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. . .
  - the difference is between **interrupts** and **exceptions**
  - what steps the **dispatcher** carries out for switching between processes
  - what **scheduling** is
    - how **preemptive scheduling** and **non-preemptive scheduling** works
    - the functioning of several common **scheduling methods**
    - why not just a single scheduling method is used by **modern operating systems**
    - how **scheduling in modern operating systems** works in detail

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

Interrupts and Exceptions

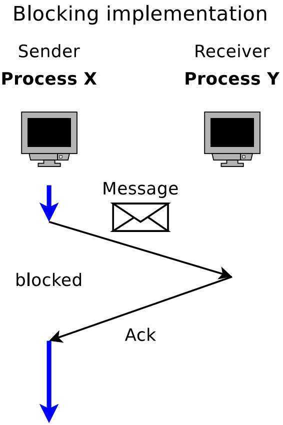
- Often unpredictable events occur to which a computer system must react
- Interrupts are events whose treatment is not allowed to become postponed
- Frequent interrupts:
  - **Error situation** (error caused by an arithmetic operation)
    - Division by zero, floating point error, address errors, . . .
  - **Software interrupt** or **exception** (is triggered by a process)
    - Examples are the exception 0x80 (see slide set 7) to switch from user mode to kernel mode and the single-stepping mode during program test (debugging, trace)
  - **Hardware interrupt**
    - Input/Output devices provide feedback to a process

Interrupt Example

- X and Y are processes, which communicate via a network
  - Both processes are executed on different computers
  - If a process does not reply within a specified time period (timeout) to a message, the message must be sent again
    - Reason: The sender assumes that the message got lost
- 2 ways exist to implement the described procedure:
  - **Blocking**
  - **Non-blocking**

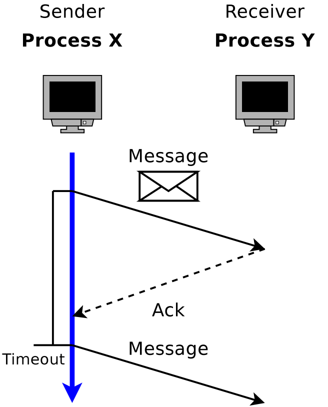
Blocking Implementation

- Process X is blocked until the message is acknowledged or the timing expires
- If the acknowledgement arrives, sender process X may continue
  - Otherwise, process X must send the message again
- Disadvantage: Long idle times for process X arise



Non-blocking Implementation

- After process X sent the message, it continues to operate normally
  - If a timeout expires because of a missing acknowledgment, the operating system suspends the process
- The context (see slide set 7) of the process is backed up and a procedure for interrupt handling is called
  - In the example, the procedure would send the message again
  - If the execution of the procedure has finished, the process becomes reactivated



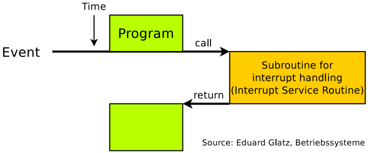
Non-blocking implementations can be realized via **interrupts** and **exceptions**

Subprograms without return value are called **procedures**. Subprograms with return value are called **functions** or **methods**

Interrupt Types – Interrupts (1/2)

- **External interrupts** are simply called interrupts
  - They are triggered by events from outside the process to be interrupted (for example, an Input/Output device reports an I/O event)
- The interrupt concept is implemented by the hardware
  - Software interrupts (exceptions) exist too
    - These are handled like hardware interrupts, but triggered by software

An interrupt indicates an event and the operating system provides an „*event handler*“, the so called **interrupt service routine**



Source: Eduard Glatz, Betriebssysteme

- The CPU is interrupted and for interrupt handling, the **interrupt service routine** of the kernel is called
  - The value of the program counter register is set to the address of the interrupt service routine, which is executed next
  - The operating system backs up the process context (see slide set 7) and restores the process context after the execution of the interrupt service routine has finished

Interrupt Types – Interrupts (2/2)

- The operating system maintains a list of addresses of all interrupt service routines
  - This list is called **interrupt vector**
- Interrupts are necessary to...
  - react quickly to signals from Input/Output devices (e.g. mouse, keyboard, HDD, network interface controller,...)
  - be able to react to time-critical events
- **Without interrupts, preemptive multitasking is impossible**
  - Preemptive multitasking means: The operating system can remove the CPU from a process before its execution is complete

Interrupt Types – Exceptions

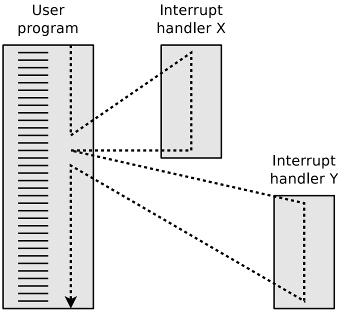
- Exceptions are **internal interrupts**
  - Are triggered by the process itself
  - Can be triggered anywhere in the program code
- Even with exceptions, the CPU is interrupted and an interrupt service routine in the kernel is activated
- Exceptions are used in software development in order to make the programs more robust against. . .
  - faulty input
  - programming errors (division by 0, addition causes overflow)
  - device errors (device not reachable, no free capacity on storage device)
- Further benefit of exceptions:
  - Separation between algorithm and error handling

Interrupt Conflicts

- 2 potential issues during interrupt handling:
  - During interrupt handling, further interrupts occur
  - Multiple interrupts occur at the same time
- 2 possible solutions:
  - **Sequential interrupt processing**
  - **Nested interrupt processing**

Sequential Interrupt Processing

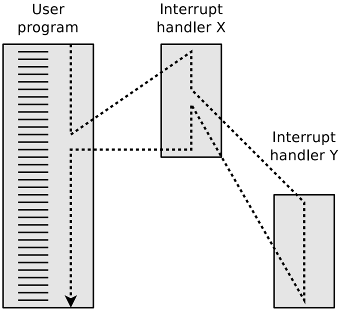
- **Sequential interrupt processing**
- Interrupts are strictly processed one after the other
- Interrupts are never interrupted
- Drawback: Priorities and time-critical reactions are ignored



Source: William Stallings. Betriebssysteme. Pearson Studium. 2003

Nested Interrupt Processing

- **Nested interrupt processing**
- Priorities are specified for the interrupts
- Interrupt handers can be interrupted, when an interrupt with higher priority occurs
- Interrupts with lower priority are delayed until all interrupts with a higher priority are handled
- Drawback: Interrupts with a low priority may be significantly delayed



Source: William Stallings. Betriebssysteme. Pearson Studium. 2003

The real-time operating systems QNX Neutrino and Windows CE 5.0 both support nested interrupts

http://www.qnx.com/developers/docs/660/topic/com.qnx.doc.neutrino.sys_arch/topic/kernel_Nested_interrupts.html
http://msdn.microsoft.com/de-de/library/ms892539.aspx

Process Switching – The Dispatcher (1/2)

- Tasks of multitasking operating systems are among others:
  - **Dispatching**: Assign the CPU to another process with a process switch
  - **Scheduling**: Determine the point in time, when the process switching occurs and 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. . .

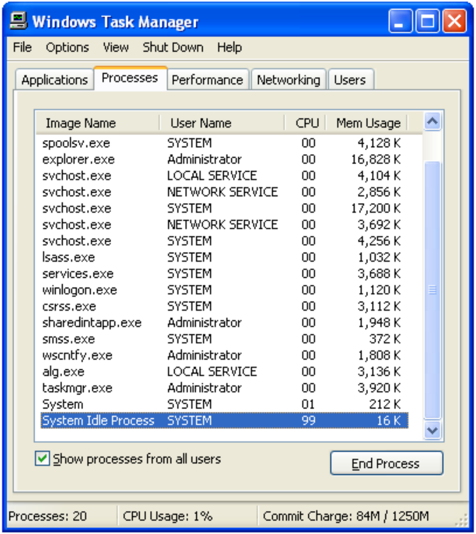
- back up the context (register contents) of the executed process in the process control block
- assign the CPU to another process
- import the context (register contents) of the process, which will be executed next, from the process control block

Process Switching – The Dispatcher (2/2)

Image Source: Wikipedia

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



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 optimal 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 control over the CPU until its execution is finished or it 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 and MacOS 8/9

  - **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, outweighs the drawbacks

Examples: Linux, MacOS X, Windows 95 and more recent versions

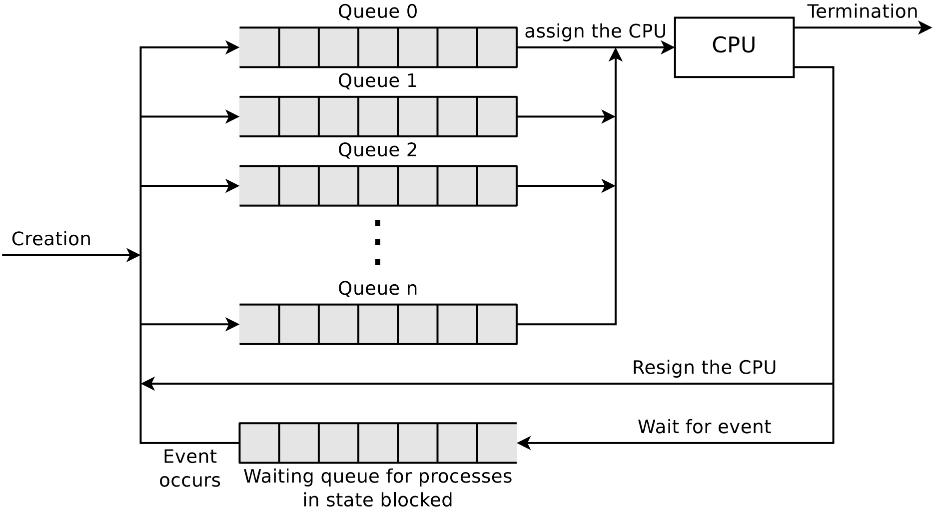
Scheduling Methods

- 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) respectively **First In First Out** (FIFO)
  - **Last Come First Served** (LCFS)
  - **Round Robin** (RR) with time quantum
  - **Shortest Job First** (SJF) and **Longest Job First** (LJF)
  - **Shortest Remaining Time First** (SRTF)
  - **Longest Remaining Time First** (LRTF)
  - **Highest Response Ratio Next** (HRRN)
  - **Earliest Deadline First** (EDF)
  - **Fair-share scheduling**
  - **Static multilevel scheduling**
  - **Multilevel feedback scheduling**

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 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 44)
- 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



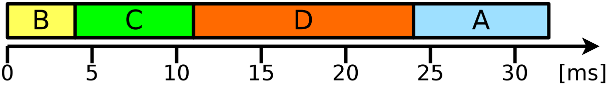
Source: William Stallings. Betriebssysteme. Pearson Studium. 2003

Example of Priority-driven Scheduling

- 4 processes shall be processed on a single CPU system
- All processes are at time point 0 in state `ready`

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

- Execution order of the processes as Gantt chart (timeline)



- Runtime of the processes

Process	A	B	C	D
Runtime	32	4	11	24

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

- Waiting time of the processes

Process	A	B	C	D
Waiting time	24	0	4	11

$$\frac{24+0+4+11}{4} = 9.75 \text{ ms}$$

First Come First Served (FCFS)

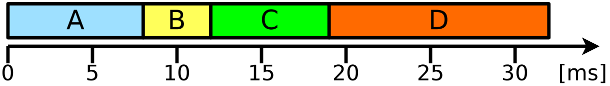
- Works according to the principle **First In First Out** (FIFO)
- Processes get the CPU assigned according to their arrival order
- This scheduling method is similar to a waiting line of customers in a store
- Running processes are not interrupted
  - It is **non-preemptive scheduling**
- FCFS is **fair**
  - 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 a long processing time have arrived before
- FCFS/FIFO can be used for batch processing ($\Longrightarrow$ slide set 1)

First Come First Served – Example

- 4 processes shall be processed on a single CPU system

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

- Execution order of the processes as Gantt chart (timeline)



- Runtime of the processes

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

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

- Waiting time of the processes

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

$$\frac{0+7+9+14}{4} = 7.5 \text{ ms}$$

Last Come First Served (LCFS)

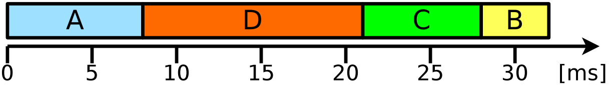
- Works according to the principle **Last In First Out** (LIFO)
- Processes are executed in the reverse order of creation
  - The concept is equal with a stack
- Running processes are **not interrupted**
  - The processes have the CPU assigned until process termination or voluntary resigning
- LCFS is **not fair**
  - The case of continuous creation of new processes, the old processes are not taken into account and thus may **starve**
- LCFS/LIFO can be used for batch processing ($\Longrightarrow$ slide set 1)
  - Is seldom used in pure form

Last Come First Served – Example

- 4 processes shall be processed on a single CPU system

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

- Execution order of the processes as Gantt chart (timeline)



- 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)

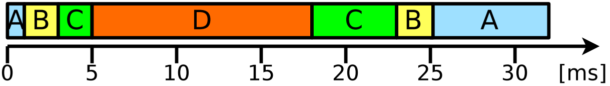
- A new process in state `ready` replaces the currently executed processes from the CPU
  - Processes, which get the CPU resigned, are inserted at the end of the queue
  - If no new processes are created, the running process has the CPU assigned until process termination or voluntary resigning
- **Prefers 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

- 4 processes shall be processed on a single CPU system

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

- Execution order of the processes as Gantt chart (timeline)



- Runtime of the processes

Process	A	B	C	D
Runtime	32	24	20	13

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

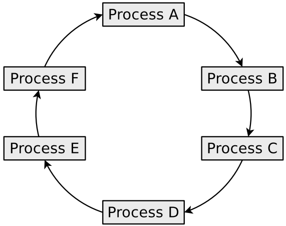
- Waiting time of the processes

Process	A	B	C	D
Waiting time	24	20	13	0

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

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 of the queue get the CPU assigned for the duration of a time slice
  - After the expiration of the time slice, the process gets the CPU resigned and it is inserted 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** between the processes
- RR with time slice size ∞ behaves like FCFS



Round Robin – RR (2/2)

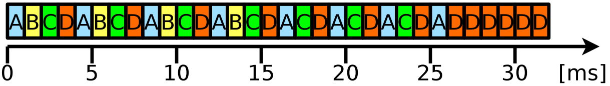
- The longer the execution time of a process is, the more rounds are required for its complete execution
- The size of the time slices influences the performance of the system
  - The shorter they are, the more process change must take place
    $\Longrightarrow$ Increased overhead
  - The longer they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

Round Robin – Example

- 4 processes shall be processed on a single CPU system
- All processes are at time point 0 in state `ready`
- Time quantum $q = 1$ ms

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

- Execution order of the processes as Gantt chart (timeline)



- Runtime of the processes

Process	A	B	C	D
Runtime	26	14	24	32

$$\frac{26+14+24+32}{4} = 24 \text{ ms}$$

- Waiting time of the processes

Process	A	B	C	D
Waiting time	18	10	17	19

$$\frac{18+10+17+19}{4} = 16 \text{ ms}$$

Shortest Job First (SJF) / Shortest Process Next (SPN)

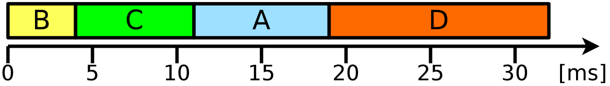
- The process with the shortest execution time get the CPU assigned first
- **Non-preemptive scheduling method**
- Main problem:
  - For each process, it is necessary to know how long it takes until its termination, which means, how long is its execution time
    - In practice this is almost never the case ($\Longrightarrow$ **unrealistic**)
- Solution:
  - The execution time of processes is estimated by recording and analyzing the execution time of prior processes
- 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 **starves**
- If the execution time of the processes can be estimated, SJF can be used for batch processing ($\Longrightarrow$ slide set 1)

Shortest Job First – Example

- 4 processes shall be processed on a single CPU system
- All processes are at time point 0 in state `ready`

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

- Execution order of the processes as Gantt chart (timeline)



- 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)

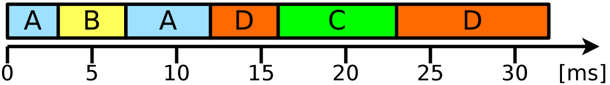
- **Preemptive** SJF is called **Shortest Remaining Time First** (SRTF)
- If a new process is created, the remaining execution time of the running process is compared with each process in state `ready` in the queue
  - If the currently running process has the shortest remaining execution time, the CPU 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
- Main problem: The remaining execution time must be known ($\Longrightarrow$ **unrealistic**)
- 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

- 4 processes shall be processed on a single CPU system

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

- Execution order of the processes as Gantt chart (timeline)



- 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)

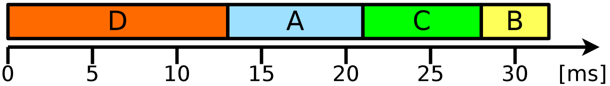
- The process with the longest execution time get the CPU assigned first
- **Non-preemptive scheduling method**
- Main problem: Just as with SJF, the execution time of each process must be known
  - In practice this is almost never the case ($\Longrightarrow$ **unrealistic**)
- LJF is **not fair**
  - **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 ($\Longrightarrow$ slide set 1)

Longest Job First – Example

- 4 processes shall be processed on a single CPU system
- All processes are at time point 0 in state `ready`

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

- Execution order of the processes as Gantt chart (timeline)



- 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)

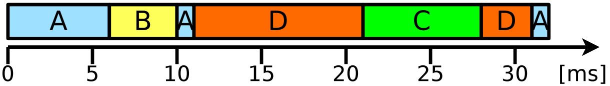
- **Preemptive** LJF is called **Longest Remaining Time First** (LRTF)
- If a new process is created, the remaining execution time of the running process is compared with each process in state `ready` in the queue
  - If the currently running process has the longest remaining execution time, the CPU 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
- Main problem: The remaining execution time must be known ($\Longrightarrow$ **unrealistic**)
- 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

- 4 processes shall be processed on a single CPU system

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

- Execution order of the processes as Gantt chart (timeline)



- 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
    - Then the scheduling operates efficiently
- After termination of a process 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)

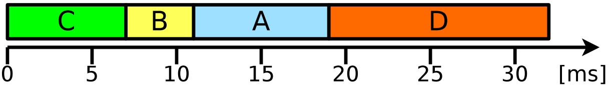
- Objective: processes should comply with their (*deadlines*) when possible
- Processes in state `ready` are **arranged according to their deadline**
  - The process with the closest deadline gets the CPU assigned
- A review and if required, a reorganization of the queue takes place when...
  - a new 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

Earliest Deadline First – Example

- 4 processes shall be processed on a single CPU system
- All processes are at time point 0 in state `ready`

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

- Execution order of the processes as Gantt chart (timeline)



- Runtime of the processes

Process	A	B	C	D
Runtime	19	11	7	32

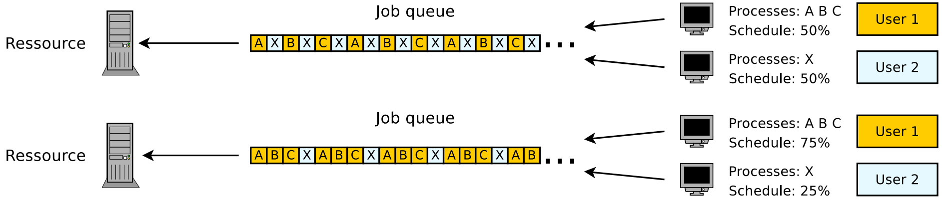
$$\frac{19+11+7+32}{4} = 17.25 \text{ ms}$$

- Waiting time of the processes

Process	A	B	C	D
Waiting time	11	7	0	19

$$\frac{11+7+0+19}{4} = 9.25 \text{ ms}$$

Fair-Share



- With **Fair-share**, resources are distributed between 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 his processes
- The users get *resource shares*

Fair share is often used in cluster and grid systems

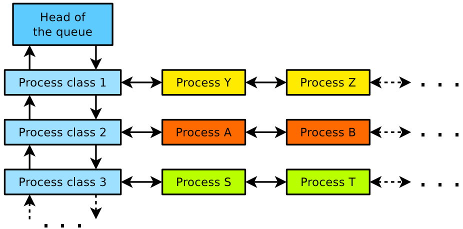
Fair share is implemented in job schedulers and meta-schedulers (e.g. Oracle Grid Engine) for assigning the jobs to resources in grid sites distributing jobs between grid sites

Multilevel Scheduling

- With each scheduling policy, compromises concerning the different scheduling criteria must be made
  - 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 method 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



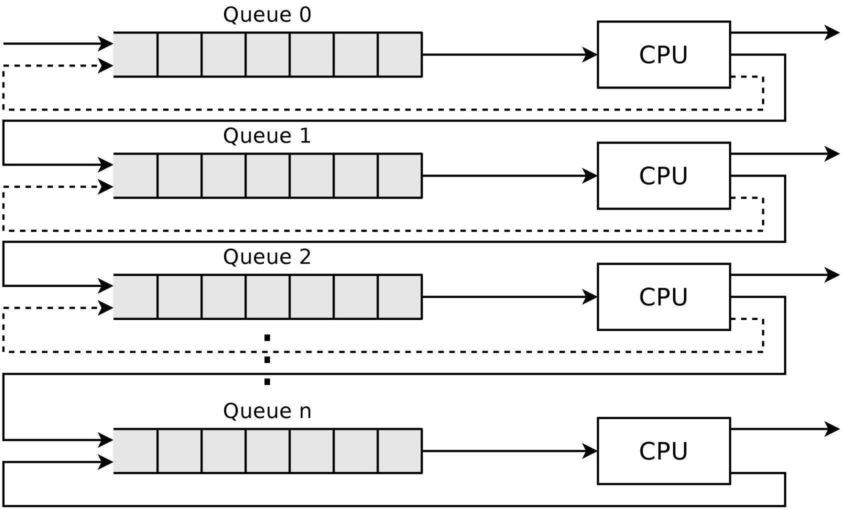
- Example of allocating the processes to different process classes (sublists) with different scheduling strategies:

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

Multilevel Feedback Scheduling (1/2)

- It is **impossible to calculate the execution time precisely in advance**
  - Solution: Processes, which utilized much execution time in the past, get **punished**
- **Multilevel feedback scheduling** works like multilevel scheduling with multiple queues
  - Each queue has a different priority or time multiplex
- Each new process is inserted in the top queue
  - This way it has the highest priority
- For each queue, Round Robin is used
  - Is a process resigns the CPU on voluntary basis, it is inserted in the same queue again
  - If a process utilized its complete time slice, it is inserted in the next lower queue, with has a lower priority
    - The priorities are therefore **dynamically** assigned with this method
- Multilevel feedback scheduling is **preemptive Scheduling**

Queues of Multilevel Feedback Scheduling



Source: William Stallings. Betriebssysteme. Pearson Studium. 2003

Multilevel Feedback Scheduling (2/2)

- Benefit:
  - **No complicated estimations** are required!
    - New processes are quickly assigned to a priority class
- **Prefers new processes** over older (longer-running) processes
- Processes with many Input and output operations are preferred because they are inserted in the original queue again when they resigns the CPU on voluntary basis
  - This way they keep their priority value
- Older, longer-running processes are delayed

Classic and modern Scheduling Methods

	Scheduling NP	Scheduling P	Fair	CPU runtime must be known	Takes priorities into account
Priority-driven scheduling	X	X	no	no	yes
First Come First Served	X		yes	no	no
Last Come First Served	X	X	no	no	no
Round Robin		X	yes	no	no
Shortest Job First	X		no	yes	no
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)

- 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

A simple Scheduling Example

Process	CPU runtime	Priority
A	5 ms	15
B	10 ms	5
C	3 ms	4
D	6 ms	12
E	8 ms	7

- 5 processes shall be processed on a single CPU system
- All processes are at time point 0 in state `ready`
- High priorities are characterized by high values

- Draw the execution order of the processes with a Gantt chart (timeline) for **Round Robin** (time quantum $q = 1\,\text{ms}$), **FCFS**, **SJF**, **LJF** and **priority-driven scheduling**
- Calculate the average runtimes and waiting times of the processes
  - Runtime = Time between creation and termination
  - Waiting time = runtime - CPU runtime

When in doubt, use FIFO

That means in detail: When the decision criterion of the scheduling method used, holds true for multiple processes, then take the oldest process $\Longrightarrow$ FIFO

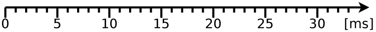
A simple Scheduling Example

Process	CPU runtime	Priority
A	5 ms	15
B	10 ms	5
C	3 ms	4
D	6 ms	12
E	8 ms	7

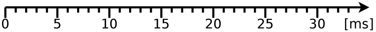
Runtime	A	B	C	D	E
RR					
FCFS					
SJF					
LJF					
PS*					

* Priority-driven scheduling

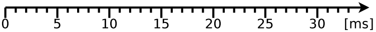
Round Robin
(time quantum = 1)



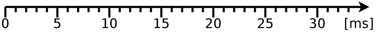
First Come
First Served



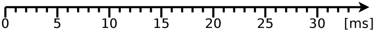
Shortest Job First



Longest Job First



Priority-driven
scheduling

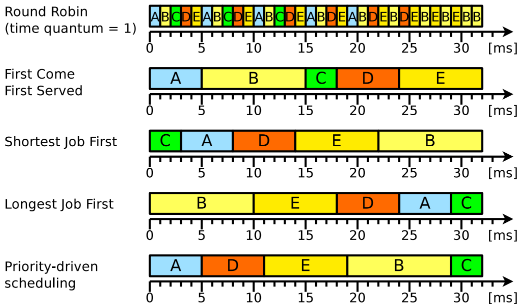


Waiting time	A	B	C	D	E
RR					
FCFS					
SJF					
LJF					
PS*					

* Priority-driven scheduling

- Waiting time = time of a process being in state `ready`

Solution – Gantt Diagram + Runtime (Turnaround Time)



Runtime	**A**	**B**	**C**	**D**	**E**
RR	20	32	13	25	30
FCFS	5	15	18	24	32
SJF	8	32	3	14	22
LJF	29	10	32	24	18
PS*	5	29	32	11	19

* Priority-driven scheduling

$$\text{RR} \quad \frac{20+32+13+25+30}{5} = 24 \text{ ms}$$

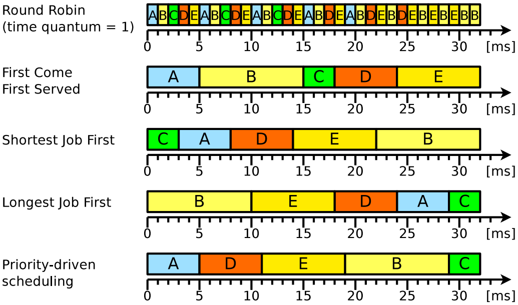
$$\text{FCFS} \quad \frac{5+15+18+24+32}{5} = 18.8 \text{ ms}$$

$$\text{SJF} \quad \frac{8+32+3+14+22}{5} = 15.8 \text{ ms}$$

$$\text{LJF} \quad \frac{29+10+32+24+18}{5} = 22.6 \text{ ms}$$

$$\text{PS} \quad \frac{5+29+32+11+19}{5} = 19.2 \text{ ms}$$

Solution – Gantt Diagram + Waiting Time



Waiting time	A	B	C	D	E
RR	15	22	10	19	22
FCFS	0	5	15	18	24
SJF	3	22	0	8	14
LJF	24	0	29	18	10
PS*	0	19	29	5	11

* Priority-driven scheduling

RR $\frac{15+22+10+19+22}{5} = 17.6$ ms

FCFS $\frac{0+5+15+18+24}{5} = 12.4$ ms

SJF $\frac{3+22+0+8+14}{5} = 9.4$ ms

LJF $\frac{24+0+29+18+10}{5} = 16.2$ ms

PS $\frac{0+19+29+5+11}{5} = 12.8$ ms

Conclusion (1/2)

- Of the investigated scheduling methods has/have...
  - **SJF** the best average runtime and best average waiting time
  - **RR** and **LJF** the worst average running times and average waiting times

Process	CPU runtime
A	24 ms
B	2 ms

- Reason:
  - If a short-running process runs before a long-running process, the runtime and wanting time of the long process process get **slightly worse**
  - If a long-running process runs before a short-running process, the runtime and wanting 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

Conclusion (2/2)

- **RR** causes frequent process switches
  - The resulting **overhead** has an additional negative effect on the system performance
- The size of the overhead depends of the length of the time slices
  - **Short time slices** $\Longrightarrow$ high overhead
  - **Long time slices** $\Longrightarrow$ response time may become too long for interactive processes

A further Scheduling Example

(Exam Question SS2009)

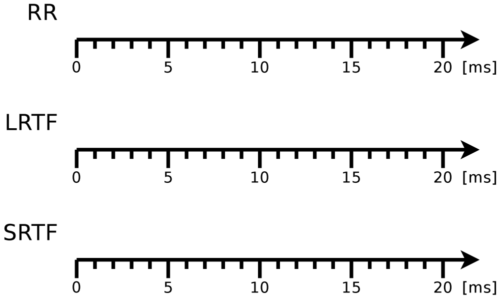
Process	CPU runtime	Creation time
A	3 ms	0 ms
B	2 ms	3 ms
C	5 ms	4 ms
D	3 ms	5 ms
E	2 ms	9 ms
F	5 ms	10 ms

- The following processes with different creation times shall be executed on a single CPU system
- Draw the execution order of the processes with a Gantt chart (timeline) for **Round Robin** (time quantum $q = 1$ ms), **FCFS**, **Longest Remaining Time First** (LRTF) und **Shortest Remaining Time First** (SRTF)
- **Attention!!!** For Round Robin, the creation time is is 0 ms for all processes. This exception is only valid for Round Robin! Please consider for the other scheduling method, the creation times that are given in the table
- Calculate the average runtimes and waiting times of the processes

Scheduling Example

(Exam Question SS2009)

Process	CPU runtime	Creation time
A	3 ms	0 ms
B	2 ms	3 ms
C	5 ms	4 ms
D	3 ms	5 ms
E	2 ms	9 ms
F	5 ms	10 ms

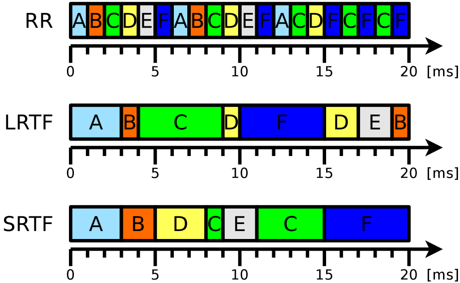


Runtime	A	B	C	D	E	F
RR						
SRTF						
LRTF						

Waiting time	A	B	C	D	E	F
RR						
SRTF						
LRTF						

Scheduling Example (Solution)

(Exam Question SS2009)



Runtime	A	B	C	D	E	F
RR	13	8	19	15	11	20
LRTF	3	17	5	12	10	5
SRTF	3	2	11	3	2	10

RR $\frac{13+8+19+15+11+20}{6} = 14,\overline{3}$ ms

LRTF $\frac{3+17+5+12+10+5}{6} = 8.\overline{6}$ ms

SRTF $\frac{3+2+11+3+2+10}{6} = 5.1\overline{6}$ ms

Waiting time	A	B	C	D	E	F
RR	10	6	14	12	9	15
LRTF	0	15	0	9	8	0
SRTF	0	0	6	0	0	5

RR $\frac{10+6+14+12+9+15}{6} = 11$ ms

LRTF $\frac{0+15+0+9+8+0}{6} = 5.\overline{3}$ ms

SRTF $\frac{0+0+6+0+0+5}{6} = 1.8\overline{3}$ ms