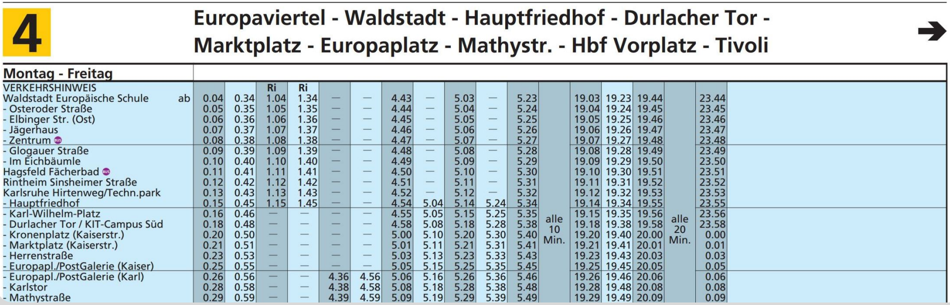
Karlsruhe Institute of Technology

Operating Systems

03. Process Scheduling

Prof. Dr. Frank Bellosa | WT 2020/2021

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – ITEC – OPERATING SYSTEMS

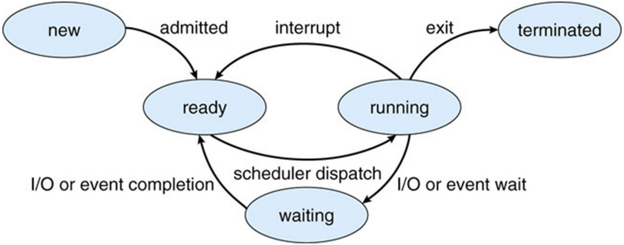


4 Europaviertel - Waldstadt - Hauptfriedhof - Durlacher Tor - Marktplatz - Europaplatz - Mathystr. - Hbf Vorplatz - Tivoli ➔

Montag - Freitag																	
VERKEHRSHINWEIS				Ri	Ri												
Waldstadt Europäische Schule	ab	0.04	0.34	1.04	1.34	—	—	4.43	—	5.03	—	5.23	alle 10 Min.	19.03	19.23	19.44	alle 20 Min.	23.44
- Osteroder Straße		0.05	0.35	1.05	1.35	—	—	4.44	—	5.04	—	5.24		19.04	19.24	19.45		23.45
- Elbinger Str. (Ost)		0.06	0.36	1.06	1.36	—	—	4.45	—	5.05	—	5.25		19.05	19.25	19.46		23.46
- Jägerhaus		0.07	0.37	1.07	1.37	—	—	4.46	—	5.06	—	5.26		19.06	19.26	19.47		23.47
- Zentrum BUS		0.08	0.38	1.08	1.38	—	—	4.47	—	5.07	—	5.27		19.07	19.27	19.48		23.48
- Glogauer Straße		0.09	0.39	1.09	1.39	—	—	4.48	—	5.08	—	5.28		19.08	19.28	19.49		23.49
- Im Eichbäumle		0.10	0.40	1.10	1.40	—	—	4.49	—	5.09	—	5.29		19.09	19.29	19.50		23.50
Hagsfeld Fächerbad BUS		0.11	0.41	1.11	1.41	—	—	4.50	—	5.10	—	5.30		19.10	19.30	19.51		23.51
Rintheim Sinsheimer Straße		0.12	0.42	1.12	1.42	—	—	4.51	—	5.11	—	5.31		19.11	19.31	19.52		23.52
Karlsruhe Hirtenweg/Techn.park		0.13	0.43	1.13	1.43	—	—	4.52	—	5.12	—	5.32		19.12	19.32	19.53		23.53
- Hauptfriedhof		0.15	0.45	1.15	1.45	—	—	4.54	5.04	5.14	5.24	5.34		19.14	19.34	19.55		23.55
- Karl-Wilhelm-Platz		0.16	0.46	—	—	—	—	4.55	5.05	5.15	5.25	5.35		19.15	19.35	19.56		23.56
- Durlacher Tor / KIT-Campus Süd		0.18	0.48	—	—	—	—	4.58	5.08	5.18	5.28	5.38		19.18	19.38	19.58		23.58
- Kronenplatz (Kaiserstr.)		0.20	0.50	—	—	—	—	5.00	5.10	5.20	5.30	5.40		19.20	19.40	20.00		0.00
- Marktplatz (Kaiserstr.)		0.21	0.51	—	—	—	—	5.01	5.11	5.21	5.31	5.41		19.21	19.41	20.01		0.01
- Herrenstraße		0.23	0.53	—	—	—	—	5.03	5.13	5.23	5.33	5.43		19.23	19.43	20.03		0.03
- Europapl./PostGalerie (Kaiser)		0.25	0.55	—	—	—	—	5.05	5.15	5.25	5.35	5.45		19.25	19.45	20.05		0.05
- Europapl./PostGalerie (Karl)		0.26	0.56	—	—	4.36	4.56	5.06	5.16	5.26	5.36	5.46		19.26	19.46	20.06		0.06
- Karlstor		0.28	0.58	—	—	4.38	4.58	5.08	5.18	5.28	5.38	5.48		19.28	19.48	20.08		0.08
- Mathystraße		0.29	0.59	—	—	4.39	4.59	5.09	5.19	5.29	5.39	5.49		19.29	19.49	20.09		0.09

Process State

- From the OS perspective, a process can be in different states:
  - new: The process has been created but was never run
  - ready: The process is waiting to be assigned to a processor
  - running: Instructions are currently being executed
  - waiting: The process is waiting for some event to occur
  - terminated: The process has finished execution (zombie state)



[SGG12]

Which jobs should be assigned to which CPU(s)?

- The dispatcher performs the actual process switch (mechanism)
  - Saving/restoring process context
  - Switching to user mode

- The CPU scheduler selects the next process to run (policy)
  - Schedulers try to
    - meet goals (e.g., efficiency, deadlines, physical environmental conditions)
    - provide fairness
    - adhere to priorities

Scheduling Problem

- Have $\mathtt{K}$ jobs ready to run
  - Jobs can be processes or threads
- Have $\mathtt{N}$ CPUs with: $\mathtt{K} > \mathtt{N} \geq \mathtt{1}$ CPUs
- Scheduling Problem
  - Which jobs should the kernel assign to which CPUs?
  - When should it make the decision?

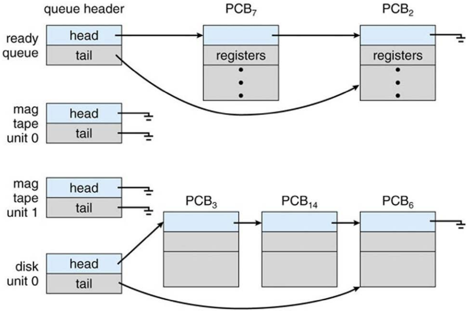
Different Schedulers

- Short-term scheduler
  - Selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked very frequently (milliseconds)
    → must be fast
- Long-term scheduler (or job scheduler)
  - Selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked very infrequently (seconds, minutes)
    → can be slow
  - The long-term scheduler controls the degree of multiprogramming
- We focus on the short-term scheduler in this lecture

Process Scheduling Queues

- Job queue: Set of all processes in the system
- Ready queue: Processes in main memory, state: ready and waiting
- Device queues: Processes waiting for an I/O device

Processes migrate among the various queues



[SGG12]

Scheduling Policies

Categories of Scheduling Policies

Different scheduling policies are needed in different environments

- Batch Scheduling
  - Still widespread in business and HPC . . .
  - No users waiting for a quick response
  - Non-preemptive algorithms acceptable → less switches → less overhead
- Interactive Scheduling
  - Need to optimize for response time
  - Preemption essential to keep processes from hogging CPU
- Real-Time Scheduling
  - Guarantee completion of jobs within time constraints
  - Need to be able to plan when which process runs and how long
  - Preemption is not always needed and is part of WCET calculation

Scheduling Goals Vary for Different Categories

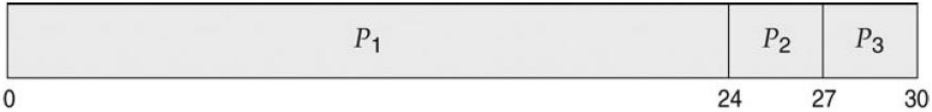
- All Systems
  - Fairness give each process a fair share of CPU
  - Resource utilization keep expensive devices busy
  - OS overhead e.g., reduce number of context switches
- Batch Scheduling
  - Throughput # of processes that complete per time unit
  - Turnaround time time from submission to completion of a job
  - CPU utilization keep the CPU as busy as possible
- Interactive Scheduling
  - Waiting time time each process waits in ready queue
  - Response time time from request to first response
    - For a job: e.g., key press to echo
    - For a scheduler: submission of a job to the first time it is dispatched
- Real-Time Scheduling
  - Meeting Deadlines finish jobs in time
  - Predictability minimize jitter

First-Come, First-Served (FCFS) Scheduling

- FCFS: Schedule the processes in the order of arrival
- Suppose that 3 processes arrive in the order: P_1, P_2, P_3 (at time 0) [SGG12]

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt Chart for the schedule is:

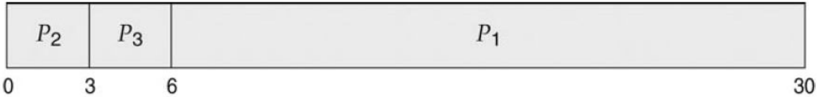


- Turnaround times: $P_1 = 24$, $P_2 = 27$, $P_3 = 30$
- Average turnaround time: $\frac{24+27+30}{3} = 27$ → Can we do better?

First-Come, First-Served (FCFS) Scheduling

- Suppose that the 3 processes arrived in the order: P_2, P_3, P_1 (at time 0) [SGG12]

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Turnaround times: $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Average turnaround time: $\frac{30+3+6}{3} = 13$
  → Much better than the previous 27

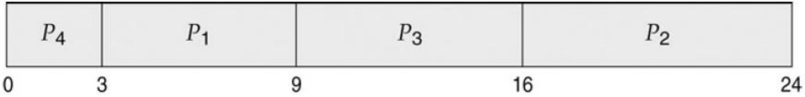
Average turnaround time depends on arrival in queue

Shortest-Job-First (SJF) Scheduling

- FCFS is prone to Convoy effect
  - All short ("fast") jobs now have to wait for the first (long) job to finish

  → Idea: Run shortest jobs first (SJF) [SGG12]
- SJF has optimal average turnaround (and waiting, and response) times
  - Assume sorted jobs by SJF: make formula for average turnaround time
  - Switch any two jobs j, k, where j<k → longer job now earlier
  - Contradiction: Average turnaround time larger (subtract times)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

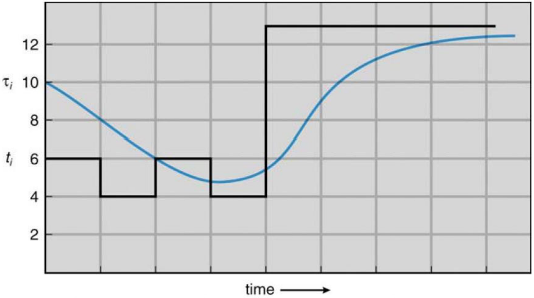


SJF: Job Length Prediction

- Challenge: Cannot know job lengths in advance
- Solution: Predict length of next CPU burst for each process
  → Schedule the process with the shortest burst next
  - Now suboptimal turnaround time possible
    (e.g., longest job has shortest bursts)
  - Still optimizes waiting and response times

SJF: Estimating the Length of Next CPU Burst

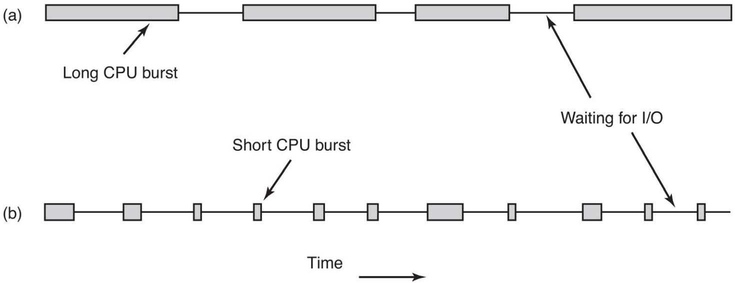
- Length of previous CPU bursts → exponential averaging
  - t_n = actual length of n^{th} CPU burst
  - τ_{n+1} = predicted value for the next CPU burst
  - Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, with $0 \leq \alpha \leq 1$
- Example: $\alpha = 0.5$ [SGG12]



CPU burst (t_i)		6	4	6	4	13	13	13 …
"guess" (τ_i)	10	8	6	6	5	9	11	12 …

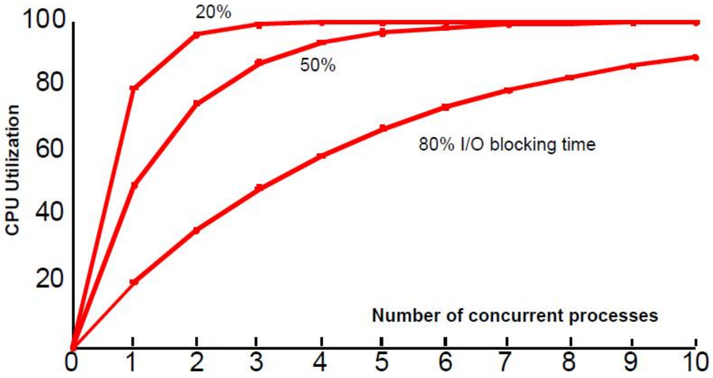
Process Behavior: Boundedness

- Processes can be characterized as:
  (a) CPU-bound process Spends more time doing computations
  → few very long CPU bursts
  (b) I/O-bound process Spends more time doing I/O than computations
  → many short CPU bursts



[TB15]

The Benefit of Multiprogramming

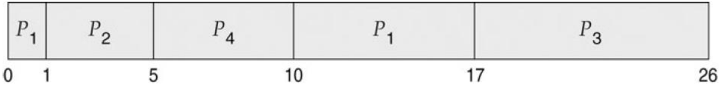


[TB15]

Preemptive Shortest-Job-First (PSJF) Scheduling

- SJF optimizes waiting time and response time (and offline also turnaround time)
- But what about throughput?
  - CPU bound jobs hold CPU until exit or I/O → poor I/O device utilization
- Idea: SJF, but preempt periodically to make a new scheduling decision
  - At each time slice schedule job with shortest remaining time next
  - Alternatively: Schedule job whose next CPU burst is the shortest [SGG12]

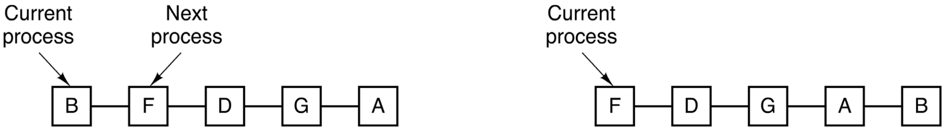
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Schedulers for Interactive Systems

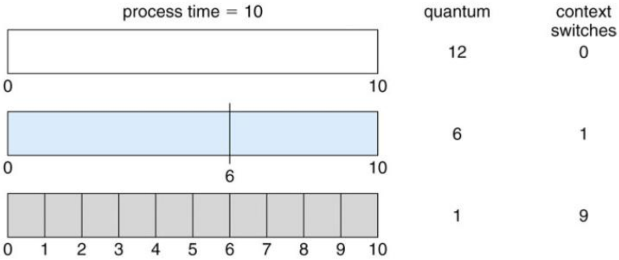
Round Robin (RR) Scheduling

- Batch schedulers suffer from starvation and do not provide fairness
- Idea: Each process runs for a small unit of CPU time
  - Time quantum/time slice length usually 10-100 milliseconds
  - Preempt processes that have not blocked by the end of the time slice
  - Append current thread to end of run queue, run next thread [TB15]



Round Robin (RR) Scheduling

- Time slice length needs to balance interactivity and overhead [SGG12]
  - Need time to dispatch new process (overhead)
  - If time slice is much larger than dispatch time
    → dispatch overhead is small compared to run-time of process
  - If the time slice length is around the dispatch time
    → 50% of CPU time is wasted for switching between processes

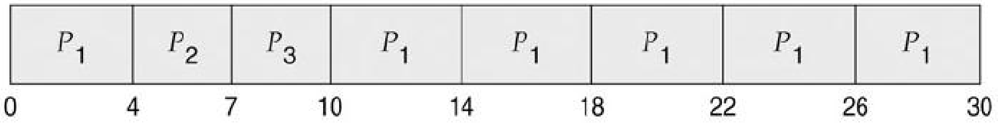


Round Robin (RR) Scheduling

- Example:
  Time slice length = 4 time units
  [SGG12]

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Gantt chart:



- Typically, higher average turnaround than SJF, but better response time
- Good average waiting time if job lengths vary

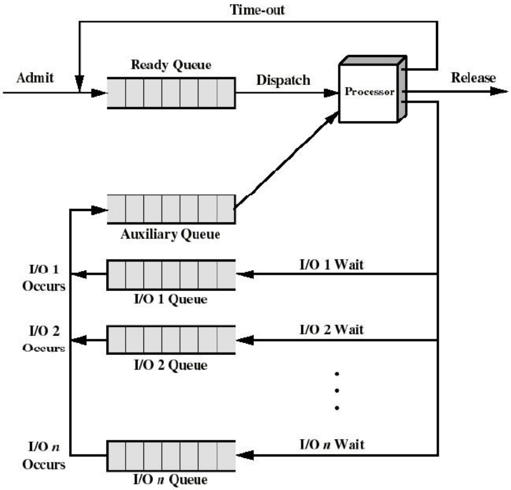
Virtual Round Robin (RR) Scheduling

- RR is unfair for I/O-bound jobs
  - I/O-bound jobs block before they use up their time quantum
  - CPU-bound jobs use up their entire quantum

  → with same number of slices, CPU-bound jobs get more CPU time

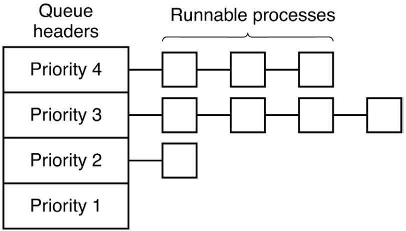
- Idea: Virtual Round Robin
  - Put jobs that didn’t use up their quantum into an additional queue
  - Store the share of the time-slice that they have not used up with the job
  - Give jobs in the additional queue priority over jobs in other queue until they have used up their quantum
  - Afterwards put them back in normal queue

Virtual Round Robin (RR) Scheduling [Sta17]



Priority Scheduling

- Not all jobs are equally important
  → Different priorities
  [TB15]
- Priority Scheduling: Associate priority number with each process
  - Allocate CPU RR to processes with the highest priority
  - Can be preemptive or non-preemptive
  - Usually: smallest integer ≡ highest priority
- SJF ≡ Priority scheduling where priority is the predicted next burst time



- Strict priority scheduling: processes with low priorities never execute if there is always a process runnable with a higher priority (starvation)
  - Possible Solution: Weaken strictness through aging
    → As time progresses increase the priority of the processes that have not run

Multi-Level Feedback Queue (MLFB) Scheduling

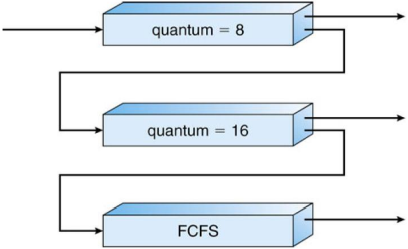
- Context switching can be expensive
  - Can we get a good trade-off between interactivity and overhead?

- Goals:
  - Give higher priority to I/O-bound jobs
    (they usually don't use up their quantum but deserve a fair CPU share)
  - Give low priority to CPU-bound jobs, but run them for longer at a time
    (rather run the job every "round" for twice the time)

- Idea: Different queues with different priorities and time slices lengths
  - Schedule queues with (static) priority scheduling
  - Double time slice length in each next-lower priority
  - Promote processes into a higher priority queue when
    they don't use up their quantum repeatedly
  - Demote processes that repeatedly use up their quantum

Multi-Level Feedback Queue (MLFB) Scheduling

- Example with three queues:



- Q_0: RR time slice length 8 ms
- Q_1: RR time slice length 16 ms
- Q_2: FCFS

- Example Scheduling:
  - A new job enters queue Q_0 which is scheduled using RR
  - When the job is dispatched, it receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - In Q_1 the job is run for additional 16 milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2

Priority Donation

- Problem: Process B may wait for result of process A
  - A has a lower priority than B

  → B has effectively lower priority now

- Solution: Priority donation (a.k.a. priority inheritance)
  - Give A priority of B as long as B waits for A
  - What if C, D and E also wait for B?
  - Should we donate priorities transitively?

  → A only gets highest priority of B, C, D, E

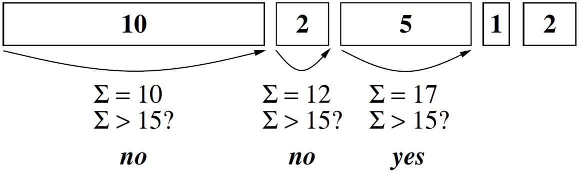
- Shouldn't A's priority increase even more if many processes wait for it?

Lottery Scheduling

- Issue number of lottery tickets to processes [WW94]
  - More tickets for processes with higher priority
  - Tickets not associated with concrete numbers
- Amount of tickets controls average proportion of CPU for each process
- ∃ a list of all runnable processes
  - A schedule operation draws a random number N and traverses the list to find the winner of the timeslice (≡ process with the N'th ticket)

total = 20

random [0 .. 19] = 15



- Processes may transfer tickets to other processes if they wait for them

Real-Time Systems

Real-Time Scheduling and WCET Analysis

- Not relevant for this lecture
- If you are interested, a good starting point is:
  - Jane W.S. Liu, “Real-Time Systems”, Prentice Hall, 2000 [Liu00]

Linux Scheduler

Linux CPU scheduler

Goals

- Fairness
- Low task response time for IO bound/interactive tasks
- High throughput for CPU bound tasks
- Low scheduling overhead
- Timeslice based on priority (manually/dynamically adjusted)
- Suitable for multi-CPU/multi-core
  - Efficiency with multiple CPUs
  - Balanced load on CPUs

"One scheduler to rule them all" –
server (network, processing), desktop, notebooks, phones, embedded, . . .

The Linux scheduler (pre-2007)

O(1) scheduler

- 140 priorities (0–99 for realtime tasks, 100–139 for user tasks)
  - Default user task priority: 120
  - „Niceness“: -20 (most favorable) to 19 (least favorable) (see `man 1 nice` and `man 2 nice`)
- Per-CPU run-queue array with one entry per priority level (140 entries)
- In fact, two entries per priority: active and expired processes
- Each entry: linked list, served in FIFO order
  - Current task moved from active to expired after timeslice
  - Swap active $\leftrightarrow$ expired if active is empty
- Timeslice depending on priority
- Bitmap (140 bit) for efficiently finding non-empty priority

The Linux scheduler (pre-2007)

Process becomes runnable

- Addition of process to priority runqueue of some CPU
  - Selection depends on load balancing and efficient cache usage
- Set bit in priority bitmask

Process becomes not runnable

- Removal of process from priority runqueue of some CPU
- Clear bit in priority bitmask (if no other same-priority task)

Priority calculation

- Static priority: “nicecess”: -20 (highest) to +19 (lowest)
- Dynamic priority: penalty for CPU bound, reward for IO bound processes

The Linux scheduler (since 2.6.63 / 2007)

- Multiple scheduling policies within scheduling classes
- Task migration between CPUs, policies, and classes
- Generic API for scheduling class
  - Enqueue task
  - Dequeue task
  - Pick next tasks
- Runqueues:
  - One runqueue instance per CPU
  - Instance contains Deadline, Realtime and Completely Fair runqueues

Scheduling classes and policies

- Stop (no policies)
- Deadline (SCHED_DEADLINE)
- Realtime (SCHED_FIFO, SCHED_RR)
- Completely Fair (SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE)
- Idle (no policies)

Scheduling class: Stop

- Highest priority
- Only used in SMP
- Single “migration/N” kernel thread per CPU
- Used for task migration, CPU hotplugging, etc.

Scheduling class: Deadline

- Highest priority (after Stop)
- Since 2013 (v3.14)
- Based on Earliest Deadline First (EDF) scheduling
  - Tasks can declare their required runtime and their period of execution
  - Tasks are allowed to run during an assigned budget
  - Will be suspended after full budget has been used
    → unlike Realtime tasks – see next slide – which can prevent all other lower-priority threads from running
- Can be used for periodic real-time tasks (e.g. video encoding/decoding)
- `https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html`

Scheduling class: Realtime

- POSIX real-time tasks
- Task priorities 0-99
- Two policies for tasks at same priority:
  - SCHED_FIFO
  - SCHED_RR, with 100ms default timeslice

- Linux command line (with root privileges)
  - `chrt -rr <level> task`

Scheduling class: CFS

CFS: Completely fair scheduler

- Scheduling policies:
  - SCHED_NORMAL: normal Linux tasks
  - SCHED_BATCH: non-interactive batch tasks
  - SCHED_IDLE: low priority tasks
- Tracks virtual runtime (vruntime) of tasks (time on CPU)
- Task with shortest vruntime runs first
- Internal data structure: self-balancing red-black tree representing timeline of future task executions (insertion based on vruntime)
  - IO-bound tasks (low vruntime) get higher priority
  - CPU-bound jobs don't get more CPU time
- Priority defines weight of task in vruntime calculation
  - Higher weight $\rightarrow$ slower vruntime increase

Scheduling class: Idle

- Lowest priority scheduling class
- One idle kernel thread per CPU: “swapper/N”
- Runs only when nothing else is runnable
- May take CPU to lower power state

Multi-CPU scheduling in Linux

Runqueue per CPU core

- Core-local scheduling without cross-core synchronization (efficiency!)
- Core may be idle while other cores have jobs waiting in their queues

Load balancing

- Periodically shift load and when no tasks on some queue
- Requires exclusive access to run-queues

[LLF+16]

References I

[Liu00] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, USA, 1st edition, 2000.

[LLF+16] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[Sta17] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, USA, 9th edition, 2017.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.

References II

[WW94] Carl A Waldspurger and William E Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 1–es, 1994.