

# OS04: Scheduling

Based on Chapter 3 of [\[Hai19\]](#)






([Usage hints](#) for this presentation)

Computer Structures and Operating Systems 2023  
Dr. Jens Lechtenbörger ([License Information](#))

Data Science: Machine Learning and Data Engineering (Prof. Gieseke)  
Dept. of Information Systems  
WWU Münster, Germany



## Speaker notes

- To toggle these notes, press `v`
  - If a slide contains audio, notes might show transcript
- Press `?` for key bindings (in particular, `a`, `o`, `n`, `p`, `Ctrl-Shift-f`)
- Presentations support two different PDF formats, see [usage notes](#) 
  - Both hyperlinked on index page
    - Concise PDF format (replace `.html` and whatever follows in [address bar](#)  with `.pdf`)
    - Print browser view to PDF (add `?print-pdf` after `.html`, then print to PDF; [suggested settings](#) )
- If you find the amount of outgoing links to be distracting, see [usage notes](#) 
  - Add `?hideLinks` (maybe with a number) after `.html`
- See [usage notes](#)  for other non-obvious features

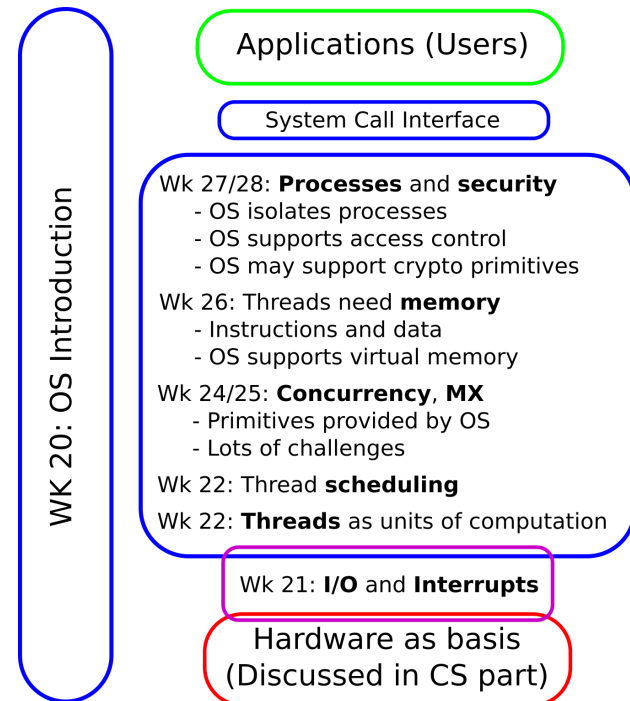


# 1. Introduction



# 1.1. OS Plan

- OS Overview [↗](#) (Wk 20)
- OS Introduction [↗](#) (Wk 21)
- Interrupts and I/O [↗](#) (Wk 21)
- Threads [↗](#) (Wk 23)
- **Thread Scheduling** [↗](#) (Wk 24)
- Mutual Exclusion (MX) [↗](#) (Wk 25)
- MX in Java [↗](#) (Wk 25)
- MX Challenges [↗](#) (Wk 25)
- Virtual Memory I [↗](#) (Wk 26)
- Virtual Memory II [↗](#) (Wk 26)
- Processes [↗](#) (Wk 27)
- Security [↗](#) (Wk 28)





## 1.2. Previously on OS ...

- What is [multitasking](#)?
- What are [blocking](#) system calls?
- What [types of threads](#) exist?
- What is [preemption](#)?

# 1.2.1. CPU Scheduling

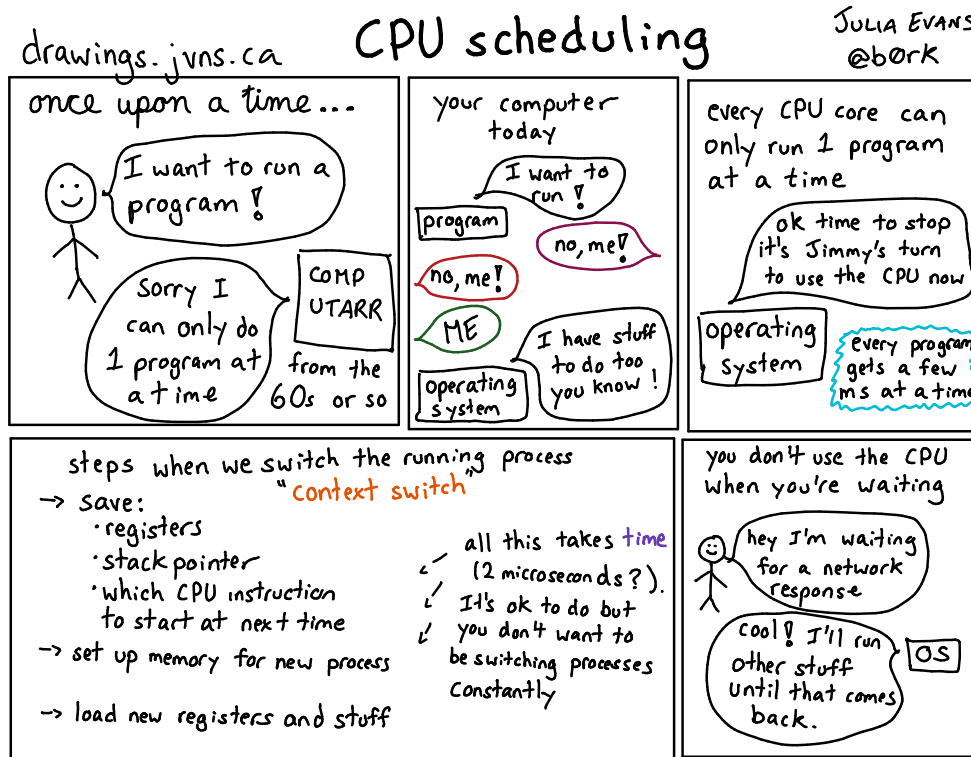


Figure © 2016 Julia Evans, all rights reserved; from [julia's drawings](#).  
Displayed here with personal permission.



## 1.3. Today's Core Questions

- How does the OS manage the shared resource CPU?  
What goals are pursued?
- How does the OS distinguish threads that could run on the CPU from those that cannot (i.e., that are blocked)?
- How does the OS schedule threads for execution?



## 1.4. Learning Objectives

- Explain thread concept (continued)
  - Including states and priorities
- Explain scheduling mechanisms and their goals
- Apply scheduling algorithms
  - FCFS, Round Robin





## 1.5. Retrieval Practice

- Before you continue, answer the following; ideally, without outside help.
  - What is a process, what a thread?
  - What does concurrency mean?
    - How does it arise?
  - What is preemption?



# 1.5.1. Thread Terminology

Threads, concurrency, and preemption

## 1. Select correct statements about thread terminology

- ☐ Running programs are managed as threads by the OS.
- ☐ Processes and threads are OS management units.
- ☐ Each thread may contain one or more threads.
- ☐ Concurrency can only arise on multi-core systems.
- ☐ Scheduling may lead to interleaved executions of multiple threads.

## 2. Select correct statements about preemption

- ☐ Concurrency can cause preemption.
- ☐ Preempted threads are garbage-collected by the OS.
- ☐ In response to an interrupt, a thread may be preempted.
- ☐ Preemption hinders effective caching.
- ☐ Management information on stacks can be used to resume preempted threads later on.





# Table of Contents

- 1. Introduction
- 2. Scheduling
- 3. Thread States
- 4. Scheduling Goals
- 5. Scheduling Mechanisms
- 6. Pointers beyond Class Topics
- 7. Conclusions



## 2. Scheduling



## 2.1. CPU Scheduling

- With multitasking, lots of threads **share resources**
  - Focus here: CPU
- **Scheduling** (planning) and **dispatching** (allocation) of CPU via OS
  - **Non-preemptive**, e.g., **FIFO scheduling** ▶
    - Thread on CPU until **yield** ↗, termination, or **blocking** ↗
  - **Preemptive**, e.g., **Round Robin scheduling** ▶
    - Typical case for desktop OSs
      1. Among all threads, schedule and dispatch one, say  $T_0$
      2. Allow  $T_0$  to execute on CPU for some time, then preempt it
      3. Repeat, go to step (1)
- (Similar decisions take place in industrial production, which you may know from operations management)



Scheduling is the planning of resource allocations. Here, we just consider the allocation of the resource CPU among multiple threads.

Concerning wording, the planning itself is called **scheduling**, while the allocation is called **dispatching**. Thus, after making a scheduling decision, the OS dispatches one thread to run on the CPU.

Two major scheduling variants are non-preemptive and preemptive ones. With non-preemptive scheduling, the OS allows the currently executing thread to continue as long as it wants. The bullet point names some situations when a thread might stop, which is when the next scheduling decision takes place.

With preemptive scheduling, the OS may pause, or preempt, a thread in the middle of its execution although it could continue with more useful work on the CPU. Here, the OS uses a timer to define the length of some time slice, for which the dispatched thread is allowed to run at most. If the thread executes a blocking system call or terminates before the timer runs out, the OS cancels the timer and makes the next scheduling decision. When the timer runs out, it triggers an interrupt, causing the interrupt handler to run on the CPU for the next scheduling decision.



## 2.2. Sample Scheduling Goals

- Scheduling is hard, as various goals with **trade-offs** exist
  - Trade-off: Improvement for one goal may negatively affect others (discussed subsequently)
- Sample goals
  - Efficient resource usage
  - Fairness (e.g., equal CPU shares per thread)
  - Response time ([definition ►](#))
  - Throughput ([definition ►](#))





# 3. Thread States



## 3.1. Reasons to Distinguish States

- Recall: Some threads may be blocked
  - E.g., wait for I/O operation to finish or `sleep` system call (recall [Simpler2Threads](#))
    - More generally, threads may perform `blocking` system calls
  - `Busy waiting` would be a waste of CPU resources
    - If other threads could run on CPU
- OS distinguishes **thread states** (details subsequently)
  - Tell threads apart that might perform useful computations (runnable) on the CPU from those that do not (blocked/waiting)
  - Scheduler does not need to consider waiting threads
  - Scheduler considers runnable threads, selects one, dispatches that for execution on CPU (which is then running)



## 3.2. OS Thread States

- Different OSs distinguish different sets of states; typically:
  - **Running**: Thread(s) currently executing on CPU (cores)
  - **Runnable**: Threads ready to perform computations
  - **Waiting** or **blocked**: Threads waiting for some event to occur
- OS manages states via **queues** ☒ (with suitable data structures)
  - **Run queue(s)**: Potentially per CPU core
    - Containing runnable threads, input for scheduler
  - **Wait queue(s)**: Potentially per event (type)
    - Containing waiting threads
      - OS inserts running thread here upon blocking system call
      - OS moves thread from here to run queue when event occurs



## 3.3. Thread State Transitions

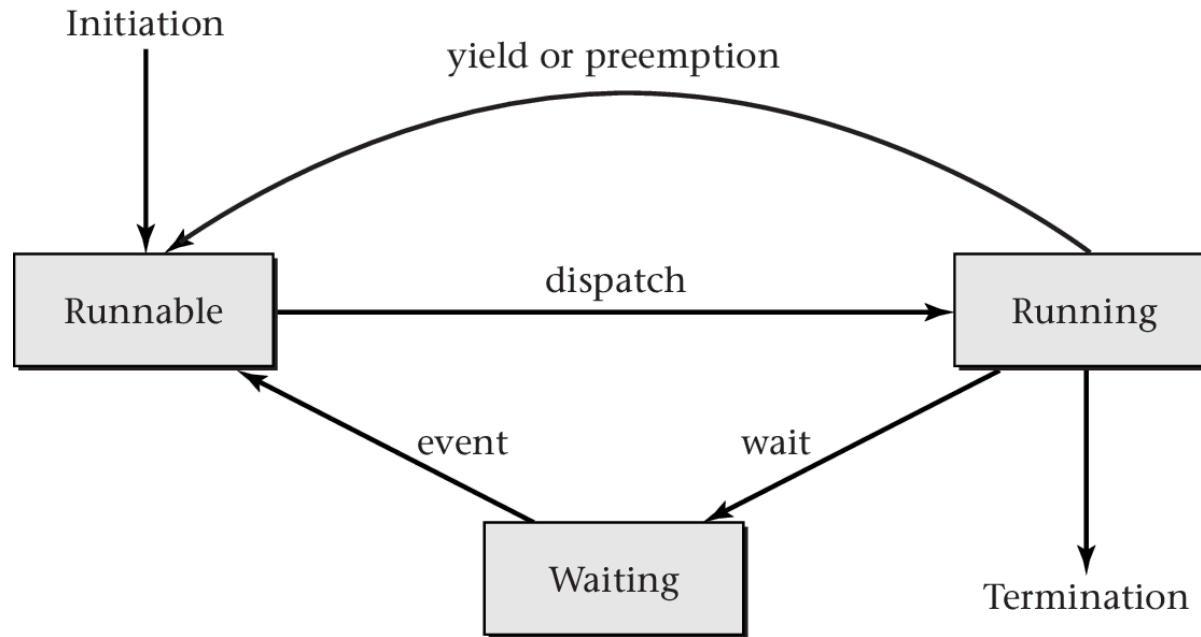


Figure 3.3 of [Hai17]

Figure by Max Hailperin under CC BY-SA 3.0; converted from  
GitHub



This diagram shows typical state transitions caused by actions of threads, decisions of the OS, and external I/O events. State changes are always managed by the OS.

Newly created threads, such as the ones you created in Java, are Runnable. When the CPU is idle, the OS' scheduler executes a selection algorithm among the Runnable threads and dispatches one to run on the CPU. When that thread yields or is preempted, the OS remembers that thread as Runnable.

If the thread invokes a blocking system call, the OS changes its state to Waiting. Once the event for which the thread waits has happened (e.g., a key pressed or some data has been transferred from disk to RAM), the OS changes the state from Waiting to Runnable. At some later point in time, that thread may be selected by the scheduler to run on the CPU again.

In addition, an outgoing arc Termination is shown from state Running, which indicates that a thread has completed its computations (e.g., the main function in Java ends). Actually, threads may also be terminated in states Runnable and Waiting, which is not shown here, but which can happen if a thread is killed (e.g., you end a program or shut down the machine).

# 3.4. Scheduling Vocabulary



## Scheduling and thread states

1. Select the correct statement about scheduling.

- ☐ If a thread's time slice runs out, the scheduler thread takes over.
- ☐ If a thread's time slice runs out, the OS blocks it.
- ☐ Runnable threads are eligible for scheduling decisions.
- ☐ After the OS changed a thread state from blocked to runnable, it dispatches that thread to run on a CPU core.
- ☐ After the OS preempted a thread, it changes its state to blocked.





# 4. Scheduling Goals





# 4.1. Goal Overview

- Performance
  - **Throughput**
    - Number of completed threads (computations, jobs) per time unit
    - More important for service providers than users
  - **Response time**
    - Time from thread start or interaction to useful reaction
- User control
  - **Resource allocation**
  - Mechanisms for urgency or importance, e.g., **priorities**




As computer users, we expect different goals from scheduling mechanisms, for which subsequent slides contain some details: First, we are usually interested in high performance in the senses of throughput and response time.

Second, we may want to exert some control to influence the scheduling decisions. For example, when you think of rented compute capacity, where you share resources with other customers, the resources allocated to you (including CPU time) depend on the amount of money you pay.

Besides, programmers can assign priorities to threads to indicate their relative urgency or importance.



## 4.1.1. Throughput

- To increase throughput, avoid idle times of CPU
  - Thus, reassign CPU when currently running thread needs to wait
  - Context switching necessary
- Recall: Context switching comes with **overhead** 
  - Overhead reduces throughput



## 4.1.2. Response Time

- Frequent context switches may help for small response time
  - However, their overhead hurts throughput
- Responding quickly to one thread may slow down another one
  - May use priorities to indicate preferences




## 4.1.3. Resource Allocation

- What fraction of resources for what purpose?
- **Proportional share scheduling**
  - E.g., multi-user machine: Different users obtain same share of CPU time every second
    - (Unless one pays more: Larger share for that user)
- **Group scheduling**: Assign threads to groups, each of which receives its proportional share  
→ Linux scheduler **later on** ►




## 4.2. Priorities in Practice

- Different OSs (and execution environments such as Java) provide different means to express priority
  - E.g., numerical priority, so-called niceness value, deadline, ...
  - Upon thread creation, its priority can be specified (by the programmer, with default value)
    - Priority recorded in [TCB](#) 
    - Sometimes, administrator privileges are necessary for “high” priorities
    - Also, OS tools may allow to change priorities at runtime



## 4.3. Thread Properties in Java

- **Java API** 
  - “Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority.”
  - May interpret as: Preemptive, **priority-driven** ► scheduling
- **Priorities via integer values**
  - Higher number → more CPU time
  - Preemptive
    - Current thread has highest priority
    - Newly created thread with higher priority replaces old one on CPU
    - Most of the time (above quote from API is vague)
- **Time slices not guaranteed (implementation dependent)**
  - **Starvation** ► possible



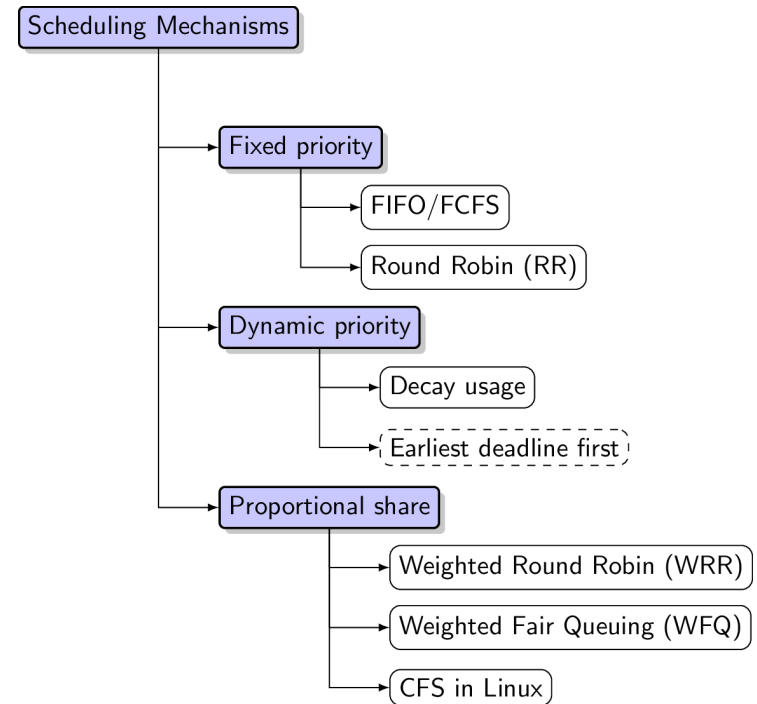
# 5. Scheduling Mechanisms





# 5.1. Three Families of Schedulers

- This section covers three families of schedulers
  1. Fixed thread priorities ►
  2. Dynamically adjusted thread priorities ►
  3. Controlling proportional shares of processing time ►
- Visualization on right shows structure of this section
  - With sample algorithms per family





## 5.1.1. Notes on Scheduling

- For scheduling with pen and paper, you need to know **arrival times** and **service times** for threads
  - Arrival time: Point in time when thread created
  - Service time: CPU time necessary to complete thread
    - (For simplicity, blocking I/O is not considered; otherwise, you would also need to know frequency and duration of I/O operations)
- OS does not know either ahead of time
  - OS creates threads (so, knowledge of arrival time is not an issue) and inserts them into necessary data structures during normal operation
  - When threads terminate, OS again participates
    - Thus, OS can compute service time after the fact
    - (Some scheduling algorithms require service time for scheduling decisions; then threads need to declare that upon start. Not considered here.)




## 5.2. Fixed-Priority Scheduling

- Use **fixed**, numerical **priority** per thread
  - Threads with higher priority preferred over others
    - Smaller or higher numbers may indicate higher priority: OS dependent
- Implementation alternatives
  - Single queue ordered by priority
  - Or one queue per priority
    - OS schedules threads from highest-priority non-empty queue
- Scheduling whenever CPU idle or some thread becomes runnable
  - Dispatch thread of highest priority
    - In case of ties: Run one until end (**FIFO** ►) or serve all **Round Robin** ►



## 5.2.1. Warning on Fixed-Priority Scheduling

- **Starvation** of low-priority threads possible
  - Starvation = continued denial of resource
    - Here, low-priority threads do not receive resource CPU as long as threads with higher priority exist
  - Careful design necessary
    - E.g., for hard-real-time systems (such as cars, aircrafts, power plants)
    - (Beware of [priority inversion](#) , a topic for a later presentation!)




## 5.2.2. FIFO/FCFS Scheduling

- FIFO = First in, first out
  - (= FCFS = first come, first served)
  - Think of queue in supermarket
- Non-preemptive strategy: Run first thread until completed (or blocked)
  - For threads of equal priority



## 5.2.3. Round Robin Scheduling

- Key ingredients
  - **Time slice** (quantum,  $q$ )
    - Timer with interrupt, e.g., every 30ms
  - **Queue(s)** for runnable threads
    - Newly created thread inserted at end
  - Scheduling when (1) timer interrupt triggered or (2) thread ends or is blocked
    1. Timer interrupt: **Preempt** running thread
      - Move previously running thread to end of runnable queue (for its priority)
      - Dispatch thread at head of queue (for highest priority) to CPU
        - With new timer for full time slice
    2. Thread ends or is blocked
      - Cancel its timer, dispatch thread at head of queue (for full time slice)
- Video tutorial in [Learnweb](#) 




## 5.3. Dynamic-Priority Scheduling

- With dynamic strategies, OS can adjust thread priorities during execution
- Sample strategies
  - **Earliest deadline first**
    - For tasks with deadlines — discussed in [\[Hai19\]](#), not part of learning objectives
  - **Decay Usage Scheduling**



## 5.3.1. Decay Usage Scheduling

- General intuition: **I/O bound** threads are at unfair disadvantage. (Why?)
  - **Decrease** priority of threads in **running state**
  - **Increase** priority of threads in **waiting state**
    - Allows quick reaction to I/O completion (e.g, user interaction)
- OS may manage one queue of threads per priority
  - Threads move between queues when priorities change
    - Falls under more general pattern of **multilevel feedback queue**  schedulers
- Technically, threads have a **base priority** that is adjusted by OS
  - Use **Round Robin scheduling** (for non-empty queue of highest-priority threads) after priority adjustments





## 5.3.2. Decay Usage Scheduling in Mac OS X

- OS keeps track of CPU **usage**
  - Usage increases linearly for time spent on CPU
    - Usage recorded when thread leaves CPU (yield or preempt)
  - Usage **decays** exponentially while thread is waiting
- Priority adjusted downward based on CPU usage
  - Higher adjustments for threads with higher usage
    - Those threads' priorities will be lower than others



## 5.3.3. Variant in MS Windows

- Increase of priority when thread leaves waiting state
  - Priority **boosting**
  - Amount of boost depends on wait reason
    - More for interactive I/O (keyboard, mouse) then other types
- After boost, priority decreases linearly with increasing CPU usage



## 5.4. Proportional-Share Scheduling

- Simple form: **Weighted** ◀ **Round Robin** (WRR)
  - Weight per thread is factor for length of time slice
  - Discussion
    - Con: Threads with high weight lead to long delays for others
    - Pro: Fewer context switches than following alternative
    - (See [next slide](#) ▶)
- Alternative: **Weighted fair queuing** (WFQ)
  - Uniform time slice
  - Threads with lower weight “sit out” some iterations



## 5.4.1. WRR vs WFQ with sample Gantt Charts

- Threads T1, T2, T3 with weights 3, 2, 1; service times > 30ms;  $q = 10\text{ms}$ 
  - Supposed order of arrival: T1 first, T2 second, T3 third
  - If threads are not done after shown sequence, start over with T1

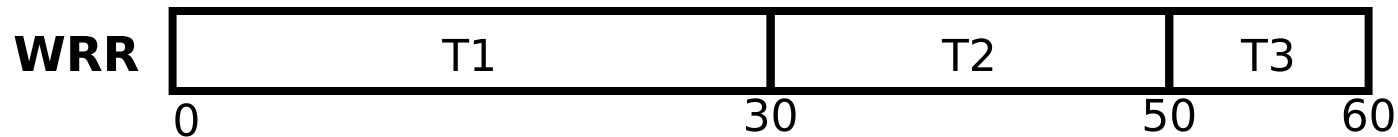


Figure  
under CC0  
1.0

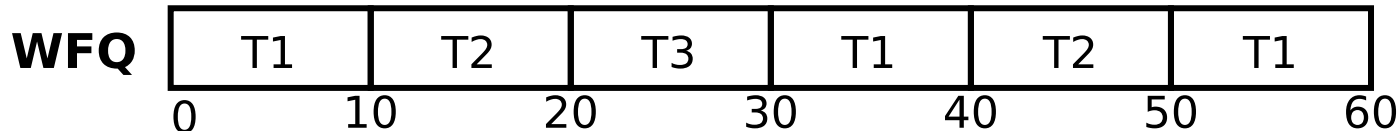


Figure  
under CC0  
1.0



# 5.5. CFS in Linux

- CFS = Completely fair scheduler
  - Actually, variant of ◀ WRR above
    - Weights determined via so-called **niceness** values
      - (Lower niceness means higher priority)
- **Core idea**
  - Keep track of how long threads were on CPU
    - Scaled according to weight
    - Time spent on CPU called **virtual runtime**
      - Represented efficiently via **red-black tree** 🚀
  - Schedule thread that is furthest behind
    - This thread received least amount of CPU time → allow to catch up
    - Until preempted or time slice runs out
  - (Some details in [Hai19])



## 5.6. When to Schedule

- Unless explicitly specified otherwise, we consider **preemptive** scheduling with **time slices** and **priorities**
  - Threads may be removed from CPU before they are “done”
    - As with Linux kernel, under stricter interpretation than ◀ **Java’s**:
      - “All scheduling is preemptive: if a thread with a higher static priority becomes ready to run, the currently running thread will be preempted and returned to the wait list for its static priority level. ↗”
- Scheduling based on thread states, priorities, and time slices
  - Sample **events** that may initiate scheduling
    - Thread state(s) change
      - E.g., thread created or finished, blocking system call, I/O finished; later: (un-) locking ↗
    - Thread priorities change
    - Time slice runs out



Let us revisit *when* scheduling takes place. It turns out that precise answers depend on design and implementation decisions. We consider *preemptive* scheduling of threads where *time slicing* enables multitasking; concerning *priorities*, we suppose that they are handled with preemption as in the case of Linux, for which a quote from a `man` page is shown on the slide.

Importantly, the scheduler only considers *runnable* threads. Thus, state changes may require a scheduling decision. E.g., if a thread invokes a blocking systems call, some other thread needs to be selected to run. Similarly, when an event occurs that makes one or more previously blocked threads runnable, scheduling *may* happen: If a newly runnable thread has highest priority, the currently running one is preempted and scheduling takes place to select a new one; otherwise, the currently running thread is likely to continue (but there may be implementations that disagree).

Similarly, if the priority of the currently running thread decreases below that of other threads or if some other thread or threads gain highest priority, scheduling must take place. In contrast, the creation of new threads with low priority does not require scheduling.

Clearly, preemptive scheduling also takes place when the OS believes that the current thread ran long enough.

Be careful not to confuse interrupt processing with scheduling. Indeed, some of the above events involve interrupts while others do not: What matters are *events* with relevance to scheduling as just discussed; whether an interrupt was involved in an event is less important.

## 5.7. Self-Study Task for Scheduling

This task is available for self-study in [Learnweb](#) .

Perform Round Robin scheduling given the following situation:

q=4	Thread	Arrival Time	Service Time
	T1	0	3
	T2	1	6
	T3	4	3
	T4	9	6
	T5	10	2



# 6. Pointers beyond Class Topics

# 6.1. Fair Queuing

- **Fair queuing** is a general technique
  - Also used, e.g., in computer networking
- Intuition
  - System is **fair** if at all times every party has progressed according to its share
    - This would require infinitesimally small steps
- Reality
  - Approximate fairness via “small” discrete steps
  - E.g., ◀CFS

## 6.1.1. CFS with Blocking

- Above description of ◀ CFS assumes runnable threads
- Blocked threads lag behind
  - If blocked briefly, allow to catch up
  - If blocked for a long time (above threshold), they would deprive all other threads from CPU once awake again
    - Hence, counter-measure necessary
      - Give up fairness
      - Forward virtual runtime to be slightly less than minimum of previously runnable threads
- Effect similar to dynamic priority adjustments of decay usage schedulers

## 6.1.2. CFS with Groups


- CFS allows to assign threads to (hierarchies of) groups
  - Scheduling then aims to treat groups fairly
- For example
  - One group per user
    - Every user obtains same CPU time
  - User-defined groups
    - E.g., multimedia with twice as much CPU time as programming (music and video running smoothly despite compile jobs)

# 7. Conclusions

## 7.1. Summary

- OS performs scheduling for shared resources
  - Focus here: CPU scheduling
  - Subject to conflicting goals
- CPU scheduling based on thread states and priorities
  - Fixed vs dynamic priorities vs proportional share
  - CFS as example for proportional share scheduling

# Bibliography

[Hai19] Hailperin, Operating Systems and Middleware – Supporting Controlled Interaction, revised edition 1.3.1, 2019. <https://gustavus.edu/mcs/max/os-book/> 

# License Information

This document is part of an [Open Educational Resource \(OER\)](#) course on Operating Systems. [Source code and source files are available on GitLab](#) under [free licenses](#).

Except where otherwise noted, the work “OS04: Scheduling”, © 2017-2023 [Jens Lechtenbörger](#), is published under the [Creative Commons license CC BY-SA 4.0](#).

*No warranties are given. The license may not give you all of the permissions necessary for your intended use.*

In particular, trademark rights are *not* licensed under this license. Thus, rights concerning third party logos (e.g., on the title slide) and other (trade-) marks (e.g., “Creative Commons” itself) remain with their respective holders.





