

# 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](mailto: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** 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

In SS2019 I erased all scheduling algorithms (SJF/SRTF/LJF/LRTF/HRRB) from my course material that require to know 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 ( $\implies$  **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**: 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...

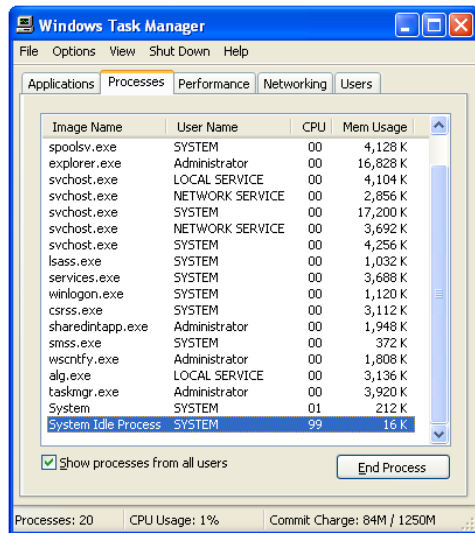
- back up the context (register contents) of the executed process in the process control block
- assign the CPU to another process
- import the context (register contents) of the process, which will be executed next, from the process control block

# 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

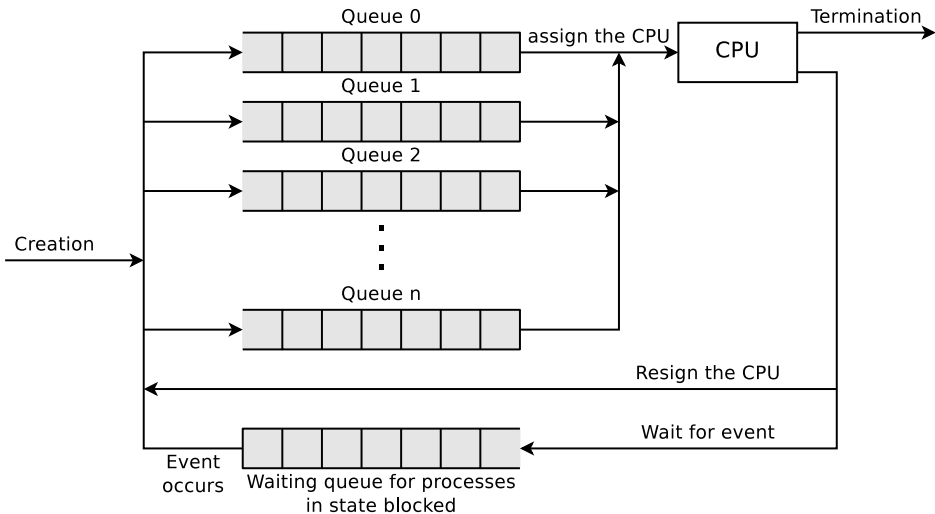
- Several scheduling methods (algorithms) exist
  - Each method tries to comply with the well-known scheduling criteria and principles in varying degrees
- Some scheduling methods:
  - **Priority-driven scheduling**
  - **First Come First Served (FCFS) = First In First Out (FIFO)**
  - ~~Last Come First 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  
⇒ **Multilevel feedback scheduling** (see slide 19)
- Risk of (static) priority-driven scheduling: Processes with low priority values may starve (⇒ **this is not fair**)
- Priority-driven scheduling can be used for interactive systems



# Priority-driven Scheduling



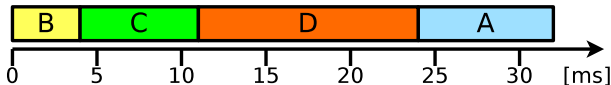
Source: William Stallings. Betriebssysteme. Pearson Studium. 2003

# 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)

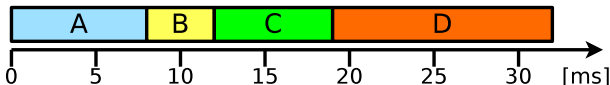
- Works according to the principle **First In First Out** (FIFO)
- Processes get the CPU assigned according to their arrival order
- This scheduling method is similar to a waiting line of customers in a store
- Running processes are not interrupted
  - It is **non-preemptive scheduling**
- FCFS is **fair**
  - 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 ( $\implies$  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

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

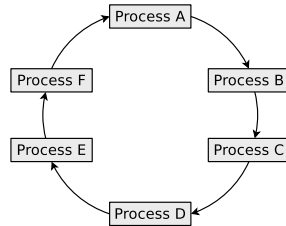
- Waiting time of the processes

Process	A	B	C	D
Waiting time	0	7	9	14

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

## 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  $\infty$  behaves like FCFS



## Round Robin – RR (2/2)

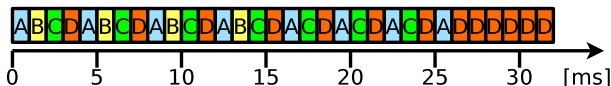
- The longer the execution time of a process is, the more rounds are required for its complete execution
- The size of the time slices influences the performance of the system
  - The shorter they are, the more process change must take place  
⇒ Increased overhead
  - The longer they are, the more gets the simultaneousness lost  
⇒ 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

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

## 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

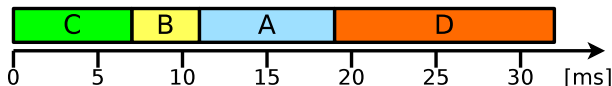


# Earliest Deadline First – Example

- 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

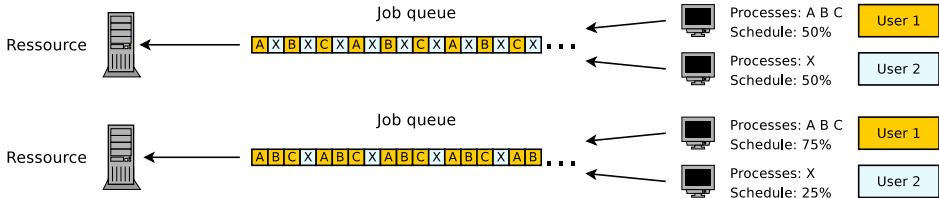
$$\frac{19+11+7+32}{4} = 17.25 \text{ ms}$$

- 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

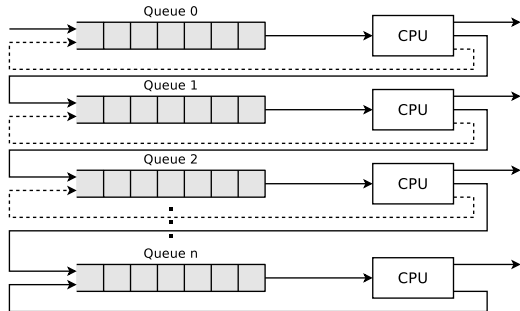
Fair share is implemented in job schedulers and meta-schedulers (e.g. Oracle Grid Engine) for assigning the jobs to resources in grid sites distributing jobs between grid sites

# Multilevel 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 with multiple queues
  - Each queue has a different priority or time multiplex (e.g. 70%:15%:10%:5%)
- Each new process is inserted in the top queue
  - This way it has the highest priority
- For each queue, Round Robin is used
  - If 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**

# Multilevel Feedback Scheduling (2/2)

- Benefit:
  - **No complicated estimations** 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  $\Rightarrow$  This way they keep their priority value
- Older, longer-running processes are delayed



Source: William Stallings, Betriebssysteme, Pearson Studium, 2003

Modern operating systems (e.g. Linux, Mac OS X and Microsoft Windows) use variants of multilevel feedback scheduling for the scheduling of the processes

# Classic and modern Scheduling Methods

	Scheduling NP	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
<del>Last Come First Served</del>	<del>X</del>	<del>X</del>	<del>no</del>	<del>no</del>	<del>no</del>
Round Robin		X	yes	no	no
<del>Shortest Job First</del>	<del>X</del>		<del>no</del>	<del>yes</del>	<del>no</del>
<del>Longest Job First</del>	<del>X</del>		<del>no</del>	<del>yes</del>	<del>no</del>
<del>Shortest Remaining Time First</del>		<del>X</del>	<del>no</del>	<del>yes</del>	<del>no</del>
<del>Longest Remaining Time First</del>		<del>X</del>	<del>no</del>	<del>yes</del>	<del>no</del>
<del>Highest Response Ratio Next</del>	<del>X</del>		<del>yes</del>	<del>yes</del>	<del>no</del>
Earliest Deadline First	X	X	yes	no	no
Fair-share		X	yes	no	no
<del>Static multilevel scheduling</del>		<del>X</del>	<del>no</del>	<del>no</del>	<del>yes (static)</del>
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