OS04: Scheduling

Based on Chapter 3 of [Hai17]

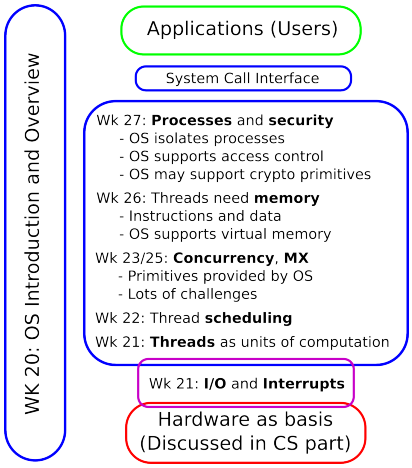
Jens Lechtenbörger

Computer Structures and Operating Systems 2019

1 Introduction

1.1 OS Plan

- OS Motivation (Wk 20)
- OS Introduction (Wk 20)
- Interrupts and I/O (Wk 21)
- Threads (Wk 21)
- Thread Scheduling (Wk 22)
- Mutual Exclusion (MX) (Wk 23)
- MX in Java (Wk 23)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- Processes (Wk 27)
- Security (Wk 27)
- Wrap-up (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

Warning! External figure **not** included: "CPU scheduling" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

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

Contents

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**
    - Thread on CPU until yield, termination, or blocking
  - **Preemptive**
    - 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 step (1)
- (Similar decisions take place in industrial production, which you may know from operations management)

2.2 Scheduling Goals

- Fairness
- Response time
- Throughput
- Efficient resource usage

Note: Above goals have **trade-offs** (discussed subsequently)

3 Thread States

3.1 Reasons to Distinguish States

- Recall: Some threads may be waiting (synonym: 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 to 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

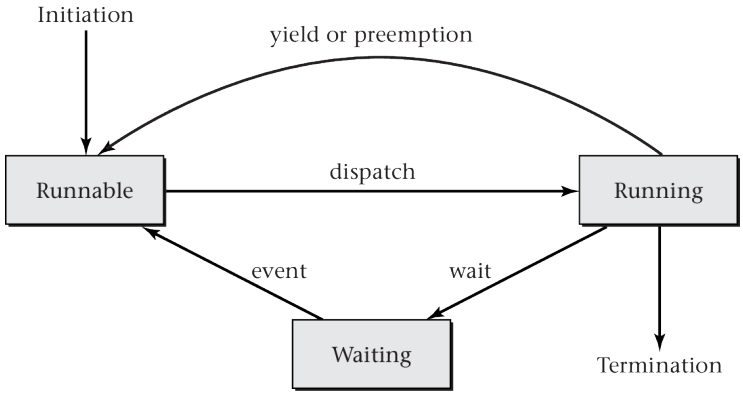


Figure 1: "Figure 3.3 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

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 Ticket to Exam - Task 1

Submit your solution to the following task in Learnweb.

Construct an example of your own choice where a thread changes states, involving all three states **running**, **runnable**, and **blocked**. Why and when do the state changes occur? At what points in time is the OS involved how?

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
  - **Urgency**
  - **Importance**
  - **Resource allocation**

4.1.1 Throughput

- Avoid threads on CPU that are waiting
- Context switching necessary
- Recall: Overhead of context switching reduces throughput
  - Minor source: Saving state
  - Major sources: Cache pollution, cache coherence protocols

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
  - Use urgency or importance to prioritize as explained next

4.1.3 Urgency vs Importance

- Urgent: When due?
  - Upcoming **deadline**; more urgent if sooner
- Important: How much at stake?
  - Tasks associated with **costs**
  - E.g., distinguish timely submissions of JiTT task vs seminar paper

4.1.4 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
- Rarely expressive enough to distinguish above cases entirely

5 Scheduling Mechanisms

5.1 Three Families of Schedulers

- Fixed thread priorities
- Dynamically adjusted thread priorities
- Controlling proportional shares of processing time

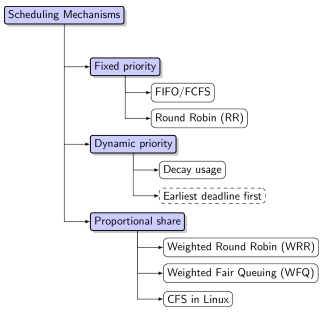


Figure 2: Scheduling Mechanisms

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 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

5.2.2 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 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 [Hai17], 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)
- Technically, threads have a **base priority** that is adjusted by OS
  - Use Round Robin scheduling after priority adjustments
- 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

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
- 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; q = 10ms
  - Supposed order of arrival: T1 first, T2 second, T3 third
  - If threads are not done after shown sequence, start over with T1

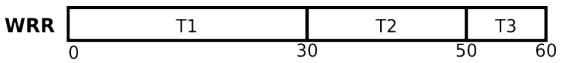


Figure 3: Figure under CC0 1.0

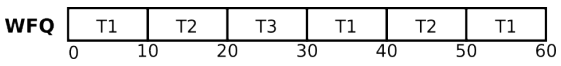


Figure 4: Figure under CC0 1.0

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 much threads were on CPU
    - Scaled according to weight
    - Called **virtual runtime**
      - Represented efficiently via red-black tree
  - Schedule thread that is furthest behind
    - Until preempted or time slice runs out
  - (Some details in in-class session and in [Hai17])

5.6 Goals and Scheduling Mechanisms

Table 1: Figure 3.8 of [Hai17]

Mechanism	Goals
Fixed priority	Urgency, importance
Earliest deadline first	Urgency
Decay usage	Importance, throughput, response time
Proportional share	Resource allocation

5.7 JiTT Assignments

Submit your solutions to the following tasks in Learnweb.

5.7.1 Scheduling

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

5.7.2 Ticket to Exam - Task 2

Perform Round Robin scheduling given the following situation:

q=3	Thread	Arrival Time	Service Time
	T1	0	2
	T2	1	5
	T3	9	4
	T4	10	4

5.7.3 Battle of the Threads

Prepare the in-class session by reading the instructions for the game Battle of the Threads.

5.7.4 Questions, Feedback, and Improvements

- What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (maybe on GitLab). You may submit individual questions as response to this task or ask questions in our Riot room and the Learnweb forum. Most questions turn out to be of general interest; please do not hesitate to ask and answer in forum and Riot room. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

6 In-Class Meeting

6.1 Comments 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!)

6.2 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

6.2.1 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

6.2.2 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)

6.3 Thread Properties in Java

- Java spec
  - "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
- Time slices not guaranteed (implementation dependent)
  - Starvation possible

6.4 Battle of the Threads

- Let's play a round of Battle of the Threads
  - Instructions
  - Last time, threads shared a magically synchronized data structure
  - This time, players share a data structure that is copied to and from a central location

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

[Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3, 2017. URL: https://gustavus.edu/mcs/max/os-book/.

License Information