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** 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/HRRN) 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 now, 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

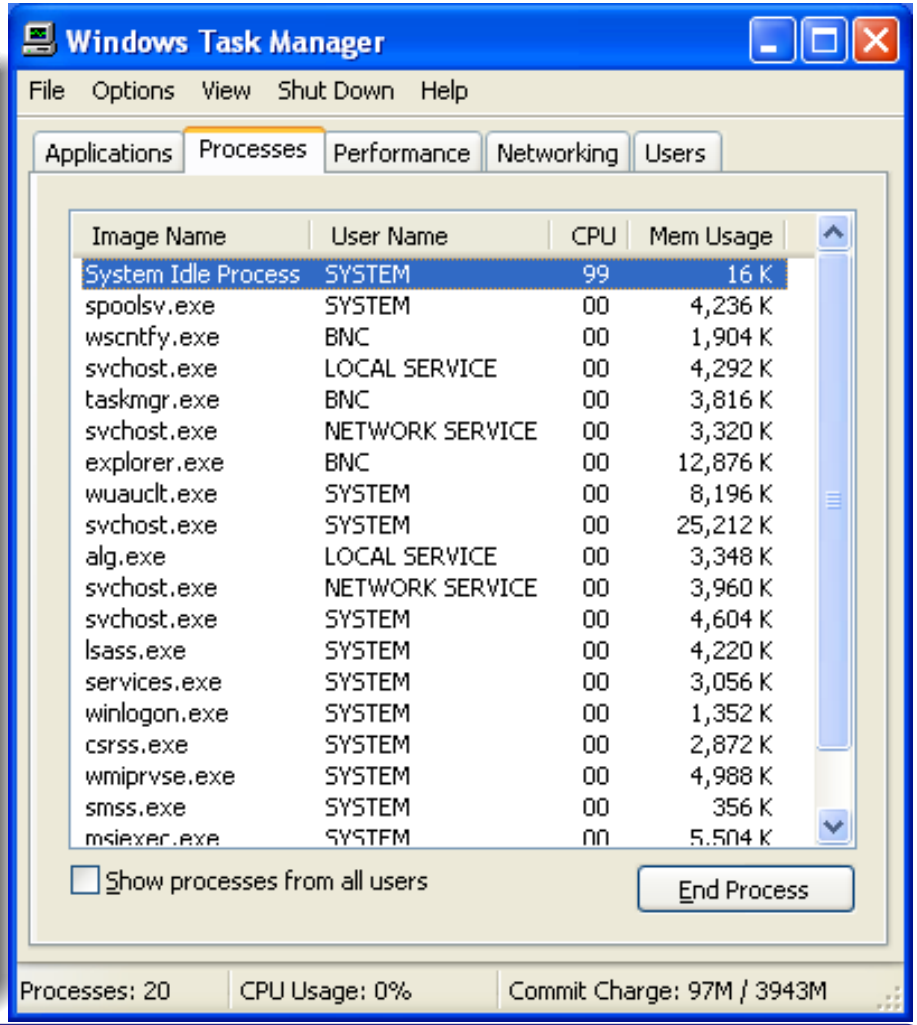
If a process switches into the state `running` or from the state `running` to another state, the dispatcher needs to...

- 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



`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 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, 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, outweighs the drawbacks

Examples: Linux, MacOS X, Windows 95/98/Me (for 32-Bit processes), Windows NT (incl. XP/Visa/7/8/10), 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 runtime
A	24 ms
B	2 ms

- 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

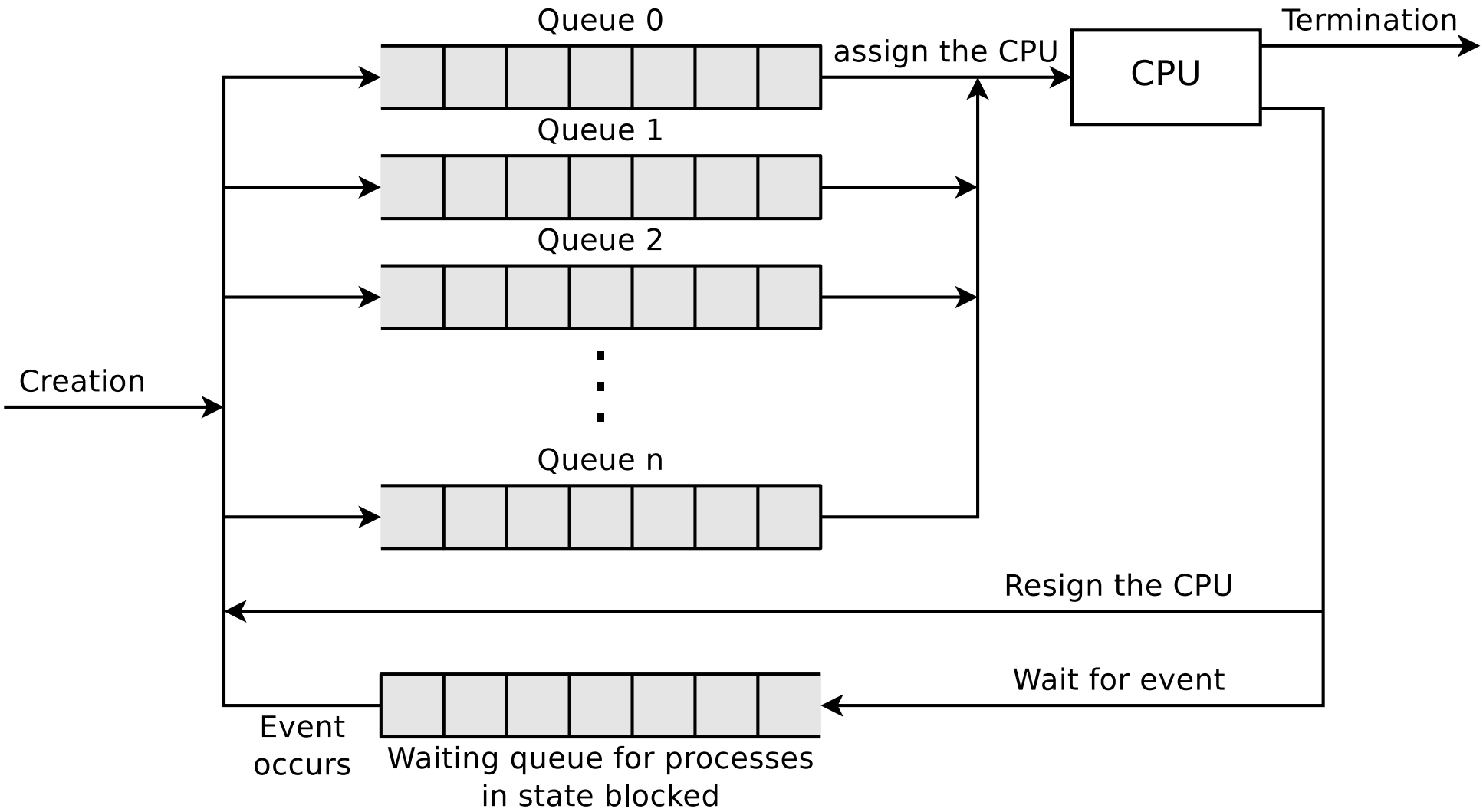
Scheduling Methods

- 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
    $\Longrightarrow$ **Multilevel feedback scheduling** (see slide 20)
- 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

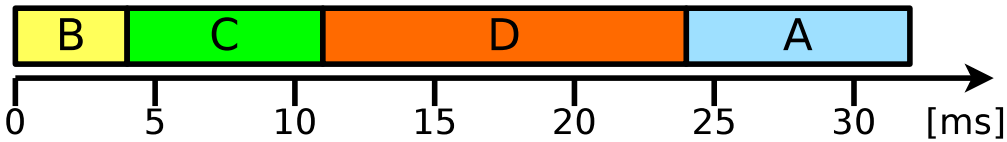


Source: William Stallings. Operating Systems. 4th edition. Prentice Hall (2001). P.401

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

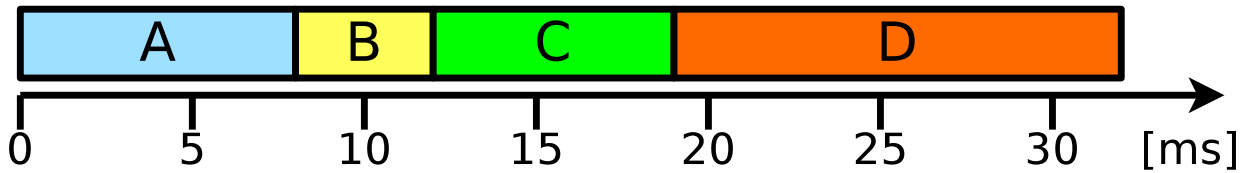
First Come First Served (FCFS)

- Works according to the principle **First In First Out** (FIFO)
- 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 long execution times 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/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

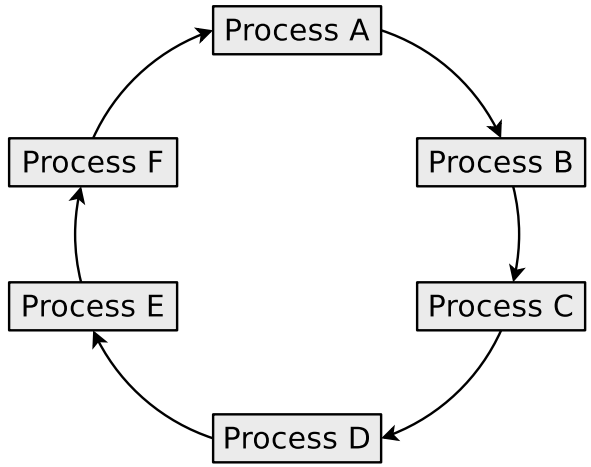
Waiting time of the processes

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 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 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 **fair** among the processes
- RR with time slice size ∞ behaves like FCFS



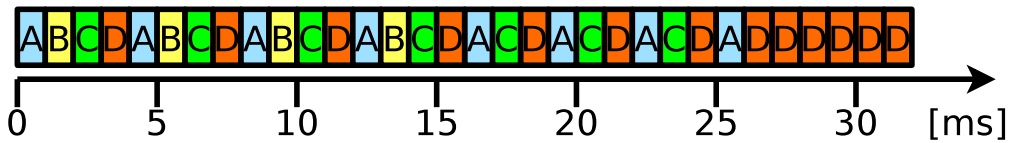
Round Robin – RR (2/2)

- 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 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 with 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/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

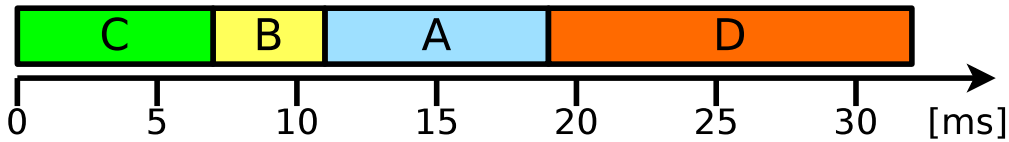
Earliest Deadline First (EDF)

- 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

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

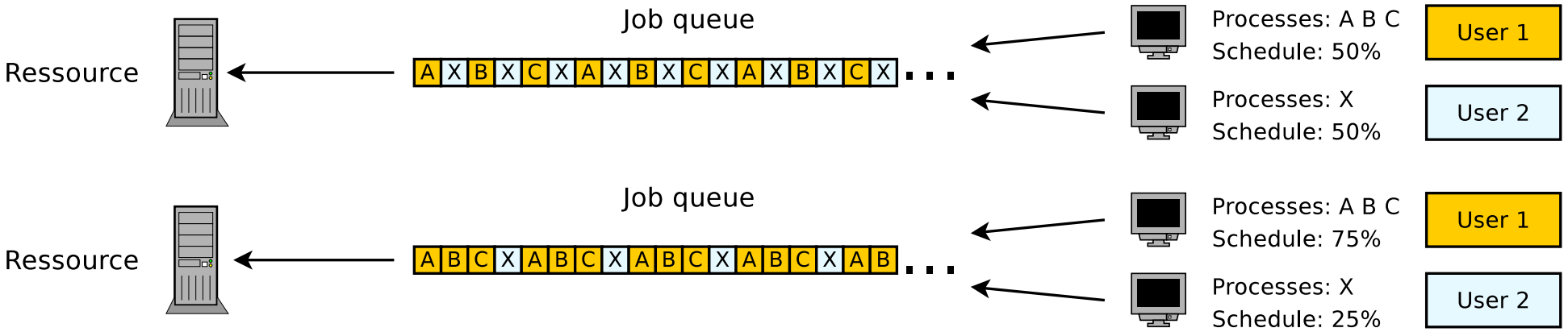
Avg. runtime = $\frac{19+11+7+32}{4} = 17.25$ ms

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 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

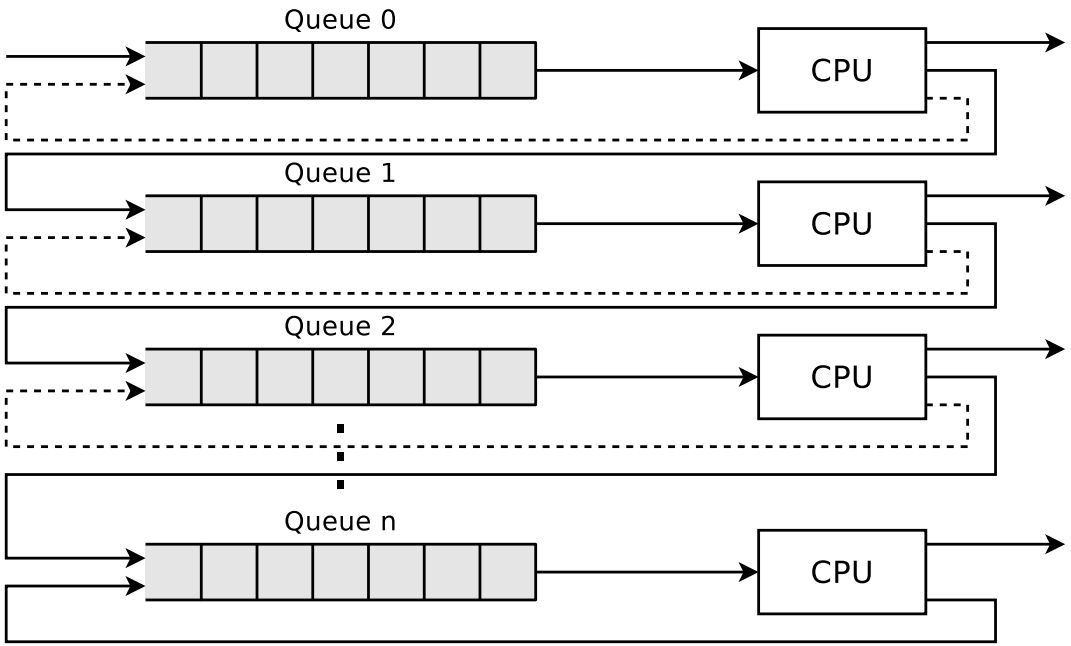
Fair share is implemented in job schedulers and meta-schedulers (e.g. Oracle Grid Engine) for assigning the jobs to resources in grid sites distributing jobs between grid sites

Multilevel Feedback Scheduling (1/2)

- It is **impossible to 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 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 in the original queue again when they resigns 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

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	Scheduling P	Fair	CPU time 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

Scheduling methods which do not play a role here for time reasons. . .

Linux 2.6.0 until 2.6.22 implements the **O(1) scheduler**. Linux since 2.6.23 implements the **Completely Fair Scheduler** (CFS).
`https://www.ibm.com/developerworks/library/l-scheduler/index.html`
`https://developer.ibm.com/tutorials/l-completely-fair-scheduler/`