Scheduling

(Usage hints for this presentation)

IT Systems, Summer Term 2024

Dr. Jens Lechtenbörger (License Information)

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

Speaker notes

This presentation is an introduction to scheduling in operating systems.

1. Introduction

Speaker notes

Let us look at essential questions and terminology of our topic.

1.1. 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?

(Based on Chapter 3 of (Hailperin 2019))

Speaker notes

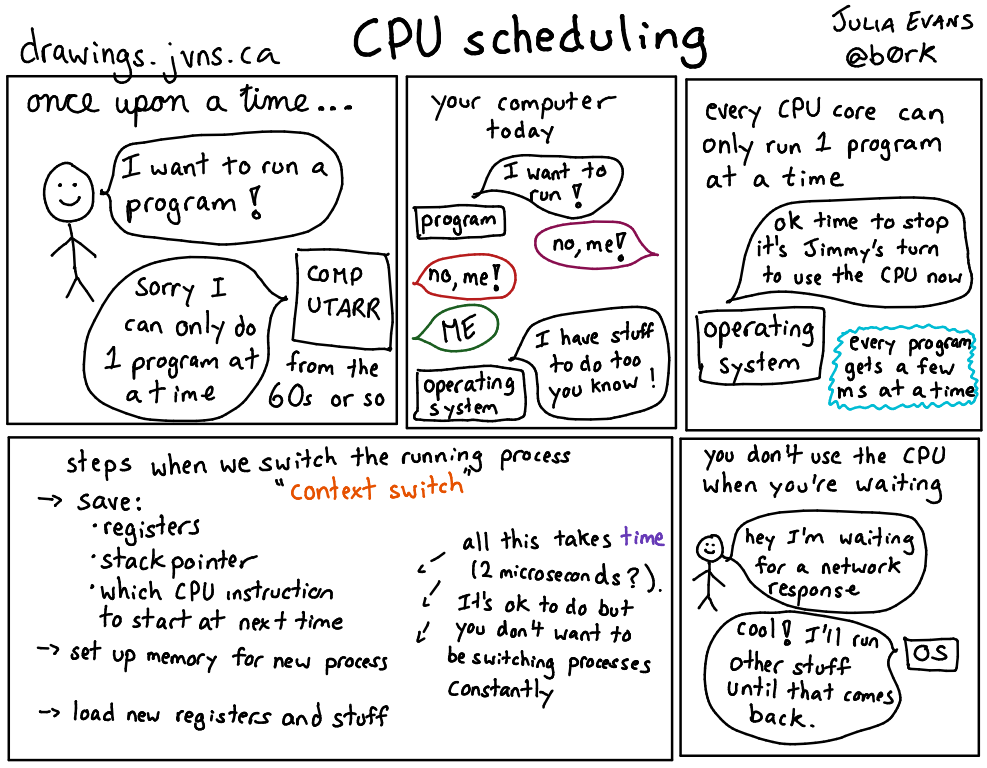
This presentation addresses the following questions:

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?

1.1.1. CPU Scheduling



CPU scheduling

Speaker notes

This drawing illustrates core ideas of scheduling. Importantly, the OS performs scheduling to enable multitasking, where threads of multiple programs take turns on a single CPU core. When taking turns, context switches take place, which add some overhead. Nevertheless, switching from a waiting thread to another thread is essential for an efficient use of the resource CPU.

1.2. Learning Objectives

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

Speaker notes

Take some time to think about the learning objectives specified here.

1.3. Retrieval Practice

- Before you continue, answer the following; ideally, without outside help.
  - What is a process, what a thread, what multitasking?
  - What does concurrency mean?
    - How does it arise?
  - What are blocking system calls?
  - What is thread switching?

Speaker notes

Before you continue, answer the questions listed here, ideally, without outside help.

1.3.1. 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.

Speaker notes

Take this quiz.

Agenda

- 1. Introduction
- 2. Scheduling
- 3. Thread States
- 4. Scheduling Mechanisms
- 5. Conclusions

Speaker notes

The agenda for this presentation is as follows.

After this introduction, we focus on general ideas related to CPU scheduling. Afterwards, we revisit the thread concept and add states as major piece of information for scheduling. Based on this updated picture, we explore different scheduling mechanisms.

Conclusions end the presentation.

2. Scheduling

Speaker notes

Let us see details of CPU 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 termination, blocking, yield
  - **Preemptive**, e.g., Round Robin scheduling
    - Typical case for desktop OSs
    - Illusion of parallel executions, even on single-core CPU
      1. Among all threads, schedule and dispatch one, say T0
      2. Allow T0 to execute on CPU for some time, then preempt it
      3. Repeat, go to step (1)
- (Similar decisions in operations management)

Speaker notes

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: Clearly, when the currently running thread terminates, i.e., executes its last instruction, the CPU is free for another thread. Similarly, when the currently running thread is blocked, a different thread should continue. Finally, a thread may give up the CPU voluntarily, which is called **yielding** and leads to a scheduling decision by the OS. As a side note, a thread may yield for performance reasons, e.g., after releasing a heavily requested resource.

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.

By rapidly switching between various threads and allowing them to run for brief intervals, the OS can create the illusion of parallel executions even when only a single CPU core is available.

As a side note, similar decisions take place in industrial production, which you may know from operations management.

2.2. Sample Scheduling Goals

- Scheduling is hard; various goals with **trade-offs**
  - Improvement for one goal may negatively affect others
- Performance
  - **Response time**
    - Time from thread start or interaction to useful reaction
  - **Throughput**
    - Number of completed threads (computations, jobs) per time unit
    - More important for service providers than users
- User control
  - **Resource allocation**
  - Mechanisms for urgency or importance, e.g., **priorities**

Speaker notes

Scheduling is no easy task as it comes with conflicting goals, where improvements for one goal may negatively affect other goals.

For example, we are usually interested in high performance, which can be measured with throughput and response time. For ordinary computer users, response time is probably the most obvious and most important performance measure. When we interact with the computer, e.g., via keyboard, touch, or mouse, we expect an immediate reaction. If that reaction takes “too long”, focused work might suffer.

Besides, throughput measures the number of tasks that complete their execution in a given time period. As ordinary computer users rarely use their machines for the completion of numerous tasks, high throughput might be less relevant for them.

In contrast, if you think of batch processing, say for deep learning based on millions of training examples to build a model, which takes hours to complete, then throughput is more relevant than response time.

Now, the trade-off is as follows: As each context switch incurs an overhead, fewer context switches are preferable for high throughput. However, with fewer context switches, it takes longer for each thread to receive CPU time, which might increase 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.

2.3. Thread Priorities

- Different OSs (and execution environments such as Java) treat priorities differently
  - 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 changing priorities at runtime
  - Scheduling takes priorities into account
    - Potentially with preemption

Speaker notes

Different operating systems and execution environments, such as Java, handle thread priorities differently. Various approaches exist to determine a thread’s priority, including numerical values, niceness levels, deadlines, and more. When creating a thread, programmers can specify its priority, which defaults to a standard value if not provided. The assigned priority is then stored in the Thread Control Block. In some cases, administrative privileges are necessary to create high-priority threads. Additionally, OS tools may exist to enable dynamic priority adjustments during runtime.

Ultimately, scheduling algorithms consider these priorities when allocating resources, potentially preempting low-priority threads when high-priority threads need CPU time.

3. Thread States

Speaker notes

We will see that the OS keeps track of different states for threads.

3.1. 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

Speaker notes

Operating systems categorize threads into various states, commonly including:

Running threads are those that are currently executing instructions on some CPU core.

Runnable threads are those that are ready to perform computations. Thus, they await allocation of a CPU core by the OS.

Waiting or **blocked** threads are those that are currently paused, because they cannot continue before some event has occurred.

To manage these states efficiently, OSs employ specialized data structures, so-called queues. To manage thread states, two types of queues are used:

One or more **run** queues, often dedicated to individual CPU cores, contain runnable threads. These serve as inputs for the scheduler.

One or more **wait** queues, sometimes organized by event type, hold blocked threads. Here, the OS records a running thread when it invokes a blocking system call. Once the awaited event occurs, the OS transfers the thread back to the run queue, reactivating it for further processing. Note that blocked threads on wait queues do not need to be considered by the scheduler.

3.2. Thread State Transitions

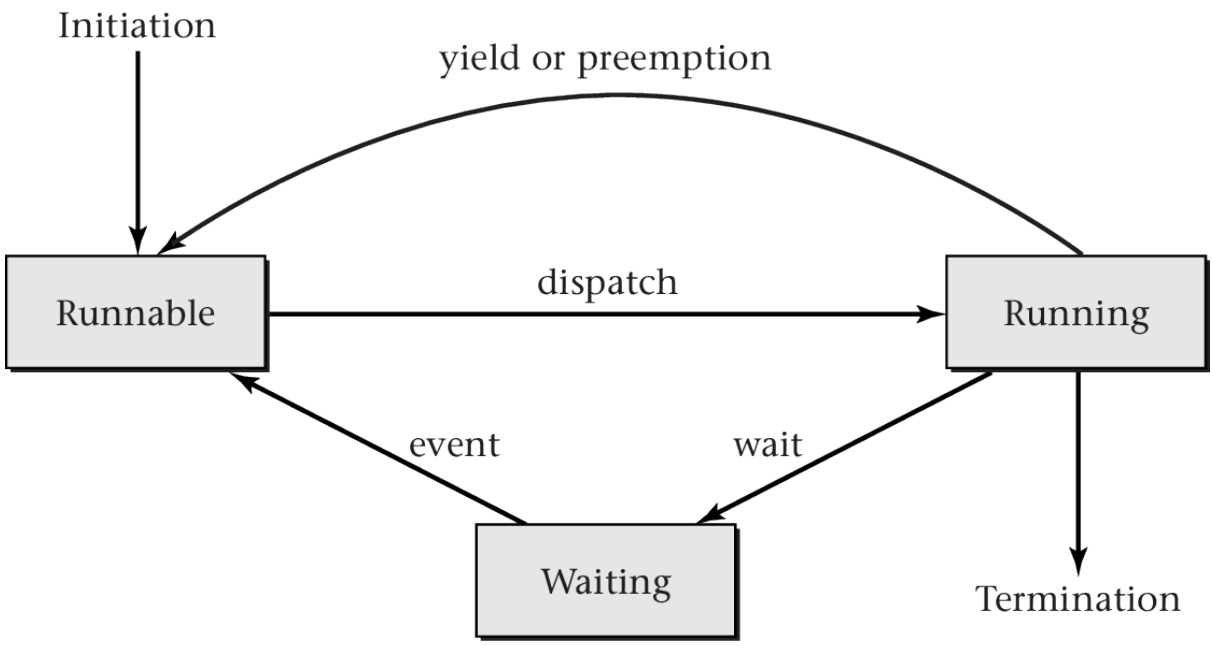


Figure 3.3 of cite:Hai17

Speaker notes

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).

3.3. 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.

Speaker notes

Take this quiz.

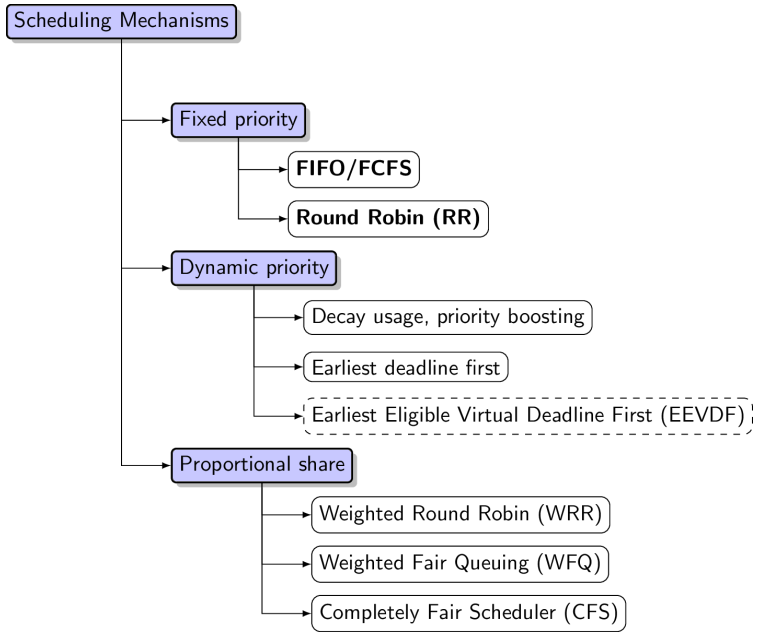
4. Scheduling Mechanisms

Speaker notes

Different scheduling mechanism exist, some of which we explore next.

4.1. Three Families of Schedulers

- Families of schedulers
  - Fixed thread priorities
    - E.g., FIFO , Round Robin
  - Dynamically adjusted thread priorities
    - E.g., decay usage in Mac OS X, priority boosting in Windows
  - Controlling proportional shares of processing time
    - E.g., weighted variants of other algorithms and Completely Fair Scheduler (CFS) in Linux
      - **Fairness**: All parties receive their share



Speaker notes

Subsequently, we look at sample scheduling algorithms for the three families visualized here. We revisit the algorithms in bold subsequently, while all algorithms except Earliest Eligible Virtual Deadline First are covered in our OS book.

The simplest family uses fixed thread priorities, for which we subsequently consider the non-preemptive variant FIFO and the preemptive variant Round Robin.

With dynamically adjusted thread priorities, the OS may change priorities of threads in response to certain events. Without going into too many details, decay usage scheduling in Mac OS and priority boosting in Windows are based on the following simple ideas: First, threads have some base priority, and the scheduler prefers threads with higher priority. Suppose that threads initially have the same base priority. Threads that are blocked, waiting for events, are at a disadvantage regarding the share of CPU time that they receive.

To counter that disadvantage, the scheduler of Mac OS gradually decays, or reduces, the priority of running threads. (And it increases the priority back to base priority when not running.) Then, if a previously blocked thread becomes runnable, its priority is high in relation to other threads that ran a lot. Consequently, it is likely that the scheduler picks the unblocked thread.

In contrast, with priority boosting in Windows, the priority of running threads is not changed, but when a previously blocked thread becomes runnable, the OS boosts, or increases, its priority. Thus, again, its priority is high in relation to other threads that ran a lot. Windows boosts the priority based on types of events. In particular, it increases the priority in larger amounts for threads that awaited user interaction, which helps to reduce response times perceived by users.

Finally, proportional share schedulers assign weights to threads, which express how large their share of CPU time should be in relation to the CPU time of other threads. E.g., some important thread may receive two times more CPU time per period of time than other threads.

Fairness is a general goal related to resource allocation, which is applicable beyond CPU scheduling. With fairness, at any point in time all parties should have received their promised share. Thus, the longer a thread runs, the more unfairly it is preferred over the others. To resolve this unfairness, CFS keeps track of how much threads run ahead and lag

behind. Then, it schedules the thread that lags behind the most. Scheduling is proportional as weights can be assigned that scale the time spent by threads on the CPU.

Finally, since 2023 🚀, Earliest Eligible Virtual Deadline First is replacing CFS, with some additions in 2024 🚀. Here, time spent on the CPU is still tracked for fairness, but used to compute a virtual deadline, where less CPU time leads to earlier deadlines. In addition, the virtual deadline is also affected by the time slice wanted by a thread, with shorter time slices leading to even earlier deadlines. The thread with the earliest virtual deadline is then dispatched by the scheduler.

Thus, I/O bound threads can be created with short time slices, which leads to early virtual deadlines with small response times once such threads become runnable. In essence, the virtual deadlines can be perceived as dynamic priorities.

4.1.1. 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, arrival time is known), inserts them into necessary data structures
  - 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.)

Speaker notes

In the past, students wondered about the following facts: For scheduling with pen and paper, arrival times and service times for threads are stated as part of tasks.

The arrival time simply is the point in time at which a thread is created.

The service time of a thread is the amount of time it needs for completion, executing on the CPU.

For simplicity, we do not consider blocking I/O operations for pen and paper tasks.

In contrast to tasks, the OS does not know arrival times or service times ahead of time, which is not a problem:

As the OS creates threads, it learns their arrival times.

When threads leave the CPU, either temporarily or at their termination, the OS again participates. Thus, the OS can compute service times during normal operation. More importantly, it “knows” that a thread wants more CPU time until it requests termination.

As a side note, there are scheduling algorithms that require service times for scheduling decisions, but we do not consider this.

4.2. 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
  - Subsequent examples: FIFO and Round Robin
  - Beware!
    - **Starvation** of low-priority threads possible
    - Recall: Starvation = continued denial of resource
      - Here, low-priority threads do not receive resource CPU as long as threads with higher priority exist

Speaker notes

Upon creation, each thread receives a fixed, numerical priority. The priorities of threads determine how much and how fast the OS assigns CPU time to threads. If multiple threads compete for CPU time, the OS prioritizes those with higher priorities over lower-priority ones.

The exact interpretation of what constitutes a “higher” or “lower” priority, depends on the OS being used.

There are two common ways to implement thread scheduling based on priority, namely single queue or separate queues per priority level.

First, with single queue ordered by priority, all threads are placed into a single queue but sorted according to their assigned priority levels. Highest-priority threads appear at the front, followed by progressively lower-priority threads. When the OS needs to schedule a new thread, it selects the first thread from the head of the queue.

Second, with separate queues per priority level, the OS maintains a set of separate queues, where each one contains only threads with identical priority levels. At any given moment, the OS schedules threads solely from the highest-priority non-empty queue. Once a higher-priority queue becomes empty, the OS moves down the list until it finds a queue containing threads ready for execution.

Subsequently, we look at FIFO and Round Robin scheduling.

Note that low-priority threads are at risk of starvation. Given enough runnable threads with higher priority, they might never be selected by a fixed-priority scheduler.

4.2.1. 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 yield or blocking)
  - For threads of equal priority

Speaker notes

First-in, first-out, also called first-come, first-serve, is a simple non-preemptive strategy. It just runs the selected thread (of highest priority) as long as that thread needs.

Checkout counters at supermarkets work in just the same fashion.

4.2.2. 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, yields, 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, yields, or is blocked
       - Cancel its timer, dispatch thread at head of queue (for full time slice)
- Video tutorial in Learnweb

Speaker notes

Round robin scheduling lies at the heart of many real-world scheduling algorithms. Round robin scheduling works with a predefined time slice, also called quantum, for which a dispatched thread is allowed to run on the CPU. The OS enforces the duration of the time slice with a clock that triggers an interrupt at the end of the time slice.

As explained already, the OS manages runnable threads in queues, among which scheduling decisions are made.

Scheduling happens when the timer interrupt is triggered or the current thread terminates, yields, or is blocked.

When the timer interrupt is triggered, its interrupt handler takes over, preempting the currently running thread. The interrupt handler performs a context switch and invokes the scheduler. The scheduler moves the previously running thread to the end of the queue for runnable threads (of its priority). Then, the scheduler dispatches the thread at the head of the highest-priority, non-empty queue to the CPU. Before performing a context switch to that thread, the scheduler configures the timer for a full time slice.

When the thread ends, yields, or performs a blocking system call, the OS performs a context switch to the scheduler. The scheduler cancels the current timer and updates thread queues appropriately: It deletes a terminated thread, it re-inserts a yielding thread at the end of a runnable queue, or it inserts a blocked thread into a wait queue. Afterwards, as in the previous case, the scheduler dispatches a new thread, again for a full time slice.

A video demonstrating round robin in action is available in Learnweb.

4.3. 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
      - “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

Speaker notes

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 the 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 the 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.

4.4. 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

Speaker notes

Please perform round robin scheduling for the scenario specified here.

5. Conclusions

Speaker notes

Let us conclude.

5.1. Summary

- OS performs scheduling for shared resources
  - Focus here: CPU scheduling
  - Subject to conflicting goals
- CPU scheduling based on thread states and priorities
  - Basic approaches use fixed priorities
  - Desktop OSs keep track of time on CPU, priorities, and events for more advanced scheduling

Speaker notes

Operating systems manage shared hardware resources like CPUs through scheduling algorithms. Focusing on CPU scheduling, these algorithms aim to allocate processor time fairly among various threads based on their priority levels and current state. However, “good” resource allocation is subject to conflicting goals such as maximizing overall system performance, ensuring responsiveness, minimizing latency, and preventing starvation of low-priority threads.

CPU scheduling typically relies on basic approaches utilizing fixed priorities. These methods involve categorizing threads into distinct groups based on their priorities and allocating CPU time accordingly. Overall, the OS aims to provide predictable behavior and acceptable response times.

Desktop operating systems usually employ more sophisticated techniques beyond simple fixed-priority schemes. They maintain detailed records about the amount of time spent on the CPU, priority settings, and pending events associated with each thread. Based on this information, desktop OSs dynamically adjust thread priorities and manage context switches, leading to improved fairness, reduced waiting times, and enhanced user experience. Nevertheless, striking an ideal balance remains challenging due to the inherent tradeoffs involved in balancing the demands of diverse workloads and varying application requirements.

Bibliography

Hailperin, Max. 2019. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1. https://gustavus.edu/mcs/max/os-book/.

Speaker notes

The bibliography contains references used in this presentation.

License Information

Source files are available on GitLab (check out embedded submodules) under free licenses. Icons of custom controls are by @fontawesome, released under CC BY 4.0.

Speaker notes