CS310 Operating Systems

Lecture 27 : Thread Scheduling, Linux Scheduling

Ravi Mittal
IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Book: Linux System Programming, Robert Love
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Modern Operating System, Andrew Tanenbaum

Reading

- Book: Linux Kernel Development, Robert Love
  - Chapter 4 Process Scheduling
- Book: Advanced Unix Programming, by Mark J Rochkind

Previous Classes

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also First In First Out (FIFO) or Run until done
  - In early systems, FCFS meant one program scheduled until done (including I/O)
  - Now, means keep CPU until thread blocks

- Simple Algorithm, Easy to implement (+)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
  - *Convoy effect:* short process stuck behind long process (-)

Shortest Job First (SJF)

- Non-preemptive
- Run whatever job has least amount of computation to do
- Provably optimal
- Need to know run times in advance

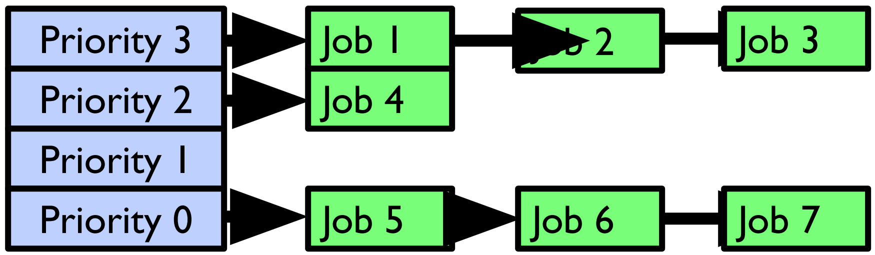
Shortest Remaining Time First (SRTF)

- Preemptive version of SJF
- If job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
- Sometimes called Shortest Remaining Time to Completion First (SRTCF)
- Both SJF and SRTF:
  - These can be applied to whole program or current CPU burst
    - Idea is to get short jobs out of the system
    - Big effect on short jobs, only small effect on long ones
    - Result is better average response time

Round Robin (RR) Scheduling

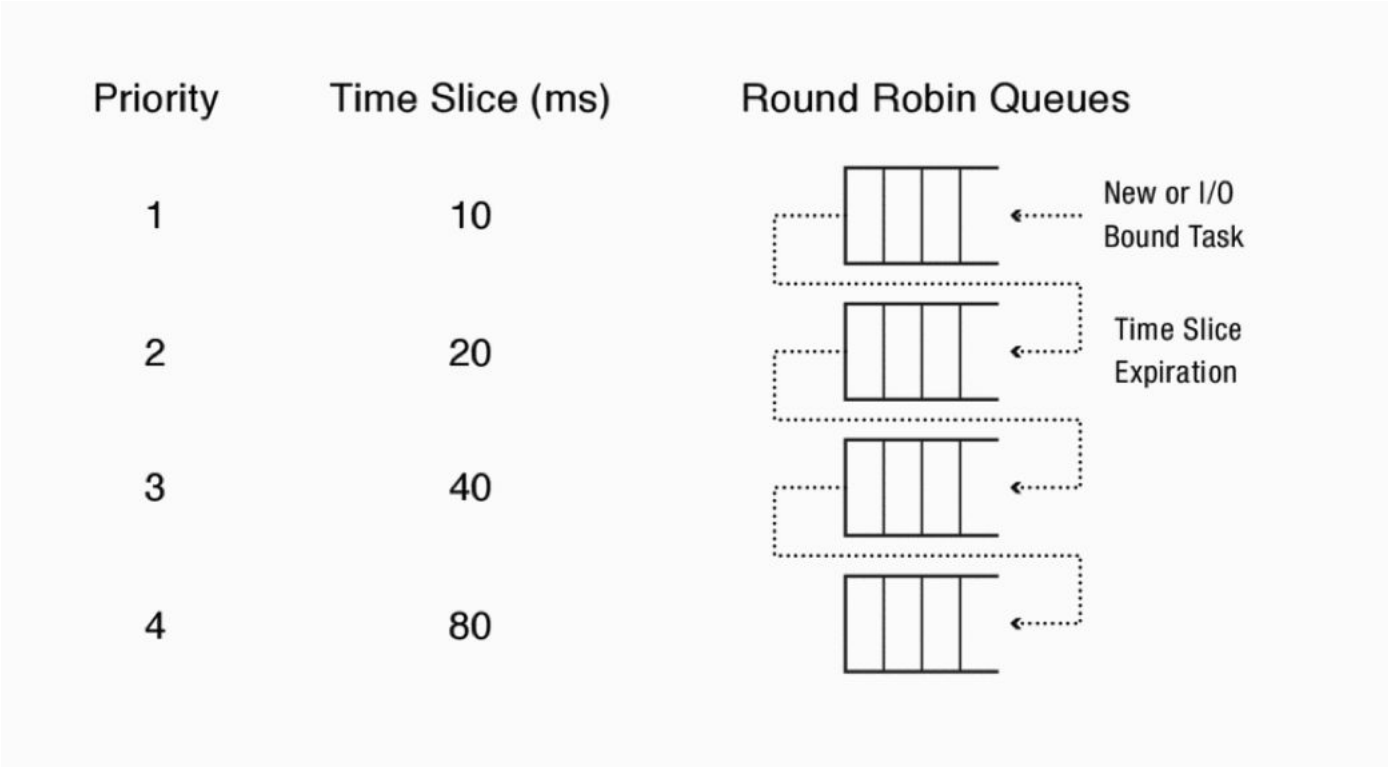
- Uses Preemption!
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
- After quantum expires, the process is preempted and added to the end of the ready queue
- n processes in ready queue and time quantum is q
    - Each process gets 1/n of the CPU time
    - In chunks of at most q time units
    - No process waits more than (n-1)q time units

Multilevel Queue Scheduling – Strict Priority



- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
  - A priority is assigned statically to each process, and a process remains in the same queue for the duration of the run time
- Problems
  - Starvation
    - Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - Happens when low priority task holds a lock needed by high-priority task

Example of Multilevel Feedback Queue



Lottery Scheduling

- Proportional share scheduling
  - 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? Users can define policy
  - 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

Use Randomness

Requires a good random number generator

Scheduling Summary

- FCFS is simple and it minimizes overhead
- If tasks are variable in size, FCFS can have very poor average response time
- FCFS suffers from convoy effect
- SJF is optimal in terms of average response time
- If tasks are variable in size, Round Robin approximates SJF
- If tasks are equal in size, Round Robin will have very poor average response time
- Round Robin avoid starvation
- MFQ scheduler can achieve a balance between responsiveness, low overhad and fairness

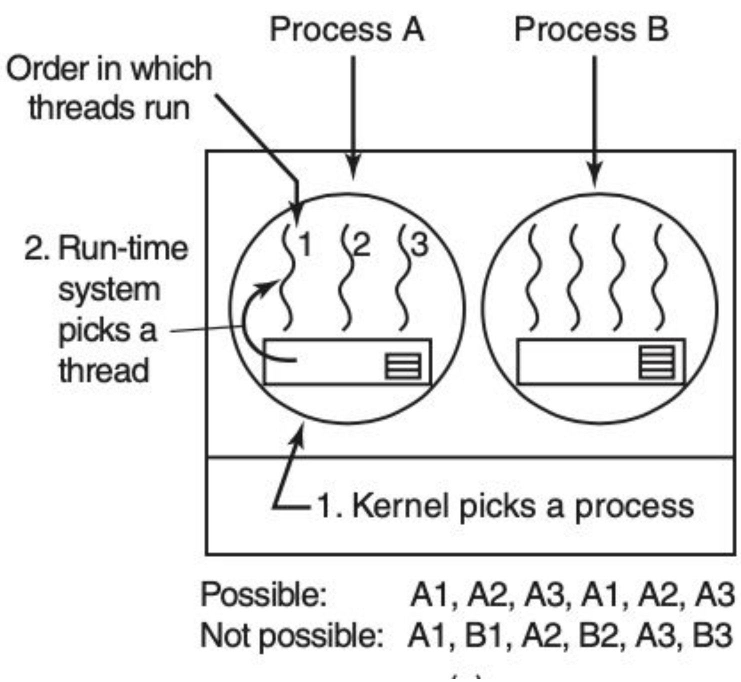
Thread - Scheduling

So, Does the OS Schedule Processes or Threads?

- Switching threads vs. switching processes incurs different costs:
  - Switch threads: Save/restore registers
  - Switch processes: Change active address space too!
    - Expensive
    - Disrupts caching
- A system have have many processes and many threads running
- Recall User level threads vs Kernel level threads
  - Scheduling for both differs

User level thread Scheduling

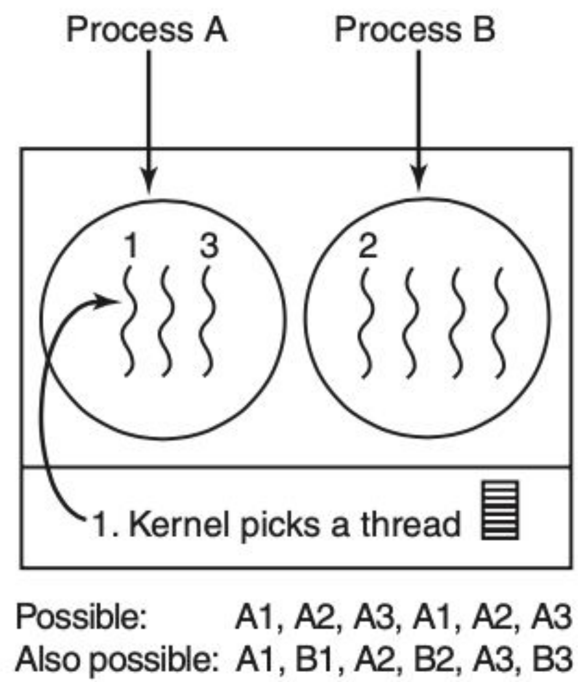
- Kernel is not aware of existence of threads
- Kernel picks up a process, say A, and giving it a control for the quantum
  - Runtime system (Thread scheduler) inside A decides which thread to run, say A1
  - As there are no clock interrupts used within Runtime System, A1 will continue to run unless it has do to I/O or process scheduler context switches it
  - When A is resumed back, A1 again starts running
  - If A1 finishes within the quantum, runtime scheduler will schedule A2 ..and so on



Runtime scheduler can run any scheduling algorithm: example: RR

Kernel level thread Scheduling

- Kernel picks up a particular thread to run
- It doesn't take into account which process the thread belongs to
  - However it can take into account if it wants to
- Doing thread scheduling at Kernel level requires full context switch
  - Changing memory map
  - Cache data management
- Slower than that in User level switching
- If a thread blocks on I/O, entire process is not blocked
  - Kernel can switch to another thread in the same process



Runtime scheduler can run any scheduling algorithm: example: RR

Linux – Completely Fair Scheduler (CFS)

Linux Scheduler Overview

- Scheduling policy attempts to satisfy conflicting goals
  - Fast process response time
  - Maximal system utilization (throughput)
- Aims to provide good interactive responses
  - favors I/O bound processes over processor bound processes
  - Done in creative manner – doesn’t neglect processor-bound processes
- Reasonably good for Real-time applications
  - So, there must be something to handle real-time processes
- Priority based Scheduling
  - So, there must be some way to define priority of a process
- Preemptive Scheduling
- Thread based Scheduling – Linux treats a process (without threads inside) as a single thread – Kernel level threads
- Two states that are of interest in scheduling: blocked and runnable

Priority Based Scheduling

- CFS was introduced in Linux 2.6.23 in 2207
  - For Normal processes .. Named as SCHED_NORMAL
  - Can be configured for batch workloads: SCHED_BATCH
- Rank processes based on their worth and need for processor time
  - User and the system can set a process’s priority
- Real-Time processes have higher priorities than those of all normal processes
- The Linux kernel implements two separate priority ranges
  - Nice value
  - Real time priority

Real Time Priority

- Range from 0 to 99, inclusive
- Opposite from nice values, higher real-time priority values correspond to a greater priority
- All real-time processes are at a higher priority than normal processes
- The real-time priority and nice value are in disjoint value spaces

ps -eo state,uid,pid,ppid,rtprio,time,comm

```
S   UID   PID  PPID RTPRIO     TIME COMMAND
S     0     1     0      - 00:00:00 init
S     0     2     0      - 00:00:00 kthreadd
S     0     3     2      - 00:00:00 ksoftirqd/0
S     0     5     2      - 00:00:00 kworker/0:0H
S     0     7     2      - 00:00:15 rcu_sched
S     0     8     2      - 00:00:00 rcu_bh
S     0     9     2     99 00:00:00 migration/0
S     0    10     2     99 00:00:00 watchdog/0
S     0    11     2     99 00:00:00 watchdog/1
S     0    12     2     99 00:00:00 migration/1
S     0    13     2      - 00:00:00 ksoftirqd/1
S     0    15     2      - 00:00:00 kworker/1:0H
S     0    16     2     99 00:00:00 watchdog/2
S     0    17     2     99 00:00:00 migration/2
S     0    18     2      - 00:00:00 ksoftirqd/2
S     0    20     2      - 00:00:00 kworker/2:0H
S     0    21     2     99 00:00:00 watchdog/3
S     0    22     2     99 00:00:00 migration/3
S     0   359     2     50 00:00:00 irq/280-mei_me
S     0   364     2      - 00:00:00 kworker/u9:0
S     0   365     2      - 00:00:00 hci0
S     0   366     2      - 00:00:00 hci0
S     0   367     2      - 00:00:00 kworker/u9:1
S     0   371     2      - 00:00:00 kworker/3:1H
S     0   375     2      - 00:00:00 kworker/1:1H
S     0   412     2      - 00:00:00 cfg80211
S     0   414     2     50 00:00:07 irq/281-iwlwifi
S     0   417     2     50 00:00:15 irq/51-DELL0767
S     0   464     2      - 00:00:00 kvm-irqfd-clean
S     0   498     1      - 00:00:00 upstart-socket-
S     0   552     2      - 00:00:00 kmemstick
S     0   553     2      - 00:00:02 rtsx_usb_ms_1
S     0   569     2      - 00:00:00 kfd_process_wq
S     0   571     2      - 00:00:00 ttm_swap
S     0   631     2      1 00:00:00 gfx
S     0   632     2      1 00:00:00 comp 1.0.0
S     0   633     2      1 00:00:00 comp 1.0.1
S     0   634     2      1 00:00:00 comp 1.0.2
S     0   635     2      1 00:00:00 comp 1.0.3
S     0   636     2      1 00:00:00 comp 1.0.4
S     0   637     2      1 00:00:00 comp 1.0.5
S     0   638     2      1 00:00:00 comp 1.0.6
S     0   639     2      1 00:00:00 comp 1.0.7
S     0   640     2      1 00:00:00 sdma0
S     0   641     2      1 00:00:00 sdma1
S   102   808     1      - 00:00:02 dbus-daemon
```

Nice Value

- A number from –20 to +19 with a default of 0
- Larger nice values correspond to a lower priority
  - you are being “nice” to the other processes on the system
- Processes with a lower nice value (higher priority) receive a larger proportion of the system’s processor compared to processes with a higher nice value (lower priority)
- Nice values are the standard priority range used in all Unix systems
  - Different Unix systems apply them in different way based on the individual scheduling algorithm
  - Mac OS X, the nice value is a control over the *absolute* timeslice allotted to a process;
  - In Linux, it is a control over the *proportion* of timeslice

ps -el

```
[(base) Ravis-MacBook-Pro-2:~ ravi itta $ ps -el
  UID   PID  PPID        F CPU PR  NI       SZ    RSS WCHAN     S             ADDR TTY           TIME CMD
    0     1     0     4004   0  3   0  4316552  13356 -      Ss               0 ??        40:10.41 /sbin/launchd
    0   149     1     4004   0      0  4306048    612 -      Ss               0 ??         1:16.89 /usr/sbin/syslogd
    0   150     1     4004   0  3   0  4336300   3384 -      Ss               0 ??         0:45.73 /usr/libexec/UserEventAgent (System)
    0   153     1     4004   0  2   0  4280516    392 -      Ss               0 ??         0:28.22 /System/Library/PrivateFrameworks/Uninstall.framework/Resourc
    0   154     1     4004   0  4   0  4853492   1860 -      Ss               0 ??         1:13.19 /usr/libexec/kextd
    0   155     1  1004004   0  5   0  5483084   2436 -      Ss               0 ??        18:53.18 /System/Library/Frameworks/CoreServices.framework/Versions/A/
    0   157     1     4004   0  2   0  4283024     64 -      Ss               0 ??         0:00.25 /Library/Frameworks/OVPNHelper.framework/Versions/Current/usr
    0   158     1     4004   0      0  4337344    900 -      Ss               0 ??         0:07.16 /System/Library/PrivateFrameworks/MediaRemote.framework/Suppo
    0   161     1     4004   0  2   0  4290712     68 -      Ss               0 ??         0:00.25 /Library/Frameworks/OpenVPNConnect.framework/Versions/Current
    0   162     1     4004   0      0  4356092  19856 -      Ss               0 ??        44:54.17 /usr/sbin/systemstats --daemon
    0   163     1     400c   0  3   0  4342128   4112 -      Ss               0 ??         2:57.13 /usr/libexec/configd
    0   165     1     4004   0  3   0  4335032   3620 -      Ss               0 ??         3:44.47 /System/Library/CoreServices/powerd.bundle/powerd
    0   170     1     4004   0  3   0  4402740   7788 -      Ss               0 ??        12:04.63 /usr/libexec/logd
    0   175     1     4004   0  9   0  4305400   1176 -      Ss               0 ??         0:30.29 /usr/libexec/watchdogd
    0   179     1  1004004   0  5   0  4509128  12868 -      Ss               0 ??        26:06.75 /System/Library/Frameworks/CoreServices.framework/Frameworks/
    0   180     1  1004004   0  2   0  4365540   5704 -      Ss               0 ??         2:34.73 /Library/Bitdefender/Central/Agent/bdagentd
    0   182     1     4004   0  3   0  4333064   2132 -      Ss               0 ??         1:10.03 /usr/libexec/diskarbitrationd
    0   192     1     4004   0  3   0  4347980   6052 -      Ss               0 ??        10:33.66 /usr/libexec/opendirectoryd
    0   193     1     4004   0  3   0  4342672   5040 -      Ss               0 ??         1:03.71 /System/Library/PrivateFrameworks/ApplePushService.framework/
    0   194     1     4004   0      0  4341228   6844 -      Ss               0 ??        14:19.77 /System/Library/CoreServices/launchservicesd
  266   195     1     4004   0  3   0  4333492   1972 -      Ss               0 ??         0:12.62 /usr/libexec/timed
  213   196     1     4004   0  3   0  4334204   1180 -      Ss               0 ??         0:06.16 /System/Library/PrivateFrameworks/MobileDevice.framework/Vers
    0   197     1     4004   0  3   0  4338632   3392 -      Ss               0 ??         1:45.89 /usr/sbin/securityd -i
    0   198     1     4004   0  2   0  4296624     36 -      Ss               0 ??         0:00.12 auditd -l
    0   203     1     4004   0  2   0  4305476     36 -      Ss               0 ??         0:00.08 autofsd
  244   204     1     4104   0  2   0  4305928    324 -      Ss               0 ??         0:01.56 /usr/libexec/displaypolicyd -k 1
    0   206     1     4004   0      0  4343844   5108 -      Ss               0 ??         2:34.96 /usr/libexec/dasd
    0   210     1     4004   0  3   0  4304940    584 -      Ss               0 ??         0:00.68 /System/Library/CoreServices/logind
    0   211     1  1004004   0      0  4340860   1128 -      Ss               0 ??         0:03.10 /System/Library/PrivateFrameworks/GenerationalStorage.framewo
    0   212     1     4004   0  3   0  4305344     68 -      Ss               0 ??         0:00.11 /usr/sbin/KernelEventAgent
    0   214     1 40004004   0  3   0  4340320   5532 -      Ss               0 ??        14:31.29 /usr/sbin/bluetoothd
  261   215     1     4004   0  6   0  4337444   3948 -      Ss               0 ??        68:41.54 /usr/libexec/hidd
    0   217     1     4004   0  3   0  4335460   1672 -      Ss               0 ??         0:45.55 /usr/libexec/corebrightnessd --launchd
    0   218     1     4004   0  5   0  4336216   1652 -      Ss               0 ??         0:11.66 /usr/libexec/AirPlayXPCHelper
    0   219     1     4004   0  3   0  4305816   1424 -      Ss               0 ??         3:20.00 /usr/sbin/notifyd
  241   221     1     4004   0  3   0  4306540    996 -      Ss               0 ??         1:51.38 /usr/sbin/distnoted daemon
    0   224     1     4004   0  3   0  4532928   1956 -      Ss               0 ??         0:40.69 /System/Library/CoreServices/coreservicesd
    0   225     1     4004   0      0  4305896   2604 -      Ss               0 ??         4:37.04 /usr/sbin/cfprefsd daemon
    0   226     1     4004   0  3   0  4379508  11932 -      Ss               0 ??        21:07.16 /usr/libexec/syspolicyd
  501   228     1 80004104   0  6   0  4396456  18332 -      Ss               0 ??         8:03.51 /System/Library/CoreServices/loginwindow.app/Contents/MacOS/l
    0   230     1     4004   0      0  4337684   3504 -      Ss               0 ??         0:16.21 /System/Library/Frameworks/Security.framework/Versions/A/XPCS
  263   275     1     4004   0  3   0  4340620   5684 -      Ss               0 ??         1:00.66 /System/Library/PrivateFrameworks/CoreAnalytics.framework/Sup
    0   278     1     4004   0      0  4344108   5588 -      Ss               0 ??        29:34.90 /usr/libexec/trustd
    0   279     1     4004   0  3   0  4374912  11704 -      Ss               0 ??        20:03.68 /usr/libexec/airportd
  202   280     1     4004   0  6   0  4344296   2372 -      Ss               0 ??       189:45.56 /usr/sbin/coreaudiod
    0   281     1     4004   0      0  4342944   6316 -      Ss               0 ??         0:39.09 /usr/libexec/lsd runAsRoot
```

CFS

- No fixed timeslices
    - This is a major change from other OSs – where time given to each processor remains fixed
- No explicit priorities
    - Does this mean priority of a process/thread changes over time?
- Amount of time for a given task is computed dynamically
    - As scheduling context changes

CFS - Algorithm

- CFS is designed to approximate perfect multitasking
  - Imagine as processor P, which is idealized in that it can executed multiple tasks simultaneously
  - If there are two tasks, both can run a the same time ; each receiving 50% of processing power
- CFS scheduler has a target latency
  - Idealized as very small duration – such that every runnable task gets at least one turn
  - Call it target latency : say 20 ms
  - If there are N tasks, each get 1/N slice of target latency
    - Example: If N = 4 each task get 5 ms
- Note that 1/N Slice : is not fixed as it depends on N
  - N keeps changing

CFS - Algorithm

- Each of these tasks (N) will contend for processor
- Now, traditional Nice value is used to weight the 1/N slice
- Low priority nice value : only a fraction of the 1/N slice is given to the task
- High priority nice value: proportionately greater fraction of the 1/N slice is given to a task
  - Nice values only modify 1/N slice
- To take care of overheads of switching, a minimum amount of time is fixed – where in a process must run before being preempted - called minimum granularity
  - Assume 4 ms
- If more number of processes are contending (say 20 processes), 1/N slice is much smaller than minimum granularity
  - No fairness

How does preemption occur?

- A task is preempted when it's period : weighted 1/N slice completes
- Which task is chosen to run now?
- Runnable task, say T2, which has the lowest virtual runtime (vruntime) among the tasks contending for the processor
  - vruntime (in nanosecond): records how long a task has run on the process
- Scheduler tracks the vruntime for all tasks
  - Runnable and blocked
- Lower a task's vruntime, more deserving the task is for the time on the processor

- vruntime = actual runtime normalized by the number of runnable processes

Rescheduling

- Rescheduling is required when the number of tasks , N, becomes such that
  - N > Target Latency / Minimum Granularity
- Now scheduler changes the Target latency to
  - N * minimum latency

Summary: Choosing the Right Scheduler

If You Care About:	Then Choose:
CPU Throughput	FCFS
Average Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness (Wait Time to Get CPU)	Round Robin
Meeting Deadlines	Earliest Deadline First (EDF)
Favoring Important Tasks	Priority

Backup Slides

Choosing Time-slice

- The scheduler policy must dictate a default timeslice
  - Not easy to choose this number
- Too long timeslice □ poor interactive performance
- Too short timeslice □ Overhead of switching processes
- I/O bound processes don’t need longer timeslice
- Linux’s CFS scheduler, however, does not directly assign timeslices to processes
  - CFS assigns processes a *proportion* of the processor
    - the amount of processor time that a process receives is a function of the load of the system
  - This assigned proportion is further affected by each process’s nice value
    - Acts as a weight, changing the proportion of the processor time each process receives

Time Slice

- The rule of thumb adopted by Linux is: choose a duration as long as possible, while keeping good system response time

- Which process to run ?
  - Decision also depends on how much proportion of the processor the process has consumed
    - If it has consumed small proportion of the processor, it is run immediately