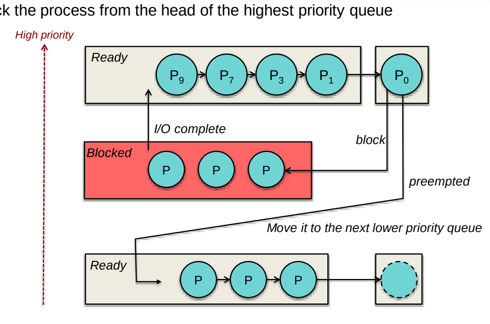


Multilevel Feedback Queues

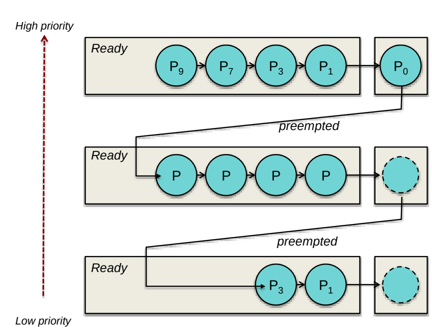
- Goals
  - Allow processes to move between priority queues based on feedback
    - Have the scheduler learn the behavior of each task and adjust priorities
- Separate processes based on CPU burst behavior
  - I/O-bound processes will end up on higher-priority queues
- Rules
  1. A new process gets the highest priority
  2. If a process does not finish its quantum (blocks on I/O) then it will stay at the same priority level (round robin) otherwise it moves to the next lower priority level

Multilevel Feedback Queues

Pick the process from the head of the highest priority queue



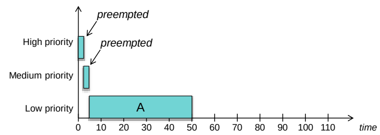
Multilevel Feedback Queues



© 2014-2015 Paul Krzyzanowski

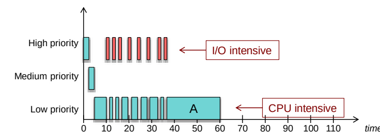
Example

One long-running process



Example

- Suppose a highly interactive process, *B* (■), starts at *T=10*
- It never uses up its quantum
  - *B* gets priority but spends a lot of its time in the blocked state
  - A (■) gets to run only when *B* is blocked



Starvation & aging

- Two problems
  - Starvation:
    If there are a lot of interactive processes, the CPU-bound processes will never get to run
  - Interactive process ending up at a low priority:
    If a process was CPU intensive (e.g., initializing a game) but then became interactive, it is forever doomed to a low priority
- Solve these process aging
  - Increase the priority of a so it will be scheduled to run
    - Simplest approach: periodically, set *all* processes to the highest priority
  - If it remains CPU-intensive, its priority will quickly fall again

Gaming the system

- What if you make an I/O operation just before the end of each time slice?
  - You get to stay at a high priority!
- Solution
  - Don't worry whether a process uses up its time slice
  - Instead, keep track of CPU time used over a larger time interval
    - If a process uses up its allotment, then lower its priority
    - Lower levels can have longer allotments

Varying time slices

Two thoughts

1. Lower priority processes get longer time slices
   - Amortize the cost of context switching
   - CPU-intensive tasks don't get to run often. When they do, let them run for a longer time.
   - Interactive tasks rarely use up their quantum anyway. Keep it short
2. Higher priority processes get longer time slices
   - Measure CPU usage per process over a longer time interval
   - Low CPU users are interactive and get high priority
   - If an interactive process needs to do some computation for a while, let it do so at high priority

Multilevel Feedback Queues

- Advantage
  - Good for separating processes based on CPU burst needs
  - Give priority to I/O bound processes
  - No need to estimate interactivity! (Estimates were often flawed)
- Disadvantages
  - Priorities get controlled by the system.
    A process is considered important because it uses a lot of I/O
  - Processes whose behavior changes may be poorly scheduled
  - System can be gamed by scheduling bogus I/O
    … but we have workarounds

Lottery Scheduling (Fair Share)

- Each process gets some % of "tickets"
  - E.g., 100 tickets total
    Process A: 50 tickets (0…49)
    Process B: 25 tickets (50…74)
    Process C: 25 tickets (75…99)
- The scheduler picks a random number = winning ticket
  - Process with the winning ticket gets to run
- Benefit: Proportional share
- Difficulty: determining ticket distribution
  - Not used for general-purpose scheduling
  - More useful for applications such as scheduling multiple virtual machines

Summary

- Schedulers
  - **Shortest remaining time first**
    - Not used – Too much computing & danger of starvation
  - **Multilevel Queues**
    - Fixed priority – multiple processes per priority level
  - **Multilevel Feedback Queues**
    - Dynamic adjustment of process priority based on CPU usage
  - **Lottery**
    - Fair share: % allocation of CPU to each process
- Parameters
  - Number of priority levels? (usually 32…140)
  - Quantum size – fixed or variable?
  - Keep track of CPU usage over a larger interval?
  - Process aging – how often do you boost the priority?

Multiprocessor Scheduling

Hierarchy of processors

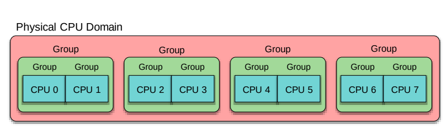
- Virtual CPUs in a hyperthreaded core
  - Equal access to memory, all caches, and processor
  - A CPU has to wait on a memory stall
    (e.g., to get data on a cache miss)
  - When the issuing logic can no longer schedule instructions from one thread and there are idle functional units in the CPU core, it will try to schedule a suitable instruction from the other thread
- Cores in a multicore processor
  - Equal access to memory and an L3 cache
- Processors in a symmetric multiprocessor (SMP) system
  - Equal access to memory but no shared cache
- NUMA (Non-Uniform Memory architecture)
  - Access to all memory but not all memory is local

Symmetric multiprocessor scheduling

- General goal: maintain processor affinity
  - Try to reschedule a process onto the same CPU
  - Cached memory may be present on the CPU's cache
  - *Keep a run queue per CPU*
- Types of affinity
  - Hard affinity: force a process to stay on the same CPU
  - Soft affinity: best effort, but the process may be rescheduled on a different CPU
    - Load balancing: ensure that CPUs are busy
    - It's better to run a task on another CPU than wait
    - *Pull migration*
      If the run queue for a CPU is empty, get a task from another CPU's run queue
    - Push migration
      Check load periodically: if not balanced, move tasks

Linux Multiprocessor Scheduling

- Scheduling domain
  - Set of CPUs that share properties & scheduling policies
- Domain contains one or more CPU groups
  - Each CPU group is treated as one unit by that domain
  - Balances load across groups – doesn't care what's in the group



© 2014-2015 Paul Krzyzanowski

Scheduling Domain Policies

- Each scheduling domain has a balancing policy
  - Valid for that level of the hierarchy
  - How often should attempts be made to balance load across groups in the domain?
  - How far can the loads in the domain get unbalanced before balancing across groups is needed?
  - How long can a group in the domain sit idle?
- Active load balancing is performed periodically
  - Moves up the scheduling domain hierarchy and checks all groups along the way
  - If any group is out of balance based on policy rules, it rebalances

Scheduler Examples

Solaris Scheduler

- Multilevel queue scheduler: 170 priorities (0-169)
  - High priority → short quantum
- Six scheduling classes
  - Each class has priorities and scheduling algorithms

1. Time sharing (0-59)
   Default class. Dynamic priorities via a multilevel feedback queue *DEFAULT*
2. Interactive (0-59)
   Like TS but higher priority for in-focus windows in GUI
3. Real-time (100-159)
   Fixed priority, fixed time quantum; high priority values
4. System (60-99)
   Used to schedule kernel threads: run until they block or complete
5. Fair share (0-59)
   Processes scheduled on % of CPU
6. Fixed priority (0-59)
   Fixed priority

Highest priority (160-169): interrupt-handling threads

Solaris Scheduler

- Default class: time sharing
  - Multilevel feedback queue
  - Small time slice for high priority queue
  - Long time slice for low priority queue
- Interactive class: similar but gives windowing apps higher priority

MLFQ

- Highest priority: threads in the real-time class
- System class: runs kernel threads (scheduler & paging)
  - Not preempted

Multilevel Queues

- Fair share: set of processes get a "CPU share"

Similar to Lottery

- Fixed priority: like time-sharing but never adjusted

Multilevel Queues

Windows Scheduler

- Two classes:
  - Variable class: priorities 0-15
  - Real-time class: priorities 16-31
- Each priority level has a queue
  - Pick the highest priority thread that is ready to run
- Relative priority
  - Threads have relative levels within their class
  - When a quantum expires, the thread's priority is lowered but never below the base
  - When a thread wakes from wait, the priority is increased
    - Higher increase if waiting for keyboard input
  - Priority is increased for foreground window processes

Windows Priorities

	Real-time	High	Above Normal	Normal	Below Normal	Idle
Time-Critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above Normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

© 2014-2015 Paul Krzyzanowski

Linux Schedulers – History

- Linux 1.2: Round Robin scheduler (fast & simple)
- Linux 2.2: Scheduling classes (multilevel queue)
  - Classes: Real-time, non-real-time, non-preemptible
  - Basic support for symmetric multiprocessing

Linux 2.4: O(N) Scheduler

- Multilevel queue with two scheduling algorithms:
- (1) Real-time with absolute priorities (but kernel is not preemptible)
  - FIFO & Round-robin options
- (2) Time-sharing: Credit-based algorithm
  - Each task has some # of credits associated with it
  - On each timer interrupt:
    - Each timer interrupt: running task loses 1 credit
    - If credits for a task == 0, the task is suspended
    - If all tasks have 0 credits:
      - Re-credit: Every task gets credits = credits/2 + priority
    - Choose next task to run: pick the one with the most credits
- Not good for systems with many tasks
  - Re-crediting requires going through every task: O(N)
- Not good for multiprocessor systems
  - One queue (in a mutex): contention & no processor affinity

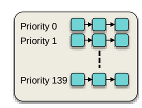
Linux 2.6: O(1) scheduler goals

Addressed three problems

- Scalability: O(1) instead of O(n) to not suffer under load
- Support processor affinity
- Support preemption in the kernel
  - High-priority (real-time) tasks can interrupt a task running in kernel mode

Linux 2.6: O(1) scheduler

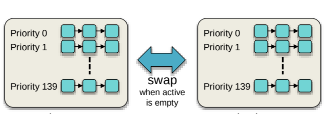
- O(1) instead of O(N): no increased overhead with more tasks
- One run queue per CPU
- Always schedule the highest priority task
  - Multiple tasks at same priority scheduled round-robin
- Multilevel queue
  - 140 queues (priority levels)
    - 0-99 = real-time
    - 100-139 = timesharing – dynamic priorities
  - Each priority level has its own time slice
    - Higher priority = LONGER time slice



runqueue

Linux 2.6: O(1) scheduler

- Two sets of queues: active & expired
  - Epoch = time when all runnable tasks begin with a fresh time slice
  - When a task uses up its time slice, it is moved to the expired list
  - When there are no runnable tasks in the active list
    - Active & expired lists are swapped: end of epoch & start of a new one
    - Simulates aging



active runqueue expired runqueue

Linux 2.6: O(1) scheduler

- Real-time tasks: static priorities
  - Choice of round-robin or FIFO
- Non real-time tasks: dynamic priorities → reward interactive tasks
  - I/O-bound processes get priority increased by up to 5 levels
  - CPU-bound processes get priority decreased up to 5 levels
  - "Interactivity credits": +credit for sleeping, -credit for running
- SMP load balancing
  - Every 200ms, check if CPU loads are unbalanced
  - If so, move tasks from a loaded CPU to a less-loaded one
  - If a CPU's runqueue is empty, move from another CPU's runqueue
- Downside of O(1) scheduler
  - A lot of code with complex heuristics

© 2014-2015 Paul Krzyzanowski

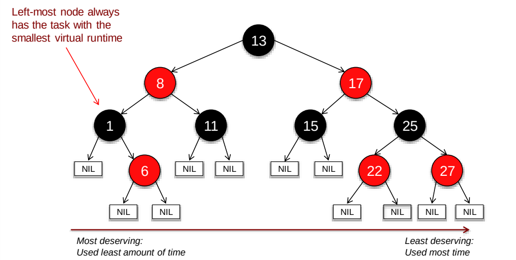
Linux Completely Fair Scheduler

- Latest scheduler (introduced in 2.6.23)
- Goal: give a "fair" amount of CPU time to tasks
- Keep track of time given to a task: "virtual runtime"
- Basic heuristic: tasks get a fair % of the processor
  - But interactive processors are unlikely to use their share
  - When an interactive task wakes up, the scheduler sees that it used less than its fair share. To try to be fair, it preempts a compute-bound task
- Priorities – affect the rate of "virtual runtime"
  - High priority task's *vruntime* grows slower than the *vruntime* of a low priority task

Linux Completely Fair Scheduler

No run queues: virtual runtime sorted red-black tree used instead

- Self-balancing binary tree: search, insert, & delete in O(log n)



From: http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg

CFS: picking a process

- Scheduling decision:
  - Pick the leftmost task (smallest virtual runtime)
- When a task is moved from running → ready:
  - Add execution time to the per-task run time count
  - Insert the task back in the sorted tree
- Heuristic: *decay factors*
  - Determine how long a task can execute
  - Higher priority tasks have lower factors of decay.
  - Avoids having run queues per priority level

Group Scheduling

- Default operation: be fair to each task
- Group scheduling: Assign one virtual runtime to a group of processes
  - Per user scheduling
  - *cgroup* pseudo file system interface for configuring groups
  - E.g., a user with 5 processes can get the same % of CPU as a user with 50 processes
- Default task group: init_task_group
- Improve interactive performance
  - A task calls `__proc_set_tty` to move to a tty task group
- `/proc/sys/kernel/sched_granularity_ns`
  - Tunable parameter to tune the scheduler between desktop (highly interactive) and server loads

Linux scheduling classes

- Modular scheduler core: Scheduling classes
  - Scheduling class defines common set of functions that define the behavior of that scheduler
    - Add a task, remove a task, choose the next task
  - Each task belongs to a scheduling class
  - `sched_fair.c`
    - implements the CFS scheduler
  - `sched_rt.c`
    - implements a priority-based round-robin real-time scheduler
- Scheduling domains
  - Group one or more processors in a hierarchy
  - One or more processors can share scheduling policies

The End