FRANKFURT UNIVERSITY OF APPLIED SCIENCES

OPERATING SYSTEMS

Scheduling

Prof. Dr. Oliver Hahm

2024-12-12

AGENDA

- Process Switching
  - Dispatcher
  - Scheduling
- Scheduling Policies (Algorithms)

What does the OS need to implement in order to enable multitasking?

PROCESS SWITCHING

DISPATCHER

DISPATCHING AND SCHEDULING

- Tasks of **multitasking** OS are among others:
  - **Dispatching**: Assign the CPU to another process (process switching)
  - **Scheduling**: Determine the order of process execution and the exact point in time when the process switch occurs
- The dispatcher carries out the state transitions of the processes
- The scheduler determines these transitions happen

PERFORMANCE CONSIDERATIONS

- The **scheduler** may run …
  - periodically (e.g., on Linux)
  - for every interrupt (e.g., on RIOT)
- Is called frequently and hence, should be as efficient as possible
- Every call to the **scheduler** may trigger the **dispatcher** to run
- Must be efficient as well

What does the dispatcher have to do?

THE DISPATCHER

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

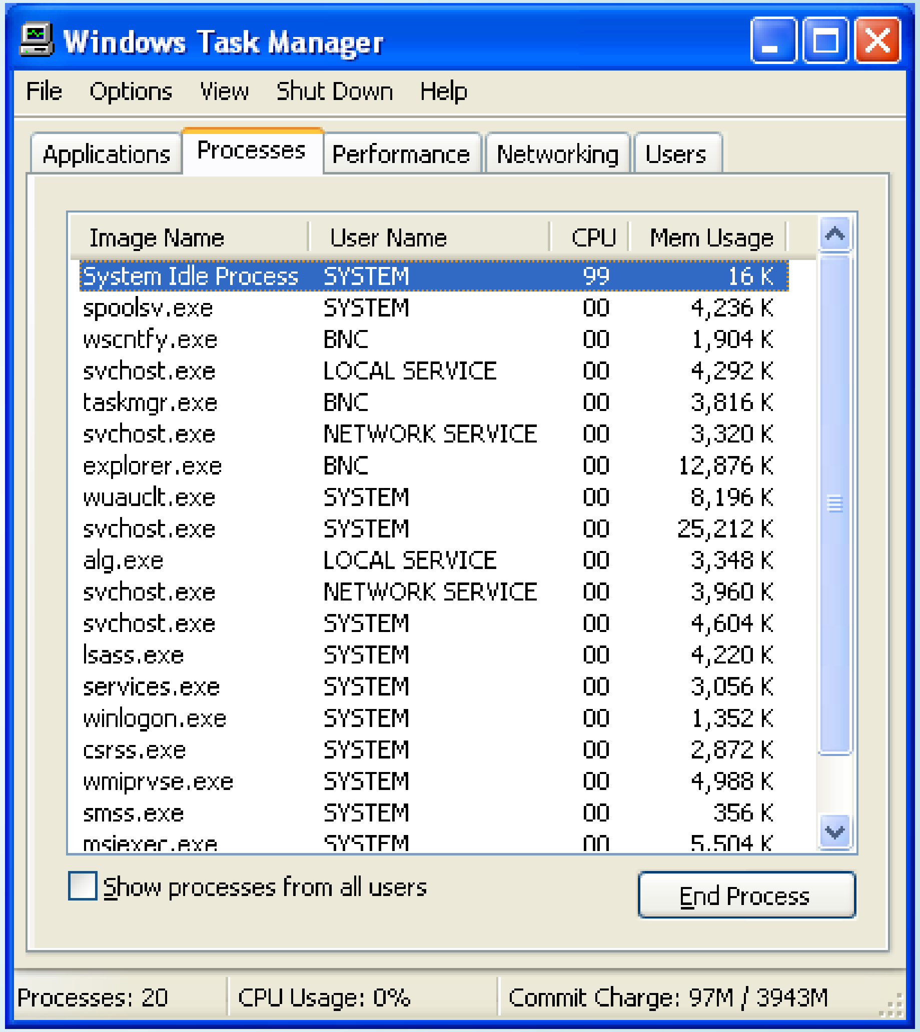
If a process switches into the state `running` or from the state `running` to another state, the dispatcher needs to...

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

Which process is executed if **no** process is on the runqueue?

IDLE PROCESS

- Many OS have a idle process
- If no process is in the state *ready* an idle process gets the CPU assigned
- The idle process is always ready to run and has the **lowest priority**
- Many modern CPU provide power-saving modes $\longrightarrow$ most OS will enter a power-saving mode when the idle process is running
- For each CPU core (in hyperthreading systems for each logical CPU) a system idle process exists



SCHEDULING

SCHEDULING CRITERIA AND SCHEDULING POLICIES

- The **scheduler** of an OS specifies the order in which the dispatcher puts the processes in the state `ready`
- The best scheduling policy (or scheduling algorithm) depends on the use case
  - No scheduling policy...
    - is optimally suited for every system and
    - can take all scheduling criteria optimal into account.
- The scheduling policy is always a tradeoff between different scheduling criteria

Scheduling criteria

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 to interrupt a running process?

NON-PREEMPTIVE AND PREEMPTIVE SCHEDULING

- Two types of scheduling policies exist
  - **Non-preemptive scheduling** or **cooperative scheduling**
    - Any running process will either run until completion or voluntarily yields
    - **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
    - **Drawback**: Higher overhead compared with non-preemptive scheduling
  - **Examples**: Linux, MacOS X, Windows 95/98/Me (for 32-Bit processes), Windows NT (incl. XP/Visa/7/8/10/11), FreeBSD, RIOT

Preemptive Scheduling in RIOT

In RIOT a running process is only removed from the run queue if a process with a higher priority becomes ready to run.

How can we measure the performance of a scheduling policy?

PERFORMANCE METRICS

Waiting Time

The time a process has to wait before getting the CPU assigned

CPU Time

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)

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

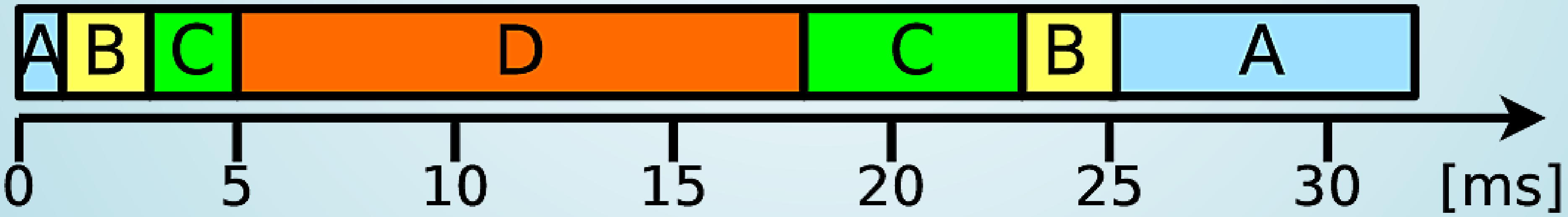
Process	CPU time
A	24 ms
B	2 ms

- 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

SCHEDULE REPRESENTATION

- The execution order of processes according to a certain scheduling strategy can be represented as a Gantt Chart
- A Gantt chart is a type of bar chart which can be used to illustrate a schedule
- Gantt charts were designed by the engineer and consultant Henry Gantt



SCHEDULING POLICIES (ALGORITHMS)

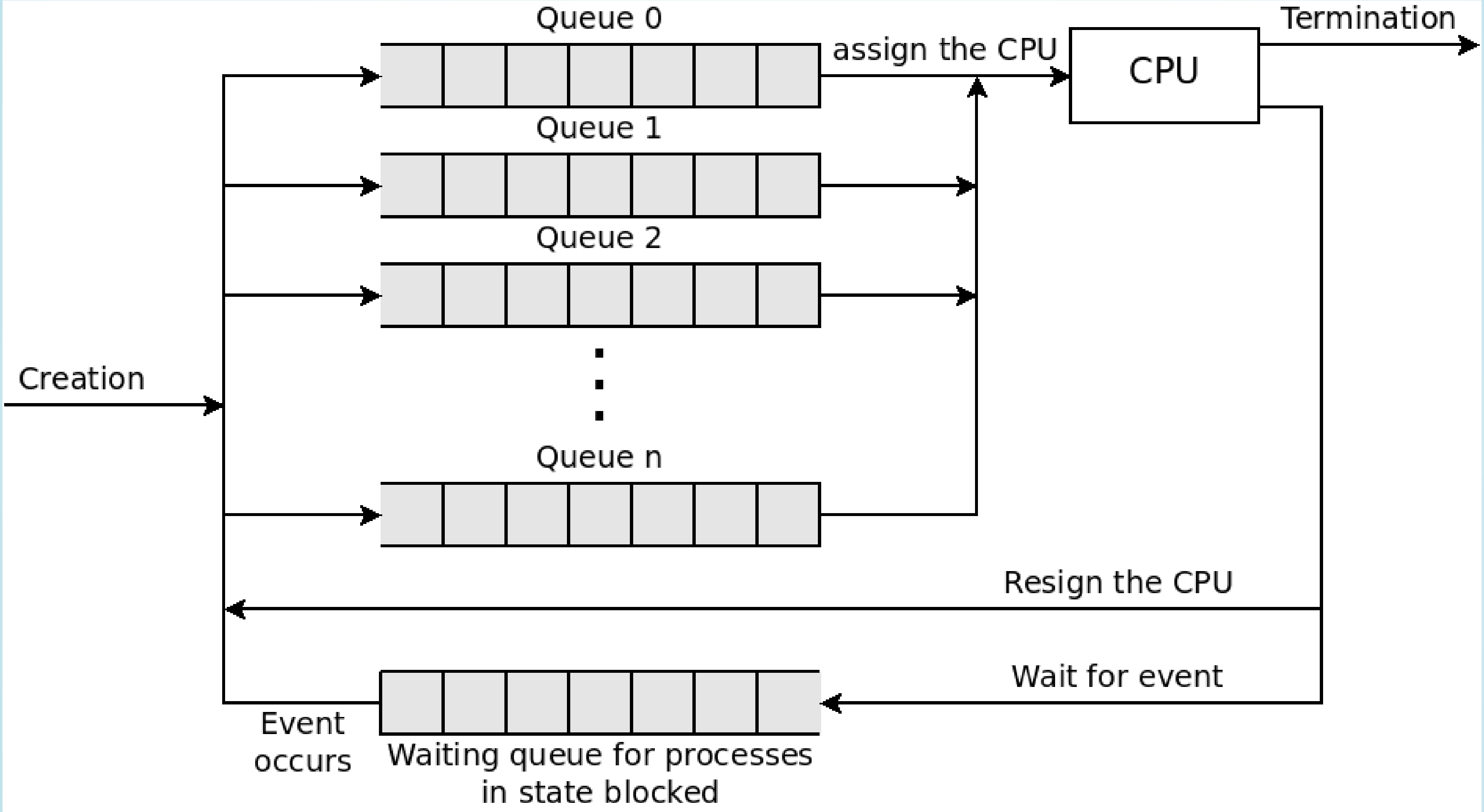
SCHEDULING POLICIES

- Several scheduling policies exist
  - Each policy tries to comply with the well-known scheduling criteria and principles in varying degrees
- Some scheduling policies:
  - 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)
  - Static multilevel scheduling
  - Multilevel feedback scheduling
  - Completely Fair Scheduler (CFS)

PRIORITY-DRIVEN SCHEDULING

- Processes are executed according to their priority (= importance or urgency)
- The highest priority process in state `ready` gets the CPU assigned
- 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 during a process' lifetime
    $\Longrightarrow$ **Multilevel feedback scheduling** (see slide)
- Risk of (static) priority-driven scheduling: Processes with low priority values may starve ($\Longrightarrow$ **this is not fair**)

PRIORITY-DRIVEN SCHEDULING

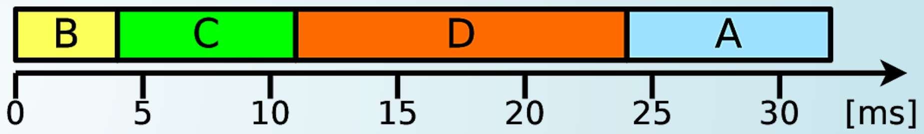


Source: William Stallings. Operating Systems. 4th edition. Prentice Hall (2001). P.401

PRIORITY-DRIVEN SCHEDULING – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- 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	15
B	4 ms	3
C	7 ms	4
D	13 ms	8



Runtime of the processes

Process	A	B	C	D
Runtime	32	4	11	24

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

Waiting time of the processes

Process	A	B	C	D
Waiting time	24	0	4	11

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

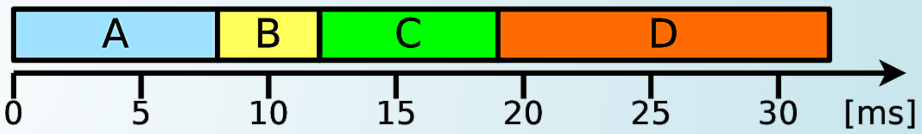
FIRST COME FIRST SERVED (FCFS)

- 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 eventually executed
- The average waiting time may be very high under certain circumstances
  - The execution of short-lived processes may have to wait for a long time if processes with long execution times have arrived before
- FCFS/FIFO can be used for $\Longrightarrow$ **batch processing**
- FIFO is used in Linux for non-preemptive **real-time** processes

FIRST COME FIRST SERVED – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- Execution order of the processes as Gantt chart

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



Runtime of the processes

Process	A	B	C	D
Runtime	8	11	16	27

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

Waiting time of the processes

Process	A	B	C	D
Waiting time	0	7	9	14

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

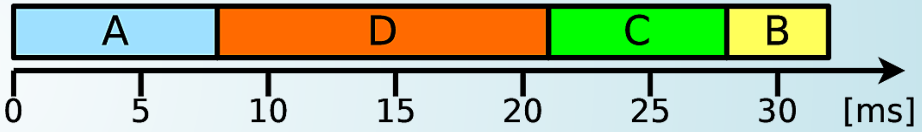
LAST COME FIRST SERVED (LCFS)

- Works according to the principle Last In First Out (LIFO)
- Processes are executed in the reverse order of creation
  - The concept is equal with a stack
- Running processes are not interrupted
  - The processes have the CPU assigned until process termination or voluntary resigning
- LCFS is not fair
  - In case of continuous creation of new processes, the old processes are not taken into account and thus may **starve**
- LCFS can be used for $\Longrightarrow$ **batch processing**
  - Is seldom used in pure form

LAST COME FIRST SERVED – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- Execution order of the processes as Gantt chart

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



Runtime of the processes

Process	A	B	C	D
Runtime	8	31	25	16

$$\frac{8+31+25+16}{4} = 20 \text{ ms}$$

Waiting time of the processes

Process	A	B	C	D
Waiting time	0	27	18	3

$$\frac{0+27+18+3}{4} = 12 \text{ ms}$$

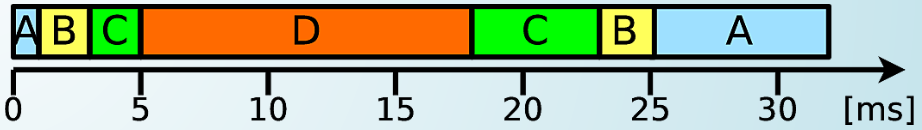
LAST COME FIRST SERVED – PREEMPTIVE VARIANT (LCFS-PR)

- A new process in state `ready` replaces the currently executed processes from the CPU
  - Preempted processes are enqueued at the end
  - If no new processes are created, the running process has the CPU assigned until process termination or voluntary resigning
- Prefers processes with a short execution time
  - The execution of a process with a short execution time may be completed before new process are created
  - Processes with a long execution time may get the CPU resigned several times and thus significantly delayed
- LCFS-PR is not fair
  - Processes with a long execution time may never get the CPU assigned and **starve**
- Is seldom used in pure form

LAST COME FIRST SERVED EXAMPLE – PREEMPTIVE VARIANT

- Four processes shall be processed on a system with a single CPU
- Execution order of the processes as Gantt chart

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



Runtime of the processes

Process	A	B	C	D
Runtime	32	24	20	13

$$\frac{32+24+20+13}{4} = 22.25 \text{ ms}$$

Waiting time of the processes

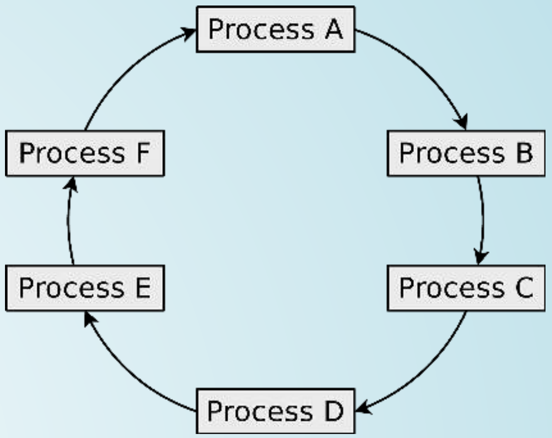
Process	A	B	C	D
Waiting time	24	20	13	0

$$\frac{24+20+13+0}{4} = 14.25 \text{ ms}$$

Which scheduling strategy may be well suited for generic user space applications?

ROUND ROBIN – RR (1/2)

- Time slices with a fixed duration (may be ∞!) are specified
- The processes are queued in a cyclic queue according to the **FIFO** principle
  - The first process of 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 resigned ($\Rightarrow$ is **preempted**) and is enqueued 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 fair among the processes
- RR with time slice size ∞ behaves like $\longrightarrow$ **FCFS**



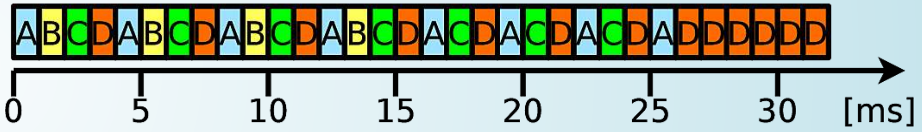
ROUND ROBIN – RR (2/2)

- The longer the execution time of a process is, the more rounds are required for its complete execution
- The duration 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 gets the simultaneousness lost
    $\Longrightarrow$ The system hangs/becomes jerky
- The usual duration of time slices is in single or double-digit millisecond range
- Prefers processes with short execution time
- **Preemptive scheduling policy**
- Round Robin scheduling can be used for interactive systems
- Round Robin is used in Linux for preemptive **real-time** processes

ROUND ROBIN – EXAMPLE

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

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



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

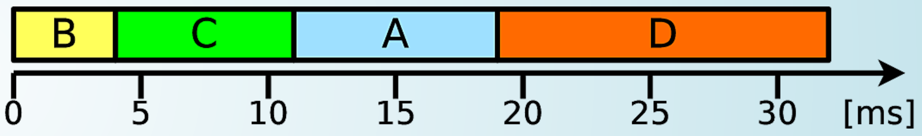
SHORTEST JOB FIRST (SJF) / SHORTEST PROCESS NEXT (SPN)

- The process with the shortest execution time get the CPU assigned first
- **Non-preemptive scheduling policy**
- **Problem:** The runtime of each process needs to be known in advance
- **Solution:** Execution time is estimated by analyzing its behavior in the past
- SJF is **not fair**
  - Prefers processes, which have a short execution time
  - Processes with a long execution time may get the CPU assigned only after a very long waiting period or starve
- If the execution time of the processes can be estimated, SJF can be used for batch processing

SHORTEST JOB FIRST – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- All processes are at time point 0 in state `ready`
- Execution order of the processes as Gantt chart

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



Runtime of the processes

Process	A	B	C	D
Runtime	19	4	11	32

$$\frac{19+4+11+32}{4} = 16.5 \text{ ms}$$

Waiting time of the processes

Process	A	B	C	D
Waiting time	11	0	4	19

$$\frac{11+0+4+19}{4} = 8.5 \text{ ms}$$

SHORTEST REMAINING TIME FIRST (SRTF)

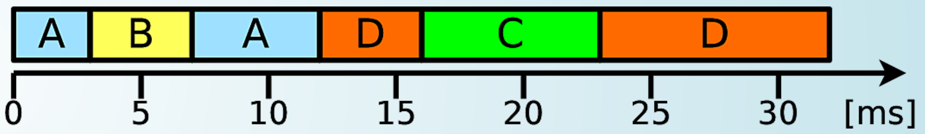
FRANKFURT UNIVERSITY OF APPLIED SCIENCES

- **Preemptive** SJF is called Shortest Remaining Time First (SRTF)
- On process creation the remaining execution time of the running process is compared with each process in state `ready` in the queue
  - If the currently running process has the shortest remaining execution time, the CPU remains assigned to this process
  - If one or more processes in state `ready` have a shorter remaining execution time, the process with the shortest remaining execution time gets the CPU assigned
- Estimation of runtime is required
- As long as no new process is created, no running process gets interrupted
  - The processes in state `ready` are compared with the running process only when a new process is created!
- Processes with a long execution time may **starve** ($\Longrightarrow$ not fair)

SHORTEST REMAINING TIME FIRST – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- Execution order of the processes as Gantt chart

Process	CPU time	Creation time
A	8 ms	0 ms
B	4 ms	3 ms
C	7 ms	16 ms
D	13 ms	11 ms



Runtime of the processes

Process	A	B	C	D
Runtime	12	4	7	21

$$\frac{12+4+7+21}{4} = 11 \text{ ms}$$

Waiting time of the processes

Process	A	B	C	D
Waiting time	4	0	0	8

$$\frac{4+0+0+8}{4} = 3 \text{ ms}$$

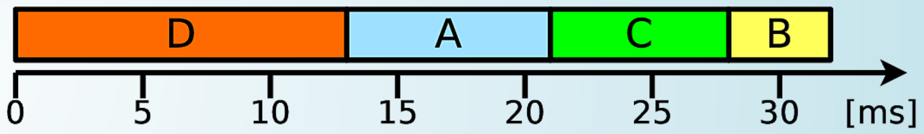
LONGEST JOB FIRST (LJF)

- The process with the longest execution time get the CPU assigned first
- **Non-preemptive scheduling policy**
- Estimation of runtime is required
- LJF is not fair
  - Prefers processes, which have a long execution time
  - Processes with a short execution time may get the CPU assigned only after a very long waiting period or **starve**
- If the execution time of the processes can be estimated, LJF can be used for batch processing

LONGEST JOB FIRST – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- All processes are at time point 0 in state `ready`
- Execution order of the processes as Gantt chart

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



Runtime of the processes

Process	A	B	C	D
Runtime	21	32	28	13

$$\frac{21+32+28+13}{4} = 23.5 \text{ ms}$$

Waiting time of the processes

Process	A	B	C	D
Waiting time	13	28	21	0

$$\frac{13+28+21+0}{4} = 15.5 \text{ ms}$$

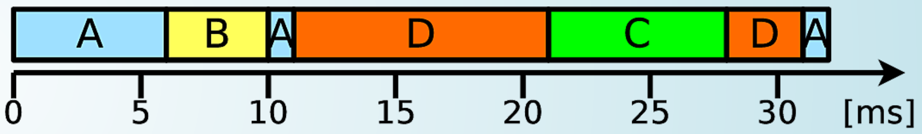
LONGEST REMAINING TIME FIRST (LRTF)

- **Preemptive** LJF is called **Longest Remaining Time First** (LRTF)
- If a new process is created, the remaining execution time of the running process is compared with each process in state `ready` in the queue
  - If the currently running process has the longest remaining execution time, the CPU remains assigned to this process
  - If one or more processes in state `ready` have a longer remaining execution time, the process with the longest remaining execution time gets the CPU assigned
- Estimation of runtime is required
- As long as no new process is created, no running process gets interrupted
  - The processes in state `ready` are compared with the running process only when a new process is created!
- Processes with a short duration may starve ($\Longrightarrow$ not fair)

LONGEST REMAINING TIME FIRST – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- Execution order of the processes as Gantt chart

Process	CPU time	Creation time
A	8 ms	0 ms
B	4 ms	6 ms
C	7 ms	21 ms
D	13 ms	11 ms



Runtime of the processes

Process	A	B	C	D
Runtime	32	4	7	20

$$\frac{32+4+7+20}{4} = 15.75 \text{ ms}$$

Waiting time of the processes

Process	A	B	C	D
Waiting time	24	0	0	7

$$\frac{24+0+0+7}{4} = 7.75 \text{ ms}$$

HIGHEST RESPONSE RATIO NEXT (HRRN)

- Fair variant of SJF/SRTF/LJF/LRTF
  - Takes the age of the process into account in order to avoid starvation
- The response ratio is calculated for each process

$$\text{Response ratio} = \frac{\text{Estimated execution time} + \text{Waiting time}}{\text{Estimated execution time}}$$

- Response ratio value of a process after creation: 1.0
  - The value rises fast for short processes
  - **Objective:** Response ratio should be as small as possible for each process
- After process termination or if a process becomes blocked, the CPU is assigned to the process with the highest response ratio
- Just as with SJF/SRTF/LJF/LRTF, the execution times of the processes must be estimated via by statistical recordings
- It is impossible that processes starve $\Longrightarrow$ HRRN is fair

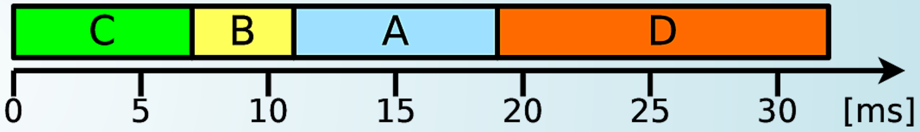
EARLIEST DEADLINE FIRST (EDF)

- Used in **real-time operating operating systems (RTOS)**
- **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 RTOS
  - Non-preemptive EDF can be used for batch processing
- EDF is used in Linux for preemptive real-time processes

EARLIEST DEADLINE FIRST – EXAMPLE

- Four processes shall be processed on a system with a single CPU
- All processes are at time point 0 in state `ready`
- Execution order of the processes as Gantt chart

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



Runtime of the processes

Process	A	B	C	D
Runtime	19	11	7	32

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

Waiting time of the processes

Process	A	B	C	D
Waiting time	11	7	0	19

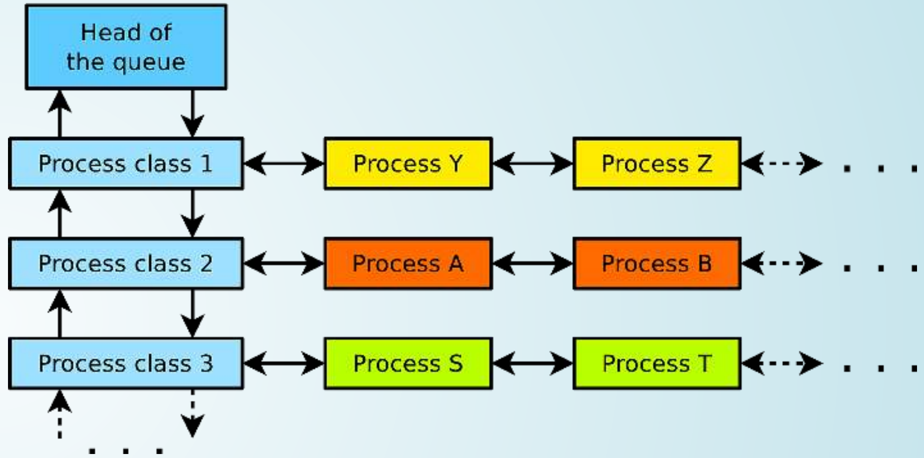
Avg. waiting time = $\frac{11+7+0+19}{4} = 9.25 \text{ ms}$

MULTILEVEL SCHEDULING

- Each scheduling policy require compromises wrt scheduling criteria
    - Procedure in practice: Several scheduling strategies are combined
      $\Longrightarrow$ **Static or dynamic multilevel scheduling**

STATIC MULTILEVEL SCHEDULING

- The list of processes of `ready` state is split into multiple sublists
  - For each sublist, a different scheduling policy may be used
- The sublists have different **priorities** or **time multiplexes** (e.g., 80%:20% or 60%:30%:10%)
  - Makes it possible to separate time-critical from non-time-critical processes



- Example of allocating the processes to different process classes (sublists) with different scheduling strategies:

Priority	Process class	Scheduling policy
1	Real-time processes (time-critical)	Priority-driven scheduling
2	Interactive processes	Round Robin
3	Compute-intensive batch processes	First Come First Served

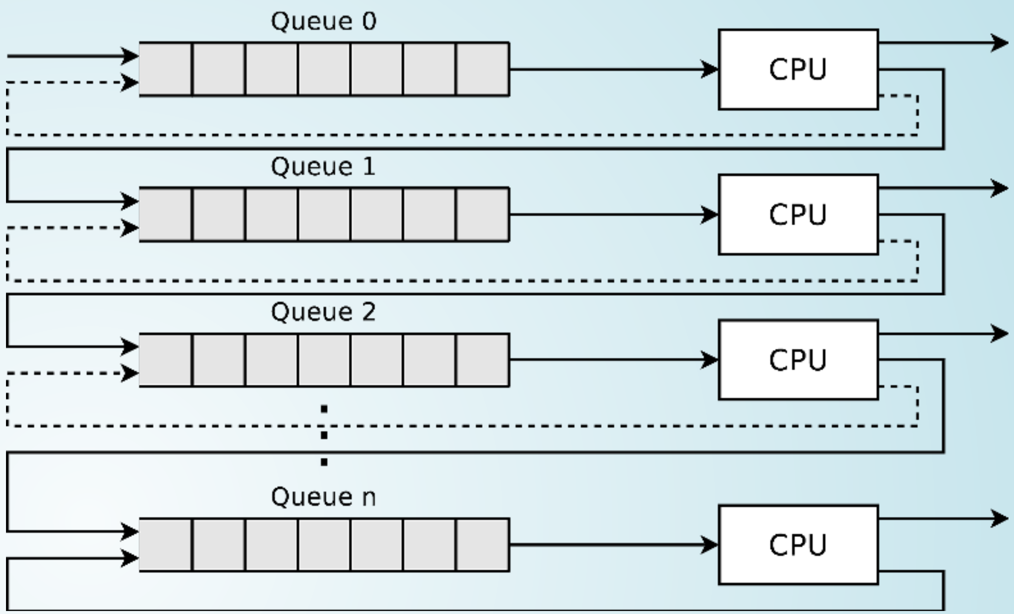
MULTILEVEL FEEDBACK SCHEDULING (1/2)

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

- 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 on voluntary basis, it is added to the same queue again
  - If a process utilized its entire time slice, it is inserted in the next lower queue, with has a lower priority
    - The priorities are therefore **dynamically** assigned with this policy
- 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



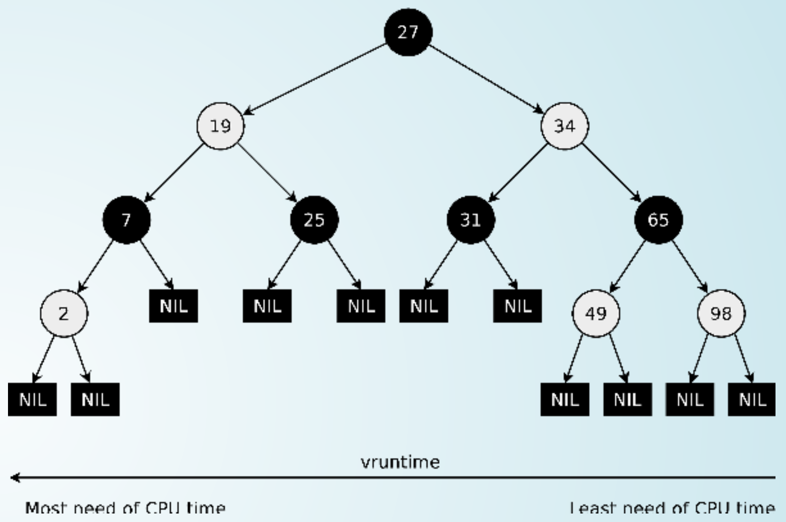
Source: William Stallings. Operating Systems. 4th edition. Prentice Hall (2001). P.413

- Processes with many I/O operations are preferred because they typically yield when waiting for I/O
- Older, longer-running processes are delayed

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

COMPLETELY FAIR SCHEDULER (LINUX SINCE 2.6.23)

- If a process gets replaced from the CPU core, the **vruntime** value is increased by the time the process did run on the CPU core
- 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

CLASSIC AND MODERN SCHEDULING METHODS

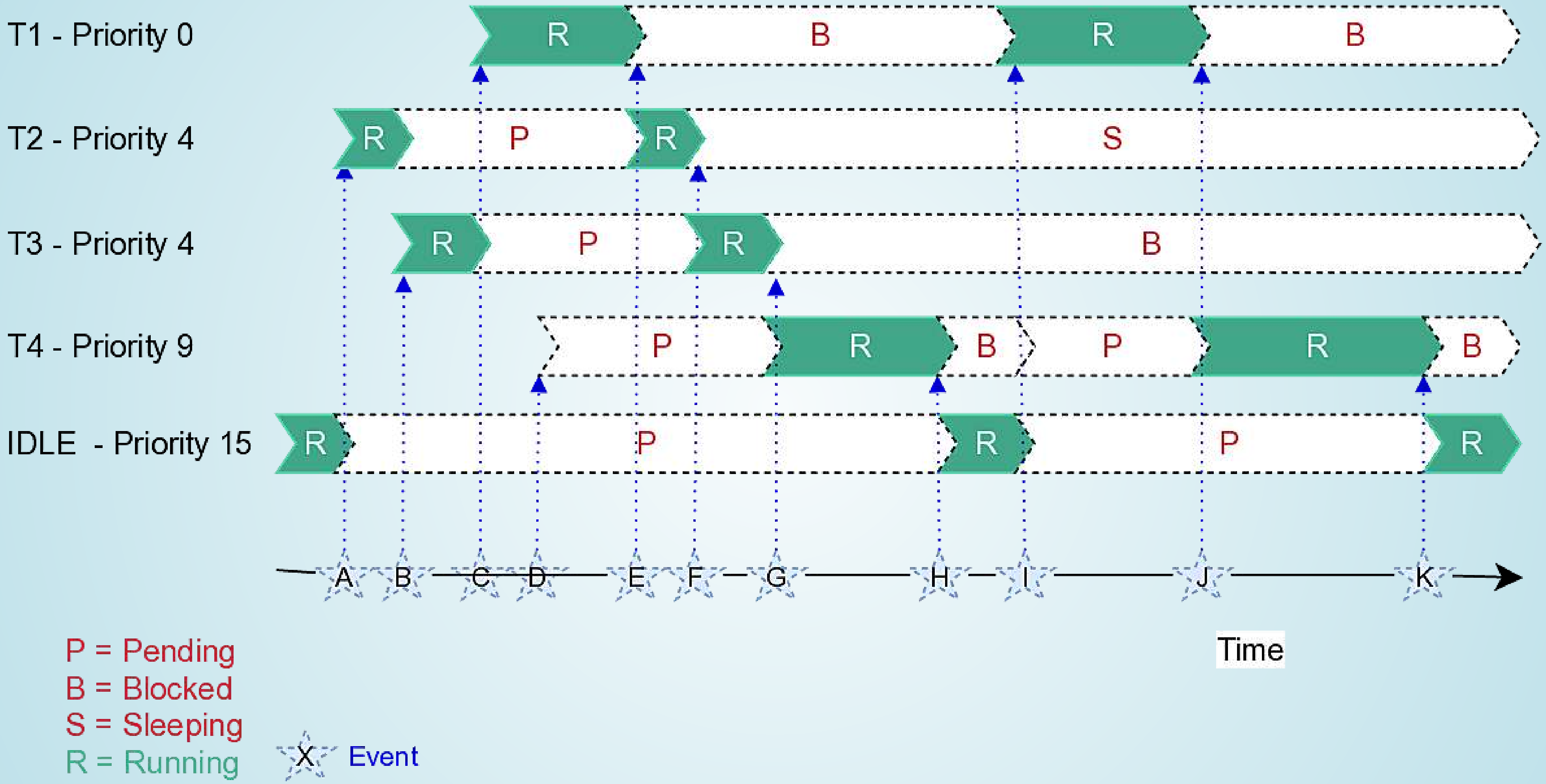
	Scheduling		Fair	CPU time	Takes priorities
	NP	P		must be known	into account
Priority-driven scheduling	X	X	no	no	yes
First Come First Served = FIFO	X		yes	no	no
Last Come First Served	X	X	no	no	no
Round Robin		X	yes	no	no
Shortest/Longest Job First	X		no	yes	no
Shortest Remaining Time First		X	no	yes	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
Static multilevel scheduling		X	no	no	yes (static)
Multilevel feedback scheduling		X	yes	no	yes (dynamic)
Completely Fair Scheduler		X	yes	no	yes

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

LINUX SCHEDULING POLICIES

- In Linux e.g., each process is assigned to a specific scheduling policy
- For **real-time** processes…
  - `SCHED_FIFO` (priority-driven scheduling, non-preemptive)
  - `SCHED_RR` (preemptive)
  - `SCHED_DEADLINE` (EDF scheduling, preemptive)
- For **non real-time** processes…
  - `SCHED_OTHER` (default Linux time-sharing scheduling) implemented as…
    - Multilevel Feedback Scheduling (until Kernel 2.4)
    - O(1) scheduler (Kernel 2.6.0 until 2.6.22)
    - Completely Fair Scheduler (since Kernel 2.6.23)

THE RIOT SCHEDULER – EXAMPLE



SUMMARY

You should now be able to answer the following questions:

- What steps does the dispatcher need to carry out for switching between processes?
- What is scheduling?
- How do preemptive scheduling and non-preemptive scheduling work?
- Explain the functioning of several common scheduling methods?
- How does scheduling in modern operating systems works in detail?