8th Slide Set
Operating Systems

Prof. Dr. Christian Baun

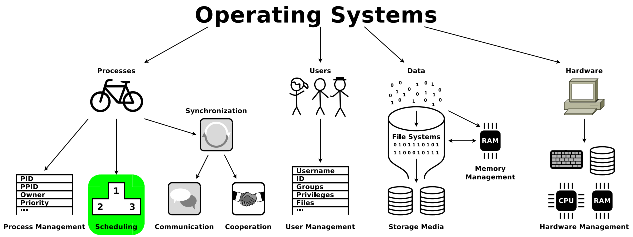
Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Faculty of Computer Science and Engineering
`christianbaun@fb2.fra-uas.de`

Learning Objectives of this Slide Set

- At the end of this slide set, you know/understand. . .
  - what steps the **dispatcher** carries out for switching between processes
  - what **scheduling** is
    - how **preemptive scheduling** and **non-preemptive scheduling** work
    - the functioning of several common **scheduling methods**
    - how **scheduling in modern operating systems** works in detail

In SS2019 I erased all scheduling algorithms (SJF/SRTF/LJF/LRTF/HRRN) from my course material that require knowing how long it takes for each process until its termination. In other words, these algorithms need to know how long is the execution time of each process. In practice, this is almost never the case ($\Longrightarrow$ **unrealistic**)

Exercise sheet 8 repeats the contents of this slide set which are relevant for these learning objectives



Process Switching – The Dispatcher (1/2)

- Tasks of multitasking operating systems are among others:
  - **Dispatching**: Switching of the CPU during a process switch
  - **Scheduling**: Determination of the point in time when the process switch occurs and of the execution order of the processes
- The **dispatcher** carries out the state transitions of the processes

We already know. . .

- During process switching, the dispatcher removes the CPU from the `running` process and assigns it to the process, which is the first one in the queue
- For transitions between the states `ready` and `blocked`, the dispatcher removes the corresponding process control blocks from the status lists and accordingly inserts them new
- Transitions from or to the state `running` always imply a switch of the process, which is currently executed by the CPU

If a process switches into the state `running` or from the state `running` to another state, the dispatcher needs to. . .

- back up the context (register contents) of the executed process in the process control block
- assign the CPU to another process
- import the context (register contents) of the process, which will be executed next, from its process control block

Process Switching – The Dispatcher (2/2)

The system idle process

- Windows operating systems since Windows NT ensure that the CPU is assigned to a process at any time
- If no process is in the state `ready`, the **system idle process** gets the CPU assigned
- The system idle process is always active and has the lowest priority
- Due to the system idle process, the scheduler must never consider the case that no active process exists
- Since Windows 2000, the system idle process puts the CPU into a power-saving mode
- For each CPU core (in hyperthreading systems for each logical CPU), exists a system idle process

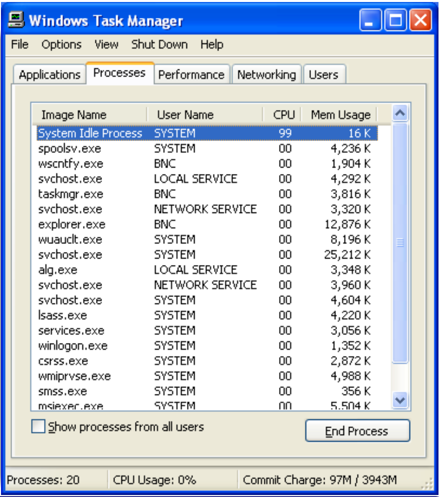

Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users

Image Name	User Name	CPU	Mem Usage
System Idle Process	SYSTEM	99	16 K
spoolsv.exe	SYSTEM	00	4,236 K
wscntfy.exe	BNC	00	1,904 K
svchost.exe	LOCAL SERVICE	00	4,292 K
taskmgr.exe	BNC	00	3,816 K
svchost.exe	NETWORK SERVICE	00	3,320 K
explorer.exe	BNC	00	12,876 K
wuauclt.exe	SYSTEM	00	8,196 K
svchost.exe	SYSTEM	00	25,212 K
alg.exe	LOCAL SERVICE	00	3,348 K
svchost.exe	NETWORK SERVICE	00	3,960 K
svchost.exe	SYSTEM	00	4,604 K
lsass.exe	SYSTEM	00	4,220 K
services.exe	SYSTEM	00	3,056 K
winlogon.exe	SYSTEM	00	1,352 K
csrss.exe	SYSTEM	00	2,872 K
wmiprvse.exe	SYSTEM	00	4,988 K
smss.exe	SYSTEM	00	356 K
msiexec.exe	SYSTEM	00	5,504 K

Processes: 20 CPU Usage: 0% Commit Charge: 97M / 3943M

`https://unix.stackexchange.com/questions/361245/what-does-an-idle-cpu-process-do`

"*In Linux, one idle task is created for every CPU and locked to that processor; whenever there's no other process to run on that CPU, the idle task is scheduled. Time spent in the idle tasks appears as "idle" time in tools such as top...*"

Scheduling Criteria and Scheduling Strategies

- During scheduling, the operating system specifies the execution order of the processes in the state `ready`
- **No scheduling strategy...**
  - **is optimally suited for each system**
  - **can take all scheduling criteria optimally into account**
    - Scheduling criteria are among others CPU load, response time (latency), turnaround time, throughput, efficiency, real-time behavior (compliance with deadlines), waiting time, overhead, fairness, consideration of priorities, even resource utilization...
- When choosing a scheduling strategy, a **compromise** between the scheduling criteria must always be found

Non-preemptive and preemptive Scheduling

- 2 classes of scheduling strategies exist
  - **Non-preemptive scheduling** or **cooperative scheduling**
    - A process, which gets the CPU assigned by the scheduler, remains in control over the CPU until its execution is finished or it voluntarily gives the control back on a voluntary basis
    - Problematic: A process may occupy the CPU for as long as it wants

Examples: Windows 3.x, MacOS 8/9, Windows 95/98/Me (for 16-Bit processes)

  - **Preemptive scheduling**
    - The CPU may be removed from a process before its execution is completed
    - If the CPU is removed from a process, it is paused until the scheduler again assigns the CPU to it
    - Drawback: Higher overhead compared with non-preemptive scheduling
    - The benefits of preemptive scheduling, especially the consideration of process priorities, outweigh the drawbacks

Examples: Linux, MacOS X, Windows 95/98/Me (for 32-Bit processes), Windows NT (incl. XP/Visa/7/8/10/11), FreeBSD

Impact on the overall Performance of a Computer

- This example demonstrates the impact of the scheduling method used on the overall performance of a computer
  - The processes P_A and P_B are to be executed one after the other

Process	CPU time
A	24 ms
B	2 ms

- If a short-running process runs before a long-running process, the runtime and waiting time of the long process process get **slightly worse**
- If a long-running process runs before a short-running process, the runtime and waiting time of the short process get **significantly worse**

Execution order	Runtime A	Runtime B	Average runtime	Waiting time A	Waiting time B	Average waiting time
P_A, P_B	24 ms	26 ms	$\frac{24+26}{2} = 25$ ms	0 ms	24 ms	$\frac{0+24}{2} = 12$ ms
P_B, P_A	26 ms	2 ms	$\frac{2+26}{2} = 14$ ms	2 ms	0 ms	$\frac{0+2}{2} = 1$ ms

Scheduling Methods

- Several scheduling methods (algorithms) exist
  - Each method tries to comply with the well-known scheduling criteria and principles in varying degrees
- Some scheduling methods:
  - **Priority-driven scheduling**
  - **First Come First Served** (FCFS) = **First In First Out** (FIFO)
  - ~~**Last Come First Served** (LCFS)~~
  - **Round Robin** (RR) with time quantum
  - ~~**Shortest/Longest Job First** (SJF/LJF)~~
  - ~~**Shortest/Longest Remaining Time First** (SRTF/LRTF)~~
  - ~~**Highest Response Ratio Next** (HRRN)~~
  - **Earliest Deadline First** (EDF)
  - **Fair-share scheduling**
  - ~~**Static multilevel scheduling**~~
  - **Multilevel feedback scheduling**
  - **Completely Fair Scheduler** (CFS)
  - **Earliest Eligible Virtual Deadline First** (EEVDF)

Modern operating systems often implement several scheduling methods

- In Linux e.g. each process is assigned to a specific scheduling method
- For **“real-time” processes**. . .
  - `SCHED_FIFO` (priority-driven scheduling, non-preemptive)
  - `SCHED_RR` (preemptive)
  - `SCHED_DEADLINE` (EDF scheduling, preemptive)
- For **“normal” processes**. . .
  - `SCHED_OTHER` (default Linux time-sharing scheduling) is implemented as:
    - Multilevel Feedback Scheduling (until Kernel 2.4)
    - O(1) scheduler (Kernel 2.6.0 until 2.6.22)
    - Completely Fair Scheduler (Kernel 2.6.23 until Kernel 6.5.13)
    - Earliest Eligible Virtual Deadline First scheduler (since Kernel 6.6)

```
$ ps a | grep okular
 359675 pts/2    Sl     0:04 okular bts_WS2122_slideset_08_en.pdf
$ chrt -p 359675
pid 359675's current scheduling policy: SCHED_OTHER
pid 359675's current scheduling priority: 0
```

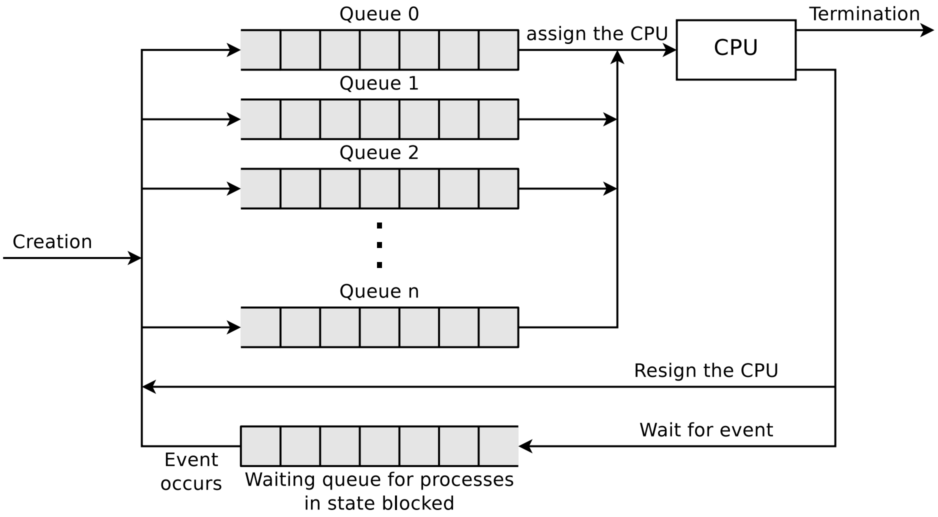
```
SCHED_OTHER:      chrt -o -p PRIO PID
SCHED_FIFO:       chrt -f -p PRIO PID
SCHED_RR:         chrt -r -p PRIO PID
SCHED_DEADLINE:   chrt -d -sched-runtime NS -sched-deadline NS -sched-period NS 0 PID
```

“A SCHED_DEADLINE task should receive *runtime* microseconds of execution time every *period* microseconds, and these *runtime* microseconds are available within *deadline* microseconds from the beginning of the period.”
Source: `https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt`

Priority-driven Scheduling

- Processes are executed according to their priority (= importance or urgency)
- The highest priority process in state `ready` gets the CPU assigned
    - The priority may depend on various criteria, such as static (assigned) priority level, required resources, rank of the user, demanded real-time criteria,. . .
- Can be **preemptive** and **non-preemptive**
- The priority values can be assigned **static** or **dynamic**
    - Static priorities remain unchanged throughout the lifetime of a process, and are often used in real-time systems
    - Dynamic priorities are adjusted from time to time $\Longrightarrow$ **Multilevel feedback scheduling** (see slide 21)
- Risk of (static) priority-driven scheduling: Processes with low priority values may starve ($\Longrightarrow$ **this is not fair**)
- Priority-driven scheduling can be used for interactive systems

Priority-driven Scheduling

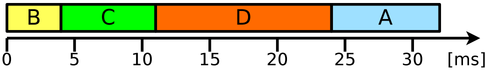


Source: William Stallings. Operating Systems. 4th edition. Prentice Hall (2001). P.401

Priority-driven Scheduling – Example

- 4 processes shall be processed on a single CPU/core system
- All processes are at time point 0 in state `ready`
- Execution order of the processes as Gantt chart (timeline)

Process	CPU time	Priority
A	8 ms	3
B	4 ms	15
C	7 ms	8
D	13 ms	4



- The CPU time is the time that the process needs to access the CPU to complete its execution
- Runtime = “lifetime” = time period between the creation and the termination of a process = (CPU time + waiting time)

Runtime of the processes

Process	A	B	C	D
Runtime	32	4	11	24

Avg. runtime $= \frac{32+4+11+24}{4} = 17.75$ ms

Waiting time of the processes

Process	A	B	C	D
Waiting time	24	0	4	11

Avg. waiting time $= \frac{24+0+4+11}{4} = 9.75$ ms

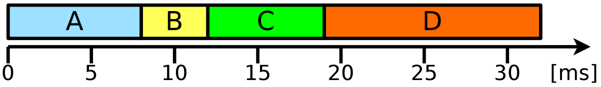
First Come First Served (FCFS)

- Works according to the principle **First In First Out** (FIFO)
- Running processes are not interrupted
  - It is **non-preemptive scheduling**
- FCFS is **fair**
  - All processes are executed
- The **average waiting time may be very high** under certain circumstances
  - Processes with short execution time may need to wait for a long time if processes with long execution times have arrived before
- FCFS/FIFO can be used for batch processing ($\Longrightarrow$ slide set 1)
- FIFO is used in Linux for non-preemptive “real-time” processes

First Come First Served – Example

- 4 processes shall be processed on a single CPU/core system
- Execution order of the processes as Gantt chart (timeline)

Process	**CPU time**	**Creation time**
A	8 ms	0 ms
B	4 ms	1 ms
C	7 ms	3 ms
D	13 ms	5 ms



- The CPU time is the time that the process needs to access the CPU to complete its execution
- Runtime = "lifetime" = time period between the creation and the termination of a process = (CPU time + waiting time)

Runtime of the processes

Process	**A**	**B**	**C**	**D**
Runtime	8	11	16	27

Avg. runtime $= \frac{8+11+16+27}{4} = 15.5$ ms

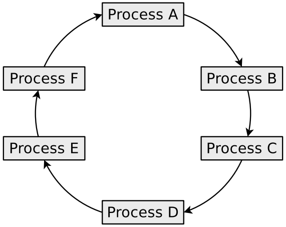
Waiting time of the processes

Process	**A**	**B**	**C**	**D**
Waiting time	0	7	9	14

Avg. waiting time $= \frac{0+7+9+14}{4} = 7.5$ ms

Round Robin – RR (1/2)

- Time slices with a fixed duration are specified
- The processes are queued in a cyclic queue according to the FIFO principle
  - The first process in the queue gets the CPU assigned for the duration of a time slice
  - After the expiration of the time slice, the process gets the CPU reassigned and it is positioned at the end of the queue
  - Whenever a process is completed successfully, it is removed from the queue
    - New processes are inserted at the end of the queue
- The CPU time is distributed **fairly** among the processes
- RR with time slice size ∞ behaves like FCFS



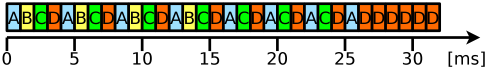
Round Robin – RR (2/2)

- The longer the execution time of a process is, the more rounds are required for its complete execution
- The size of the time slices influences the performance of the system
  - The shorter they are, the more process switches must take place $\Longrightarrow$ Increased overhead
  - The longer they are, the more simultaneousness gets lost $\Longrightarrow$ The system hangs/becomes *jerky*
- The size of the time slices is usually in single or double-digit millisecond range
- **Prefers processes with short execution time**
- **Preemptive scheduling method**
- Round Robin scheduling can be used for interactive systems
- Round Robin is used in Linux for preemptive “real-time” processes

Round Robin – Example

- 4 processes shall be processed on a single CPU/core system
- All processes are at time point 0 in state `ready`
- Time quantum $q = 1$ ms
- Execution order of the processes as Gantt chart (timeline)

Process	CPU time
A	8 ms
B	4 ms
C	7 ms
D	13 ms



- The CPU time is the time that the process needs to access the CPU to complete its execution
- Runtime = "lifetime" = time period between the creation and the termination of a process = (CPU time + waiting time)

Runtime of the processes

Process	A	B	C	D
Runtime	26	14	24	32

Avg. runtime $= \frac{26+14+24+32}{4} = 24$ ms

Waiting time of the processes

Process	A	B	C	D
Waiting time	18	10	17	19

Avg. waiting time $= \frac{18+10+17+19}{4} = 16$ ms

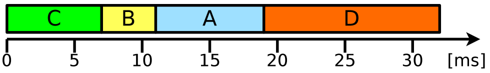
Earliest Deadline First (EDF)

- Objective: Processes should comply with their (deadlines) when possible
- Processes in `ready` state are **arranged according to their deadline**
  - The process with the closest deadline gets the CPU assigned next
- The queue is reviewed and reorganized whenever. . .
  - a new process switches into state `ready`
  - or an active process terminates
- Can be implemented as **preemptive and non-preemptive scheduling**
  - Preemptive EDF can be used in real-time operating systems
  - Non-preemptive EDF can be used for batch processing
- EDF is used in Linux for preemptive "real-time" processes

Earliest Deadline First – Example

- 4 processes shall be processed on a single CPU/core system
- All processes are at time point 0 in state `ready`
- Execution order of the processes as Gantt chart (timeline)

Process	CPU time	Deadline
A	8 ms	25
B	4 ms	18
C	7 ms	9
D	13 ms	34



- The CPU time is the time that the process needs to access the CPU to complete its execution
- Runtime = "lifetime" = time period between the creation and the termination of a process = (CPU time + waiting time)

Runtime of the processes

Process	A	B	C	D
Runtime	19	11	7	32

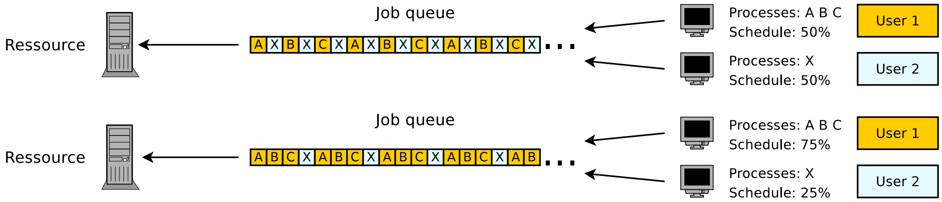
Avg. runtime $= \frac{19+11+7+32}{4} = 17.25$ ms

Waiting time of the processes

Process	A	B	C	D
Waiting time	11	7	0	19

Avg. waiting time $= \frac{11+7+0+19}{4} = 9.25$ ms

Fair-Share



- **Fair-Share** distributes the available resources among groups of processes in a fair manner
- Special feature:
  - The computing time is allocated to the users and not the processes
  - The computing time, which is allocated to a user, is independent from the number of its processes
- Users get resource shares

Fair share is often used in cluster and grid systems

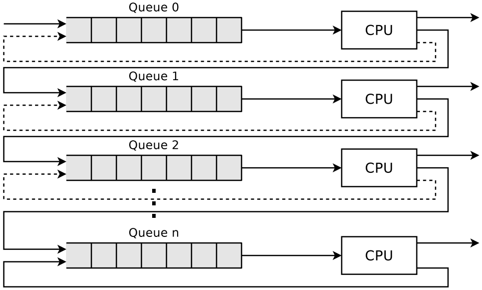
Fair share is implemented in job schedulers and meta-schedulers (e.g. SUN/Oracle/Univa/Altair Grid Engine) for assigning the jobs to resources in grid sites and distributing jobs among grid sites

Multilevel Feedback Scheduling (1/2)

- It is **impossible to predict the execution time precisely in advance**
    - Solution: Processes, which utilized much execution time in the past, get **sanctioned**
- **Multilevel feedback scheduling** works with multiple queues
    - Each queue has a different priority or time multiplex (e.g. 70%:15%:10%:5%)
- Each new process is added to the top queue
    - This way, it has the highest priority
- Each queue uses Round Robin
    - If a process returns the CPU voluntarily, it is added to the same queue again
    - If a process utilized its entire time slice, it is inserted into the next lower queue, which has a lower priority
        - The priorities are therefore **dynamically** assigned with this method
- Multilevel feedback scheduling is **preemptive scheduling**

Multilevel Feedback Scheduling (2/2)

- Benefit: **No complicated estimations!**
    - New processes are quickly assigned to a priority category
- **Prefers new processes** over older (longer-running) processes



- Processes with many input and output operations are preferred because they are inserted back into the original queue again when they voluntarily resign the CPU on voluntary basis

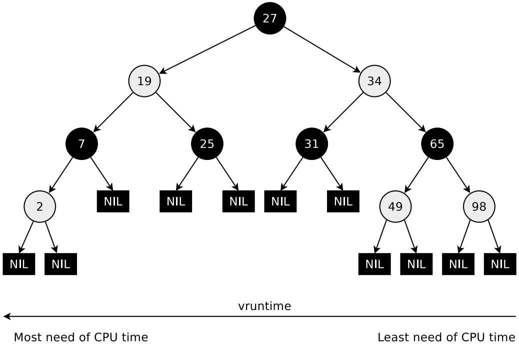
  $\Longrightarrow$ This way, they keep their priority value
- Older, longer-running processes are delayed

Source: William Stallings. Operating Systems. 4th edition. Prentice Hall (2001). P.413

Many modern operating systems use variants of multilevel feedback scheduling for the scheduling of the processes.
Examples: Linux for “normal” processes (until Kernel 2.4), Mac OS X, FreeBSD, NetBSD and the Windows NT family

Completely Fair Scheduler (Linux since 2.6.23) – Part 1/4

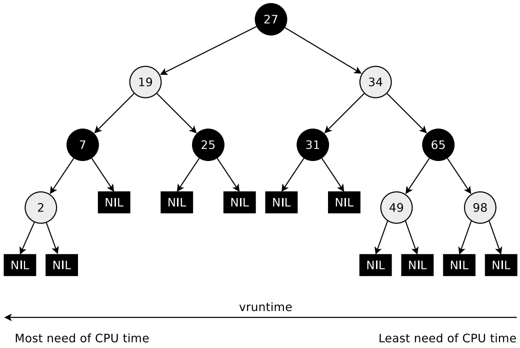
- The kernel implements a CFS scheduler for every CPU core and maintains a variable **vruntime** (virtual runtime) for every `SCHED_OTHER` process
  - The value represents a virtual processor runtime in nanoseconds



- vruntime indicates how long the particular process has already used the CPU core
  - The process with the lowest vruntime gets access to the CPU core next
- The management of the processes is done using a **red-black tree** (self-balancing binary search tree)
  - The processes are sorted in the tree by their vruntime values

Completely Fair Scheduler (Linux since 2.6.23) – Part 2/4

- Aim: All processes should get a similar (fair) share of computing time of the CPU core they are assigned to $\Longrightarrow$ For n processes, each process should get $1/n$ of the CPU time

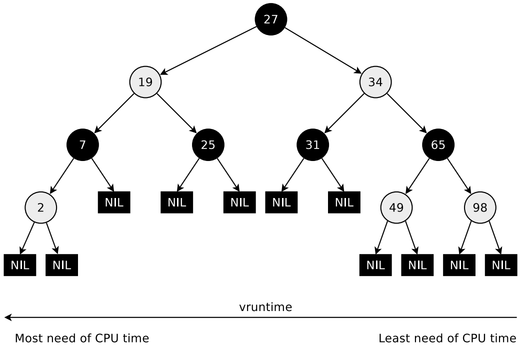


- If a process got the CPU core assigned, it can run until its vruntime value has reached the targeted portion of $1/n$ of the available CPU time
- The scheduler aims for an equal vruntime value for all processes

The CFS scheduler only takes care of the scheduling of the "normal" (non-real-time) processes that are assigned to the scheduling method SCHED_OTHER

Completely Fair Scheduler (Linux since 2.6.23) – Part 3/4

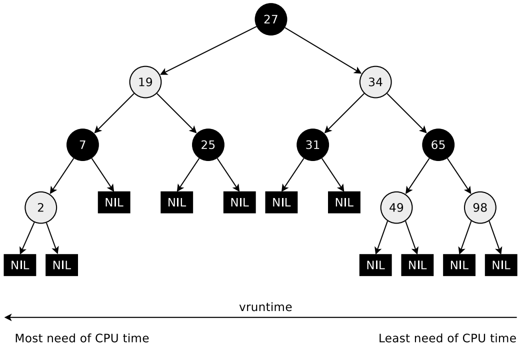
- The values are the keys of the inner nodes
- Leaf nodes (NIL nodes) have no keys and contain no data
- NIL stands for *none*, *nothing*, *null*, which means it is a null value or null pointer



- For fairness reasons, the scheduler assigns the CPU core next to the leftmost process in the tree
- If a process gets replaced from the CPU core, the vruntime value is increased by the time the process ran on the CPU core

Completely Fair Scheduler (Linux since 2.6.23) – Part 4/4

- The nodes (processes) in the tree move continuously from right to left
  $\Longrightarrow$ fair distribution of CPU resources



- The scheduler takes into account the static process priorities (`nice` values) of the processes
- The vruntime values are weighted differently depending on the `nice` value
  - In other words: The virtual clock can run at different speeds

Earliest Eligible Virtual Deadline First – Part 1/4

- The Linux kernel since v6.6 uses the EEVDF scheduler instead of CFS
- EEVDF combines the fairness concept from CFS (Completely Fair Scheduler) with deadline-based scheduling like EDF (Earliest Deadline First) from real-time systems
- **CFS and EEVDF both. . .**
  - aim to provide all processes a **similar (fair) share** of computing time of the CPU core they are assigned to
  - use the static process priorities (`nice` values) to let the **virtual clock (`vruntime`) run at different speeds**
- EEVDF introduces some new values for every process:
  - **lag**, **eligibility**, **eligible time**, **virtual deadline**

Some sources worth reading about EEVDF

An EEVDF CPU scheduler for Linux. *Jonathan Corbet* (March 2023). `https://lwn.net/Articles/925371/`
EEVDF Patch Notes. *Kuanch* (August 2024). `https://hackmd.io/@Kuanch/eevdf`
Thinking about eevdf. *Chunyu* (August 2024). `https://chunyu.sh/blog/thinking-about-eevdf/`

Earliest Eligible Virtual Deadline First – Part 2/4

- With EEVDF, the kernel maintains a **lag** value for every process
  - The lag of a process is the difference between the ideal (calculated) CPU time the process should have gotten and the CPU time it got
  - Negative lag $\Longrightarrow$ too much CPU time has been allocated to the process
  - Positive lag $\Longrightarrow$ the process has not received its fair share of CPU time
    - Only processes with a positive lag value are **eligible** to run
  - Motivation for maintaining the lag and eligibility: **More fairness**

Source: `https://chunyu.sh/blog/thinking-about-eevdf/`

Process lag = Process current weighted vruntime - weighted average of every process's vruntime

- The next slide includes an example that visualizes the calculation of the process lag
  - In the example, 3 processes are assigned to the same CPU core and start at the same time
    $\Longrightarrow$ Initially, they all have a lag of value zero

Earliest Eligible Virtual Deadline First – Part 3/4

Source of this example: **Completing the EEVDF scheduler**. *Jonathan Corbet* (April 2024). `https://lwn.net/Articles/969062/`

Process	A	B	C
lag [ms]	0	0	0
Eligible	yes	yes	yes

- No process has a negative lag $\Longrightarrow$ all are eligible
- We assume: All processes have the same static priority (`nice` value) and the time slice length for every process is 30 ms
- We assume: The scheduler decides A runs first, and A runs for the full time slice

Process	A	B	C
lag [ms]	-20	10	10
Eligible	no	yes	yes

- Each process gets 1/3 of the total CPU time (10 ms of 30 ms)
  $\Longrightarrow$ A did run 30 ms, so its lag is -20 ms
  $\Longrightarrow$ B and C did not run and have now 10 ms lag each
- We assume: The scheduler decides B runs next, and B runs for the full time slice

Process	A	B	C
lag [ms]	-10	-10	20
Eligible	no	no	yes

- Each process gets 1/3 of the total CPU time (10 ms of 30 ms)
  $\Longrightarrow$ B did run 30 ms, so its lag is -10 ms
  $\Longrightarrow$ C did not run and has now 20 ms lag
- The scheduler's next decision will be C

The sum of all lag values of the processes that are assigned to a CPU core is always value zero

Earliest Eligible Virtual Deadline First – Part 4/4

- EEVDF maintains a **virtual deadline** for every process
  - The process with the earliest deadline that is eligible to run will run next
    - The virtual deadline is calculated using the **eligible time** for the process and its **time slice length** (depends of the static priority = `nice` value)
- **Eligible time**
  - Remember: Processes with a...
    - positive lag are eligible to run
    - negative lag got too much CPU time allocated and are ineligible to run
  - EEVDF calculates for every ineligible process its **eligible time**, which is the time when it will become eligible again
- Reason for maintaining the virtual deadline: **Better latency** for real-time and interactive processes

EEVDF aims to be fairer and offer better latency compared to CFS

Classic and modern Scheduling Methods

	Scheduling NP	Scheduling P	Fair	CPU time must be known	Takes priorities into account
Priority-driven scheduling	X	X	no	no	yes
First Come First Served = FIFO	X		yes	no	no
~~Last Come First Served~~	~~X~~	~~X~~	~~no~~	~~no~~	~~no~~
Round Robin		X	yes	no	no
~~Shortest/Longest Job First~~	~~X~~		~~no~~	~~yes~~	~~no~~
~~Shortest Remaining Time First~~		~~X~~	~~no~~	~~yes~~	~~no~~
~~Longest Remaining Time First~~		~~X~~	~~no~~	~~yes~~	~~no~~
~~Highest Response Ratio Next~~	~~X~~		~~yes~~	~~yes~~	~~no~~
Earliest Deadline First	X	X	yes	no	no
Fair-share		X	yes	no	no
~~Static multilevel scheduling~~		~~X~~	~~no~~	~~no~~	~~yes (static)~~
Multilevel feedback scheduling		X	yes	no	yes (dynamic)
~~O(1)-Scheduler~~		~~X~~	~~yes~~	~~no~~	~~yes~~
Completely Fair Scheduler		X	yes	no	yes
Earliest Eligible Virtual Deadline First		X	yes	no	yes

- NP = non-preemptive scheduling, P = preemptive scheduling
- A scheduling method is “fair” when each process gets the CPU assigned at some point
- It is impossible to calculate the execution time precisely in advance

Linux 2.6.0 until 2.6.22 implements the **O(1) scheduler**. It does not play a role here for time reasons
`https://www.ibm.com/developerworks/library/l-scheduler/index.html`