# Operating Systems

Timo Hönig
Bochum Operating Systems and System Software (BOSS)
Ruhr University Bochum (RUB)

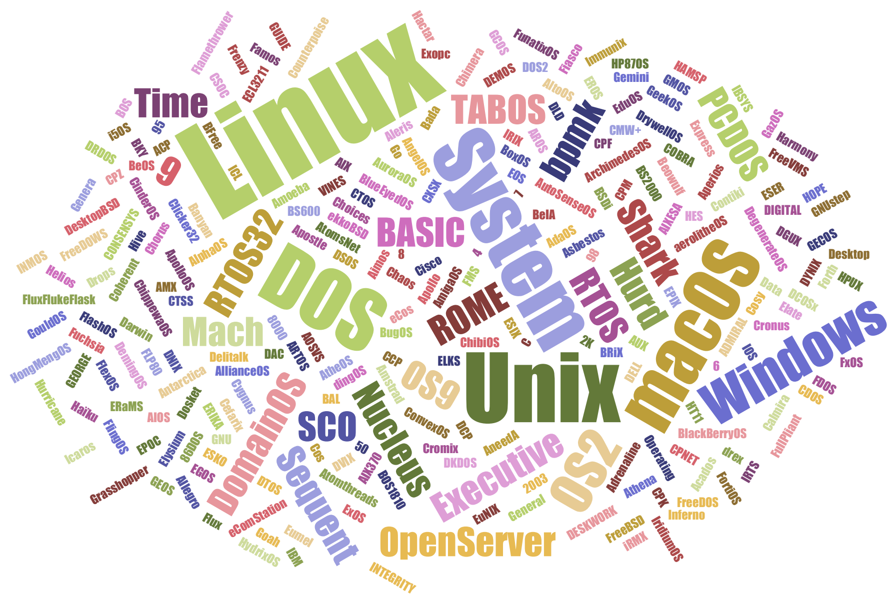
II. Abstractions and Data Structures
April 19, 2023 (Summer Term 2023)

RUB Faculty of Computer Science

RUHR UNIVERSITÄT BOCHUM RUB

www.informatik.rub.de
Chair of Operating Systems and System Software

# Agenda



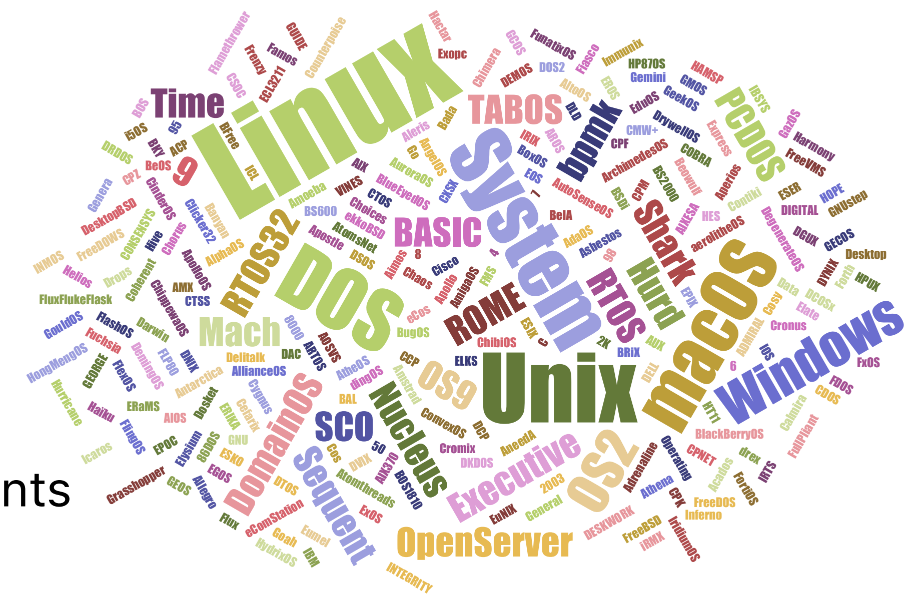

- Recap
- Organizational Matters
- Processes
  - CPU Scheduling
  - Synchronisation and Deadlocks
  - Inter-process Communication
- Memory Management
  - Primary Storage
  - Secondary Storage
- Multi- and Manycore Systems
- Summary and Outlook

**Literature References**

Silberschatz, Chapter 1, “Introduction”

Tanenbaum, Chapter 1, “Introduction”

# Recap

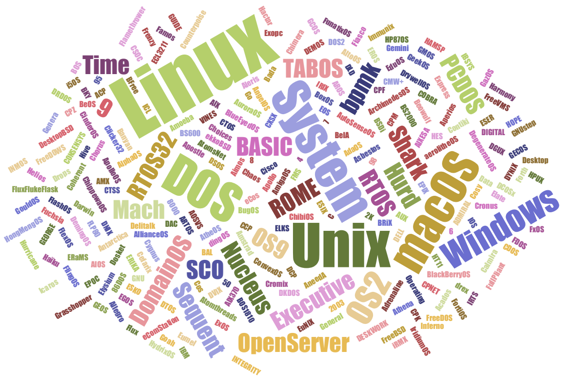



- what is an operating system?
  - discussion: definitions
  - diversity of demands
  - hardware and application requirements → special- or multipurpose systems
- a glimpse into history
  - serial/batch processing, resident monitor
  - CPU vs. I/O bursts → CPU utilisation
  - multiprogramming, interactive systems

# Agenda


- Recap
- Organizational Matters
- Processes
  - CPU Scheduling
  - Synchronisation and Deadlocks
  - Inter-process Communication
- Memory Management
  - Primary Storage
  - Secondary Storage
- Multi- and Manycore Systems
- Summary and Outlook

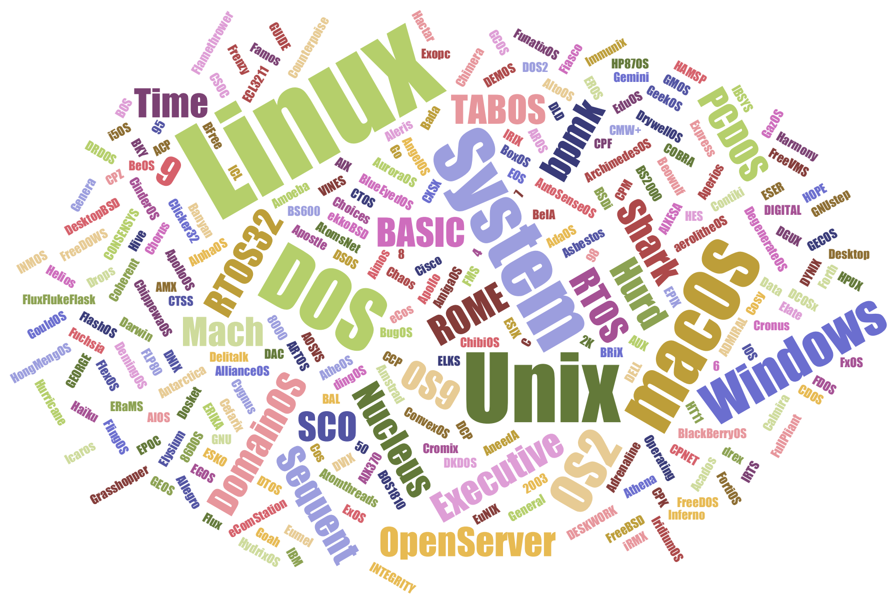
# Organizational Matters




- lecture
  - Wednesday, 10:15 – 11:45
  - format: synchronous, **hybrid**
    - → in presence (Room HID, Building ID)
    - → online lecture (Zoom)
  - language: English/German
  - exercises: **group allocation almost complete**
    - if you are not yet signed up → Moodle


- manage course material, asynchronous communication: Moodle
- `https://moodle.ruhr-uni-bochum.de/course/view.php?id=50698`

# Agenda




- Recap
- Organizational Matters
- Processes
  - CPU Scheduling
  - Synchronisation and Deadlocks
  - Inter-process Communication
- Memory Management
  - Primary Storage
  - Secondary Storage
- Multi- and Manycore Systems
- Summary and Outlook

# A process...

- Horning/Randell:

A *process* is a triple $(S, f, s)$, where $S$ is a state space, $f$ is an action function in that space, and $s$ is the subset of $S$ which defines the initial states of the process.* A process generates all the computations generated by its action function from its initial states. It is *(strictly) deterministic* if its action function is (strictly) deterministic. (Note that each computation generated by a strictly deterministic process is uniquely determined by its initial state.)† Similarly, processes are *temporal variants* if they differ only in having action functions which are temporal variants.

*Process Structuring* • 9

From (2, 2), $g_2$ generates only the computation $C_3 = \langle(2, 2), (2, 4), (3, 9)\rangle$. Since $g_2$ is strictly deterministic, $g_1$ is deterministic.

A *process* is a triple $(S, f, s)$, where $S$ is a state space, $f$ is an action function in that space, and $s$ is the subset of $S$ which defines the initial states of the process.* A process generates all the computations generated by its action function from its initial states. It is *(strictly) deterministic* if its action function is (strictly) deterministic. (Note that each computation generated by a strictly deterministic process is uniquely determined by its initial state.)† Similarly, processes are *temporal variants* if they differ only in having action functions which are temporal variants.

We frequently wish to study processes which are related, but not identical. A process is *weakly contained* in another process if all of its computations are also generated by that process, and *strongly contained* if, in addition, wherever its action function is defined, the action function of the containing process has (at least) all the same values. These definitions simply mean that the containing process can do everything that the contained process can, and possibly more. The distinction between weak and strong containment becomes important only in Section 3, where, as a result of a combination, the state variables of a process may assume values which would never be assigned by the process itself.

EXAMPLE: The process $P_1 = (S(V), g_1, s_1)$ where $s_1 = \{(2, 2)\}$, generates the computations $C_1$ and $C_3$ (and an infinite number of others). It strongly contains its temporal variant $P_2 = (S(V), g_2, s_1)$, which generates only $C_3$.

EXAMPLE: The process $P_3 = (S(V), f_1, s_1)$ is strictly deterministic, and generates only the computation $C_2$. It is strongly contained in the process $P_4 = (S(V), f_1, s_2)$, where $s_2 = \{(n, n) \mid n > 0\}$, which generates all computations of the form $\langle(n, n^i) \mid i = 1, 2, 3, \cdots\rangle$. However, $P_3$ is only weakly contained in the process $P_5 = (S(V), f_5, s_2)$, where $f_5(x, y) = \{y \leftarrow 2 \cdot y\}$, which generates all computations of the form $\langle(n, 2^i \cdot n) \mid i = 0, 1, 2, \cdots\rangle$.

From any action function, we can deduce the corresponding *successor function*, whose value in any state is the immediate successor (or if the action function is multiple-valued, the immediate successors) of that state, as defined by the action function. Conversely, given a successor function, we can infer a corresponding action function** by noting the changes to values of state variables between each state and its immediate successor(s). The two functions can thus be used interchangeably. We generally find action functions the more convenient, but some definitions in Section 4 are more naturally stated in terms of successor functions.

EXAMPLE: The action function $g_2$ corresponds to the successor function $G_2$, where

$G_2(2, 2) = (2, 4)$
$G_2(2, 4) = (3, 9)$

and $G_2$ is undefined elsewhere.

We now turn to the concept of "processor" and its relation to that of "process". A *processor* is a pair $(D, I)$, where $D$ is a physical "device" (we leave this term undefined) which can be placed in specified initial states, and $I$ is an *interpretation* of its physical status which indicates at what instants of time, and by what means, the device represents successive states. Each sequence of states following from an initial state is a *computation* of the processor.

The definition of "processor" as a pair may seem somewhat artificial, but is necessary to

* Wegner [35] defines an *information structure*

Horning J. J. and B. Randell: *Process Structuring*. ACM Computing Surveys, Vol. 5, No. 1, March 1973.
https://dl.acm.org/doi/10.1145/356612.356614

# A process...

- Dennis/van Horn (Programming Semantics for Multiprogrammed Computations, Communications of the ACM, 1966):

*„[...] ist das Aktivitätszentrum innerhalb einer* ***Folge von Elementar-operationen****. Damit wird ein Prozess zu einer* ***abstrakten Einheit****, die sich durch die Instruktionen eines* ***abstrakten Programms*** *bewegt, wenn dieses auf einem Rechner ausgeführt wird.“*

- Habermann (Introduction to Operating System Design, CMU, 1976):

*„[...] wird durch ein Programm* ***kontrolliert*** *und benötigt zur Ausführung dieses Programms einen* ***Prozessor****.“*

# A process...

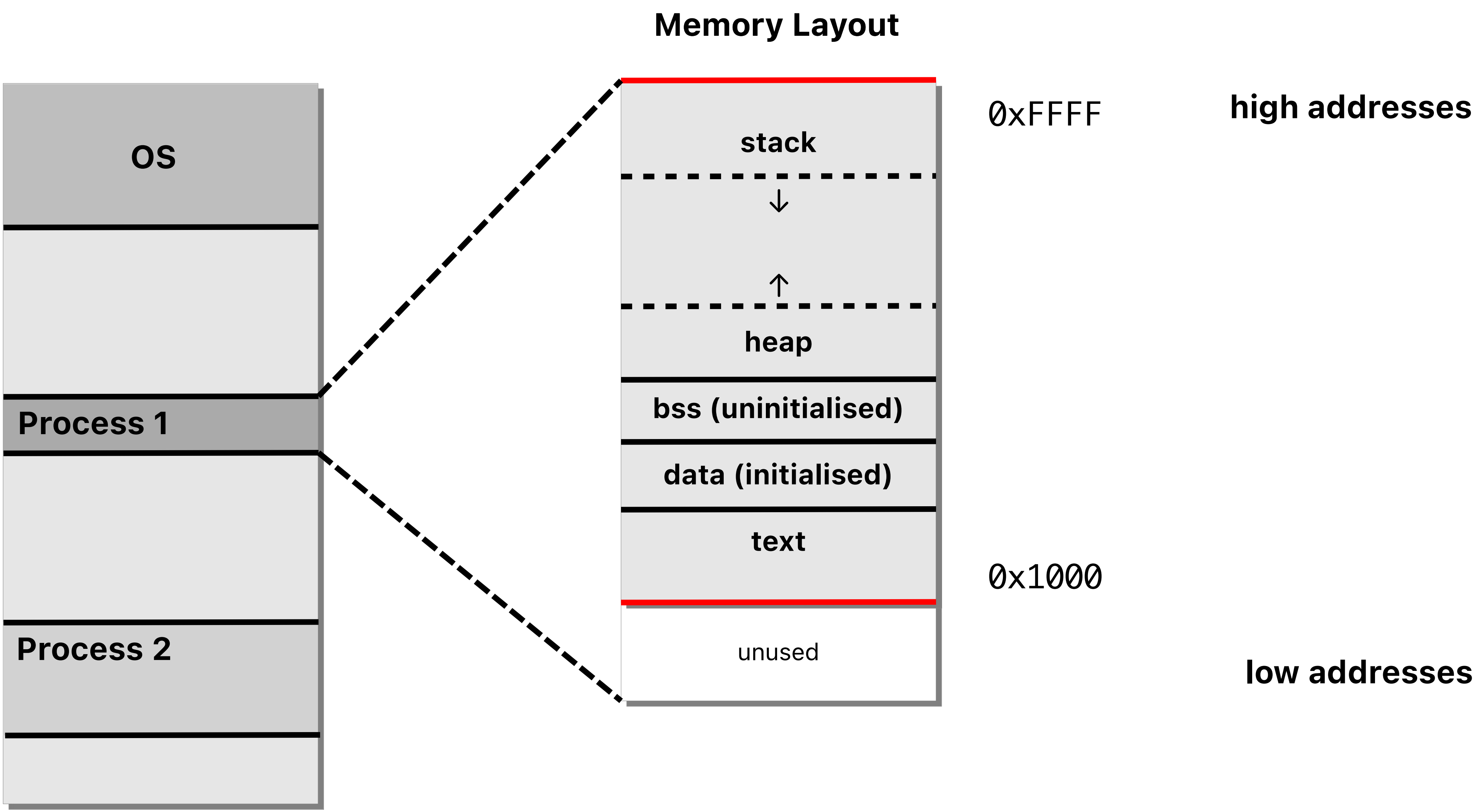
- unknown author:

> ***„[...] is a program in execution“***

- this includes a **process context** which usually contains...
  - **memory**: code, data, and stack segment (`text`, `data`, `bss`, `stack`, `heap`)
  - processor **register contents**, for example:
    - instruction pointer
    - stack pointer
    - general purpose registers
  - process **state**
  - user/group IDs, access rights
  - **priority**
  - (currently) **occupied** operating resources
    - open files, I/O devices, etc.

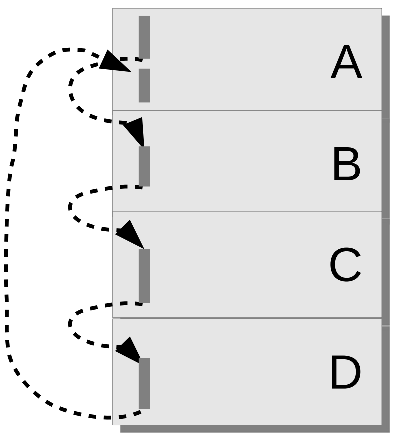
a process is represented by a **process control block** (PCB)

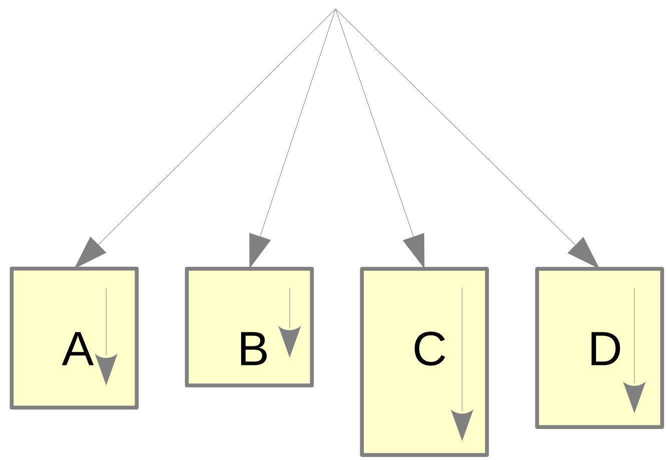
# A process...

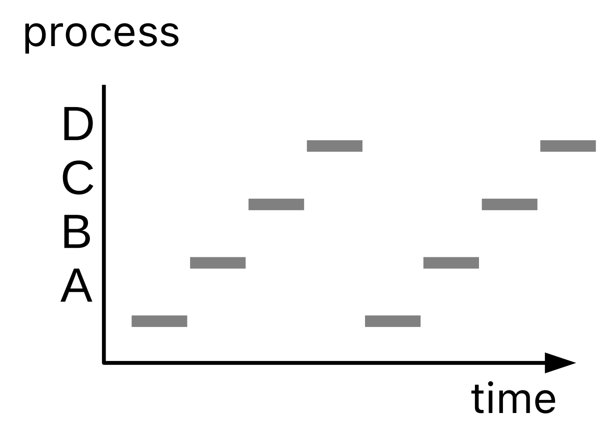

Memory Layout
OS
Process 1
Process 2
stack
↓
↑
heap
bss (uninitialised)
data (initialised)
text
unused
0xFFFF
0x1000
high addresses
low addresses

# Process Model

| Multi-program Operation | Concurrent Processes | Multiplexing of the CPU |
|---|---|---|



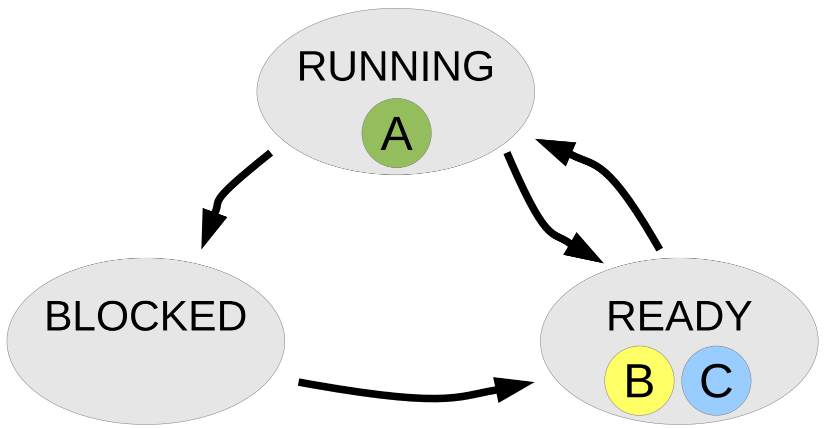




**Technical View**
- 1 instruction pointer
- context switch

**Conceptual View**
- 4 independent sequential, control flows

**Real-time View**
- only 1 process is active at any time (uni processor)
- Gantt chart

# Process Behaviour and States

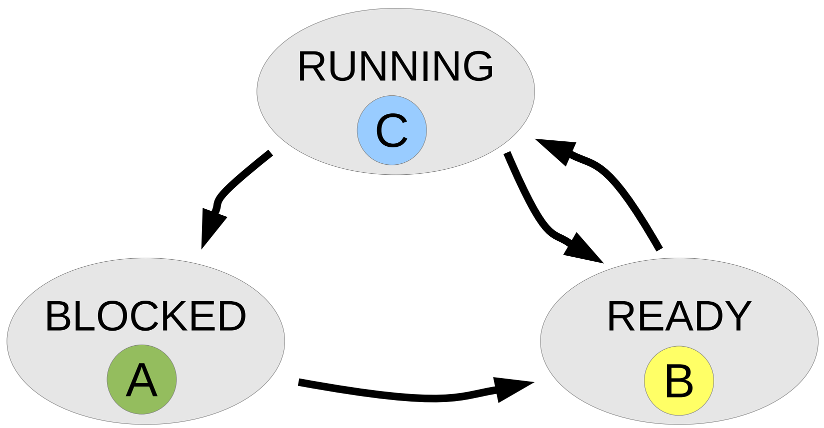




## Process States

- **RUNNING**
  - process is just being executed
- **READY**
  - process is ready to compute, is waiting for the CPU
- **BLOCKED**
  - process is waiting for the completion of an I/O activity

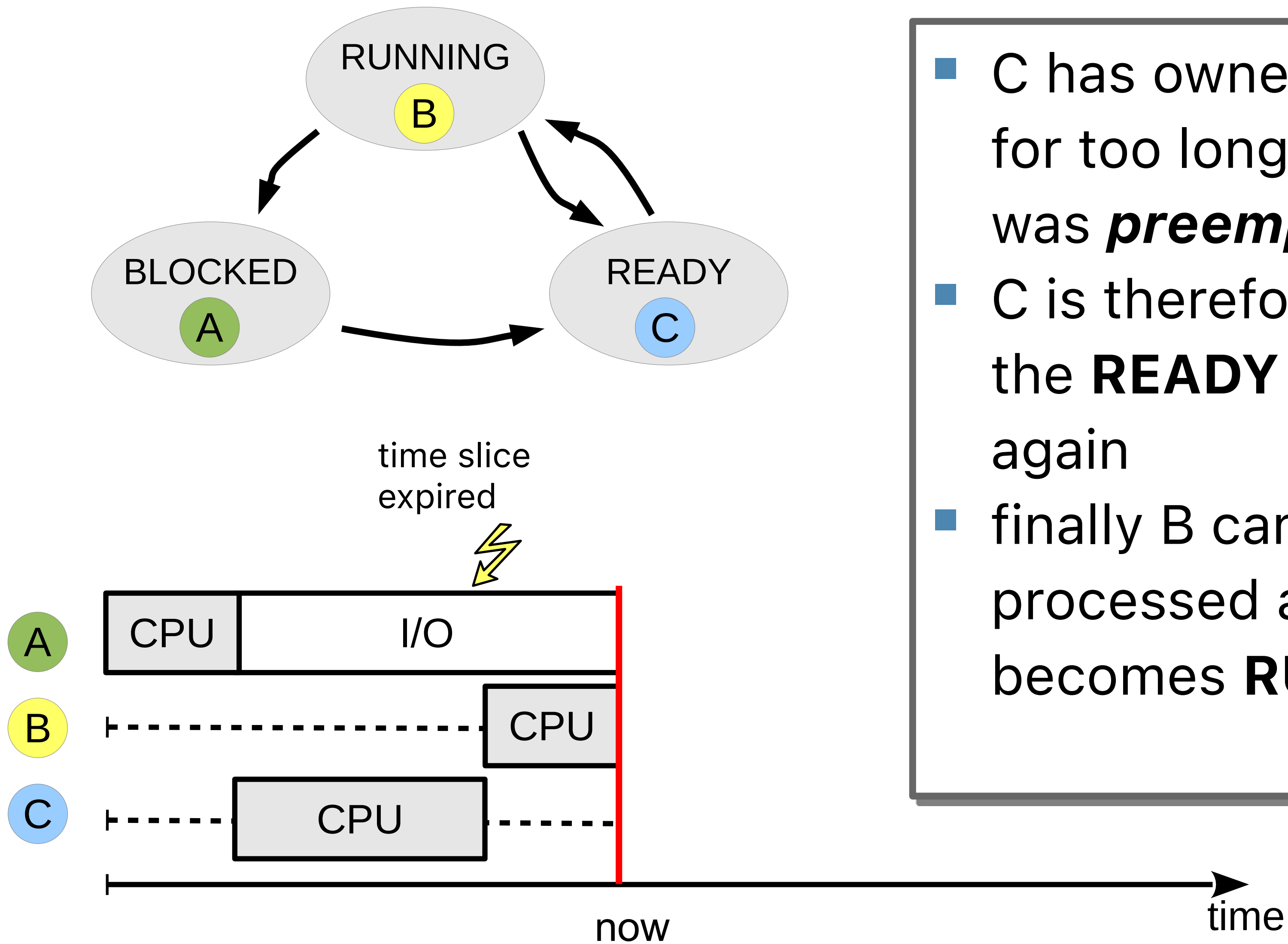
# Process Behaviour and States





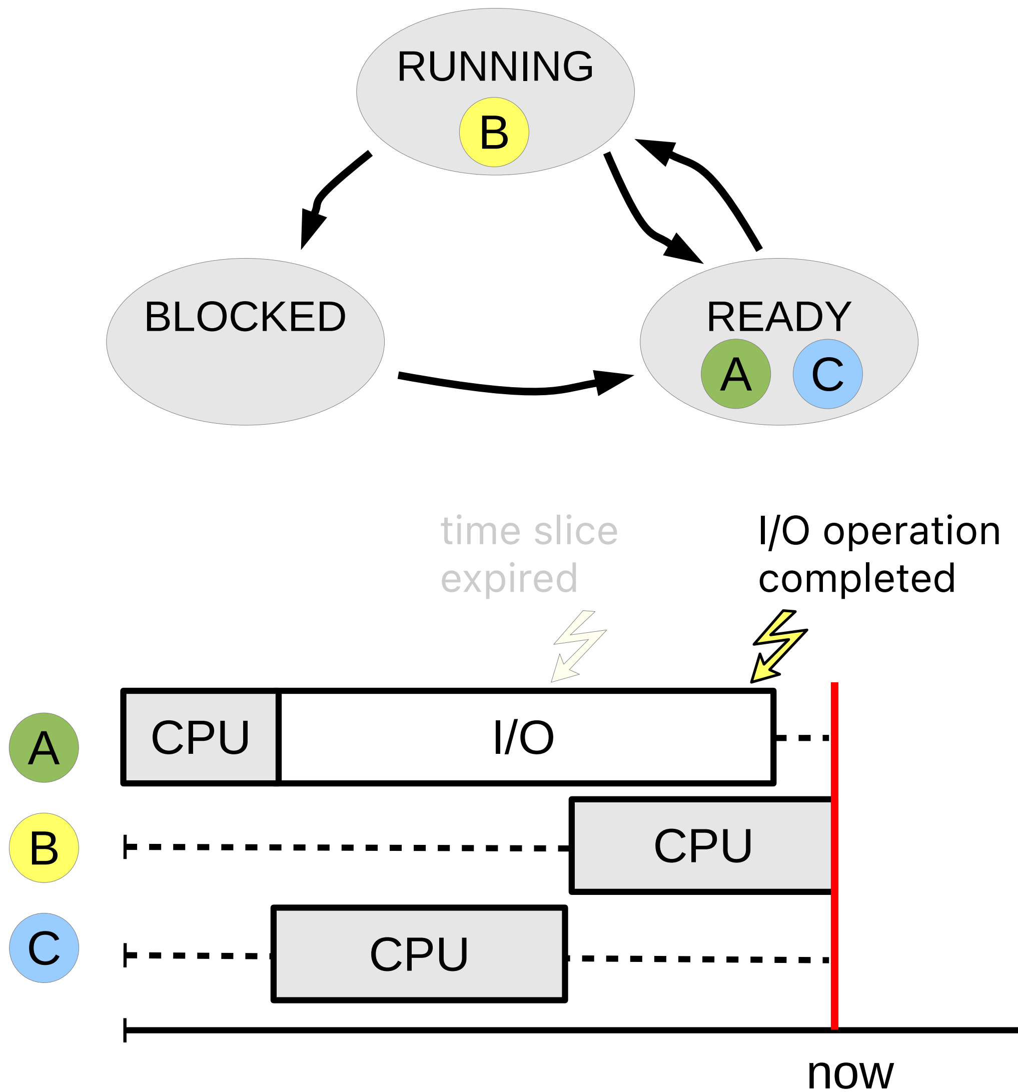
- process A has started an I/O operation and has entered the **BLOCKED** state
- since A does not need the CPU <u>now</u>, the operating system has selected process C and transferred it from **READY** to **RUNNING**
- a context switch from A to C took place

# Process Behaviour and States



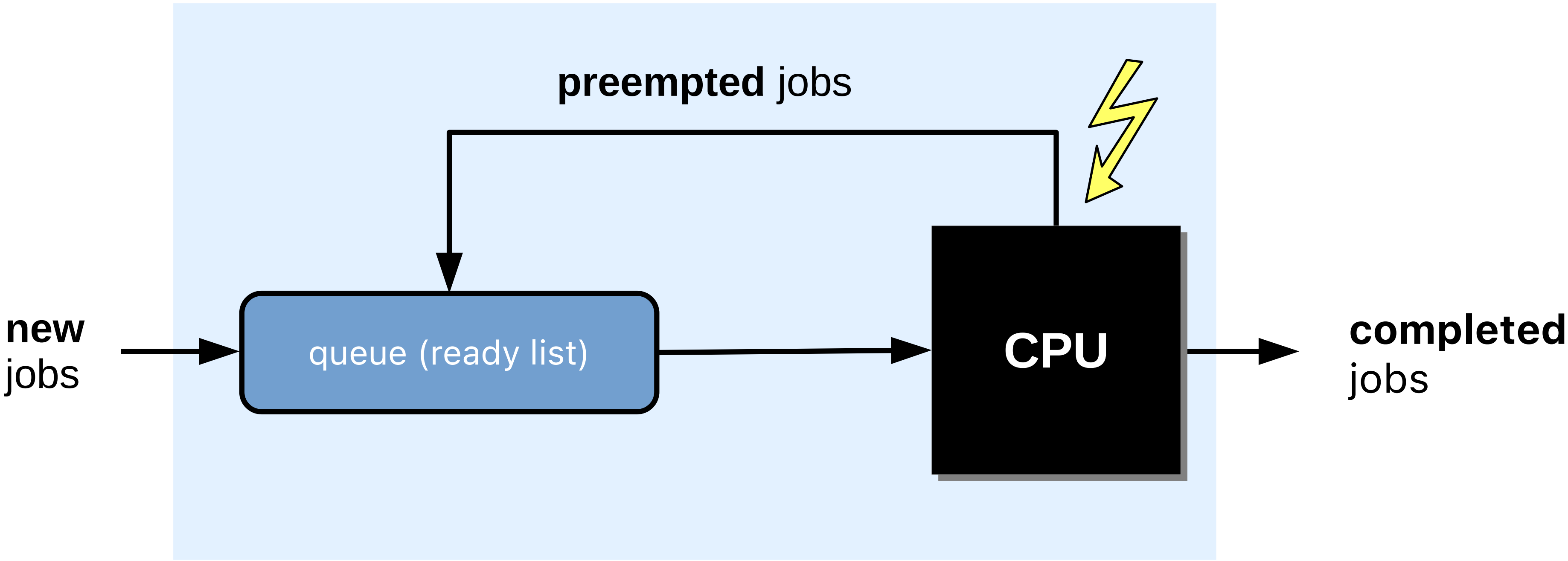

- C has owned the CPU for too long, and thus was ***preempted***
- C is therefore now in the **READY** state, again
- finally B can be processed and becomes **RUNNING**

# Process Behaviour and States




- the I/O operation of A is now completed
- hence, A now becomes **READY** and awaits for the allocation of the CPU

# CPU Scheduling




A **scheduling algorithm** is characterized by the order of processes in the queue and the conditions under which the processes are added to the queue (ready list).

# CPU Scheduling

- CPU scheduling (dt. CPU-Zuteilung, *Ablaufplanung*)
- ensures **ordered** flow of competing processes
- fundamental questions
  - what **types of events** lead to **preemption** (dt. *Verdrängung*)?
  - in which **order** should processes run?
- objective of a **scheduling algorithm**
  - user-oriented → short response times, low latency, for example
  - system-oriented → optimised CPU usage, for example
- **note:** no scheduling algorithm can meet all needs

# Process Synchronisation

- example: uncoordinated device access (i.e., printer)

Process A

```
...
print("Hallo Otto\n");
print("Ruf mich an.\n");
print("Tel.: 420815\n");
...
```

Process B

```
...
print("Karl-");
print("Ich mag dich.\n");
...
```

# Process Synchronisation

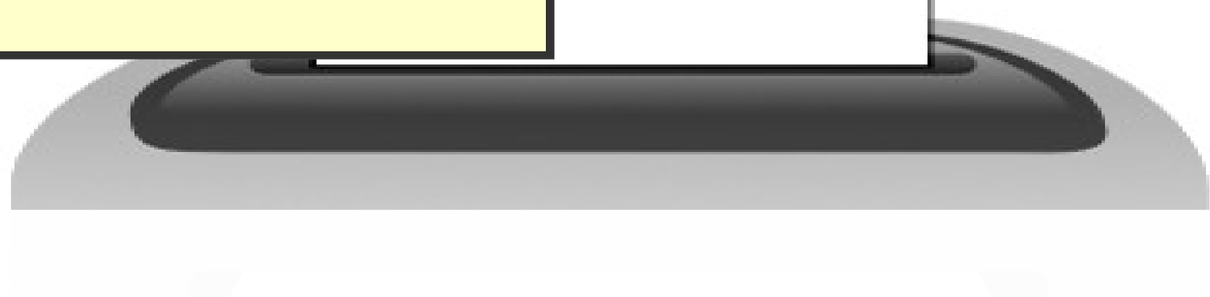
- example: uncoordinated device access (i.e., printer)

Process A

```
...
print("Hallo Otto\n");
print("Ruf mich an.\n");
print("Tel.: 420815\n");
...
```

Process B

```
...
print("Karl-");
print("Ich mag dich.\n");
...
```



- processes must access shared resources in a coordinated manner
- OS must provide necessary abstractions for well-organised resource access

# Process Synchronisation

- root cause: **critical sections**
- possible solution: **mutual exclusion**
  - mutex abstraction

Process A

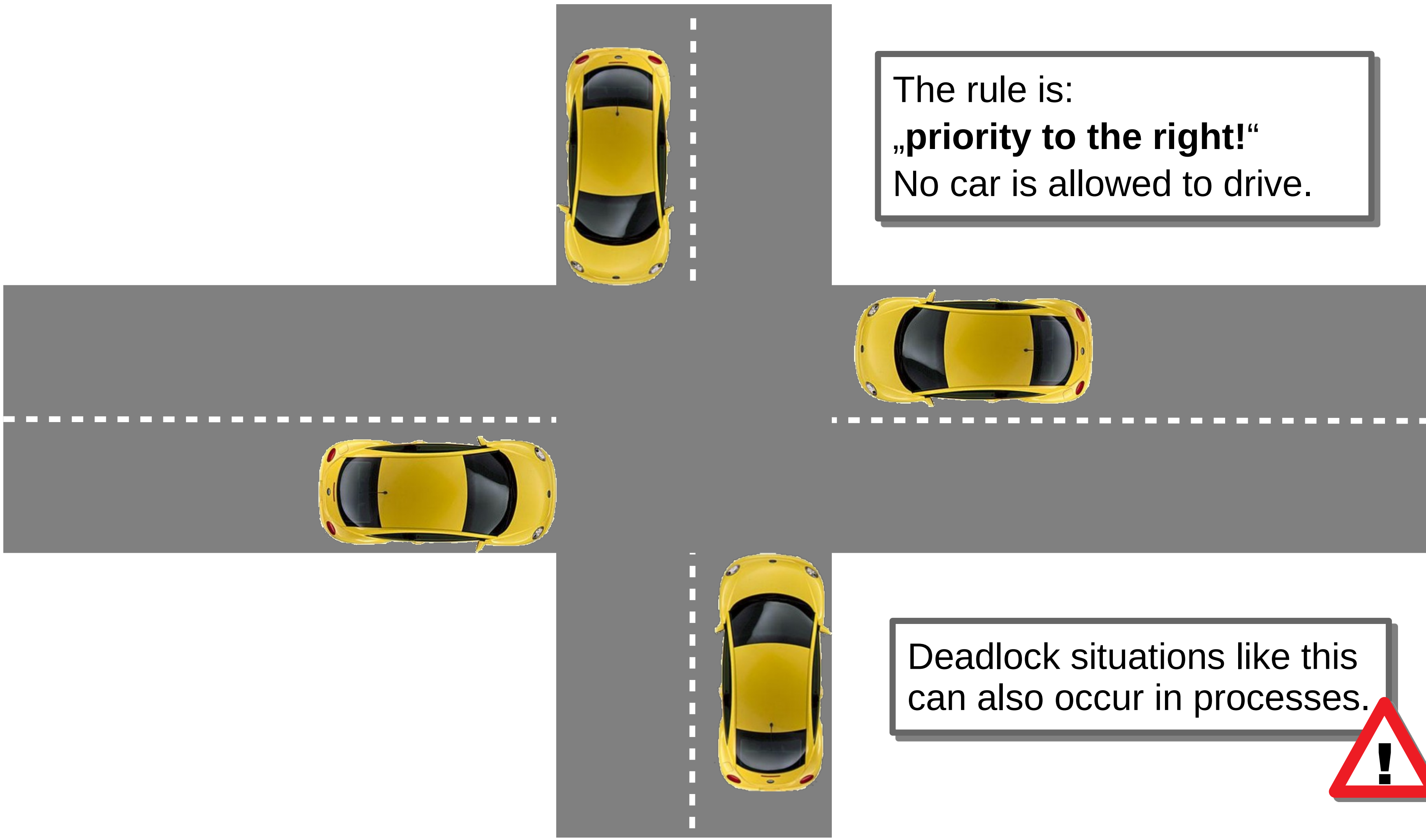
```
...
lock(&printer_mutex);
print("Hallo Otto\n");
print("Ruf mich an.\n");
print("Tel.: 420815\n");
unlock(&printer_mutex);
...
```

Process B

```
...
lock(&printer_mutex);
print("Karl-");
print("Ich mag dich.\n");
unlock(&printer_mutex);
...
```

If one of the processes A or B is between **lock** and **unlock**, the other one cannot pass the **lock** and blocks there until the critical section is free again (**unlock**).

# Deadlocks (dt. *Verklemmungen*)



The rule is:
**„priority to the right!“**
No car is allowed to drive.

Deadlock situations like this can also occur in processes.

# Inter-process Communication

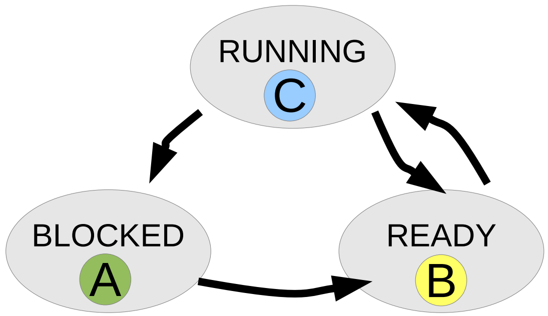
- enables multiple processes to **cooperate** with each other
- **local:** print daemon, X-Server
- **remote:** web/file/database server
  - client/server systems
- **abstractions/programming models**
  - **shared memory**
    - **several processes** use the **same memory** area, at the **same time**
    - requires additional **synchronisation**
  - **message exchange**
    - semantics of a fax (sending a **copy** of a message)
    - synchronous or asynchronous

# Processes: Interim Conclusion

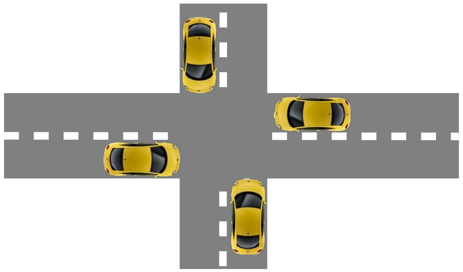
- **operating system abstractions**
  - process := program in execution
  - executed by a processor (e.g., CPU core)

> A *process* is a triple $(S, f, s)$, where $S$ is a state space, $f$ is an action function in that space, and $s$ is the subset of $S$ which defines the initial states of the process.* A process generates all the computations generated by its action function from its initial states. It is (*strictly*) *deterministic* if its action function is (strictly) deterministic. (Note that each computation generated by a strictly deterministic process is uniquely determined by its initial state.)† Similarly, processes are *temporal variants* if they differ only in having action functions which are temporal variants.

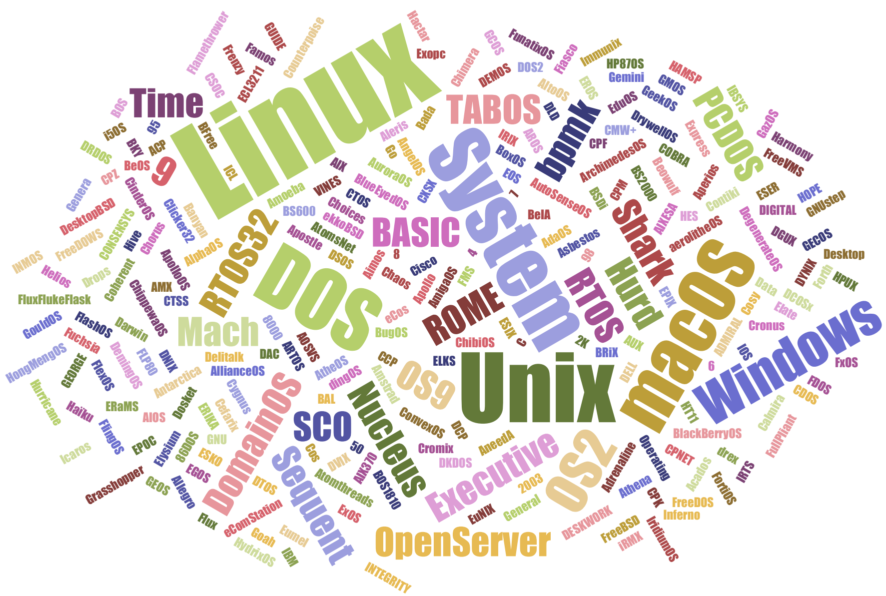
- **process states, scheduling**
  - operating systems keep track of process states
  - scheduling: allocation of CPU time, preemption




- **mode of operation**
  - cooperative and competing processes
  - concurrent execution, **mutual exclusion**



# Agenda

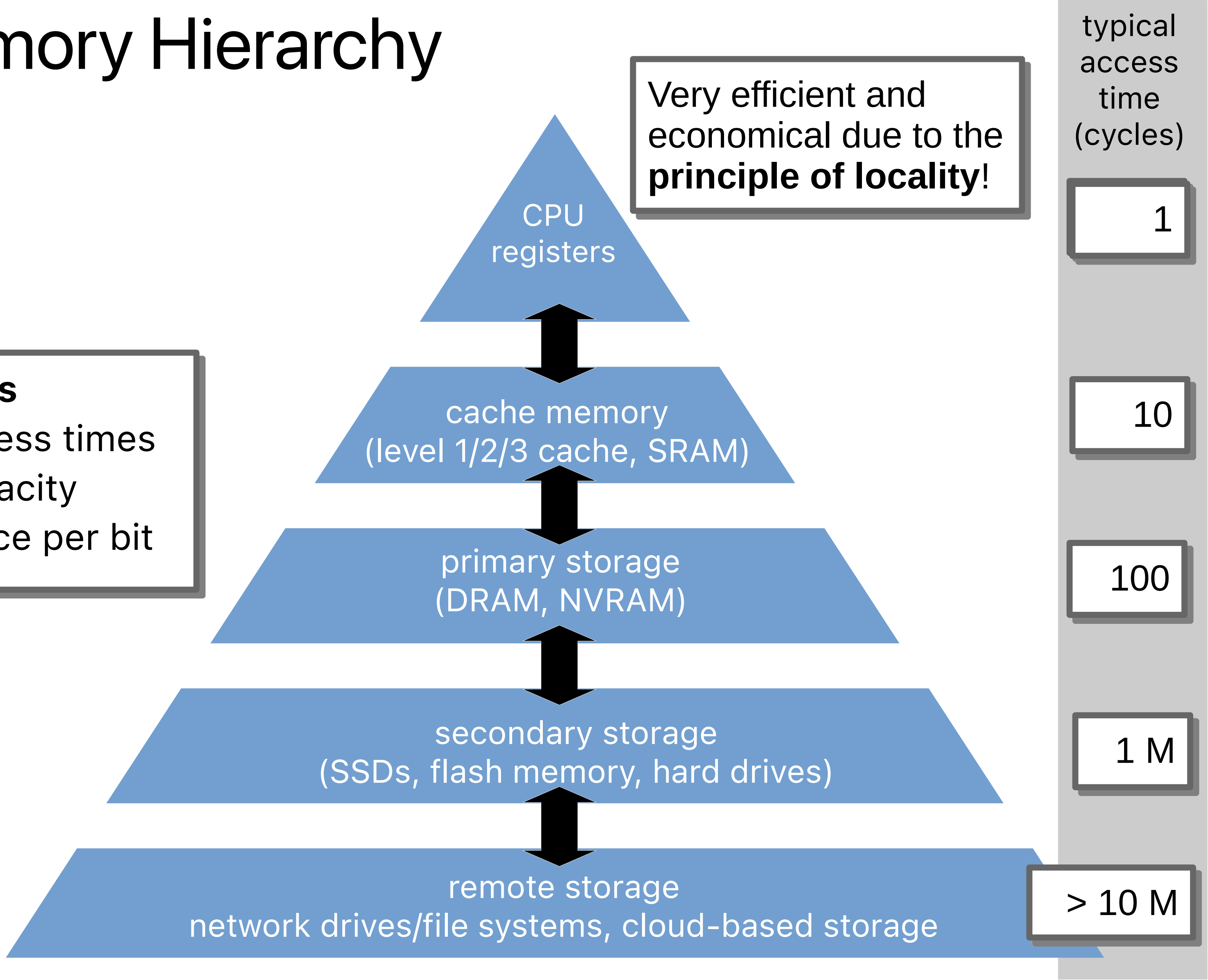


- Recap
- Organizational Matters
- Processes
  - CPU Scheduling
  - Synchronisation and Deadlocks
  - Inter-process Communication
- Memory Management
  - Primary Storage
  - Secondary Storage
- Multi- and Manycore Systems
- Summary and Outlook

# The Memory Hierarchy

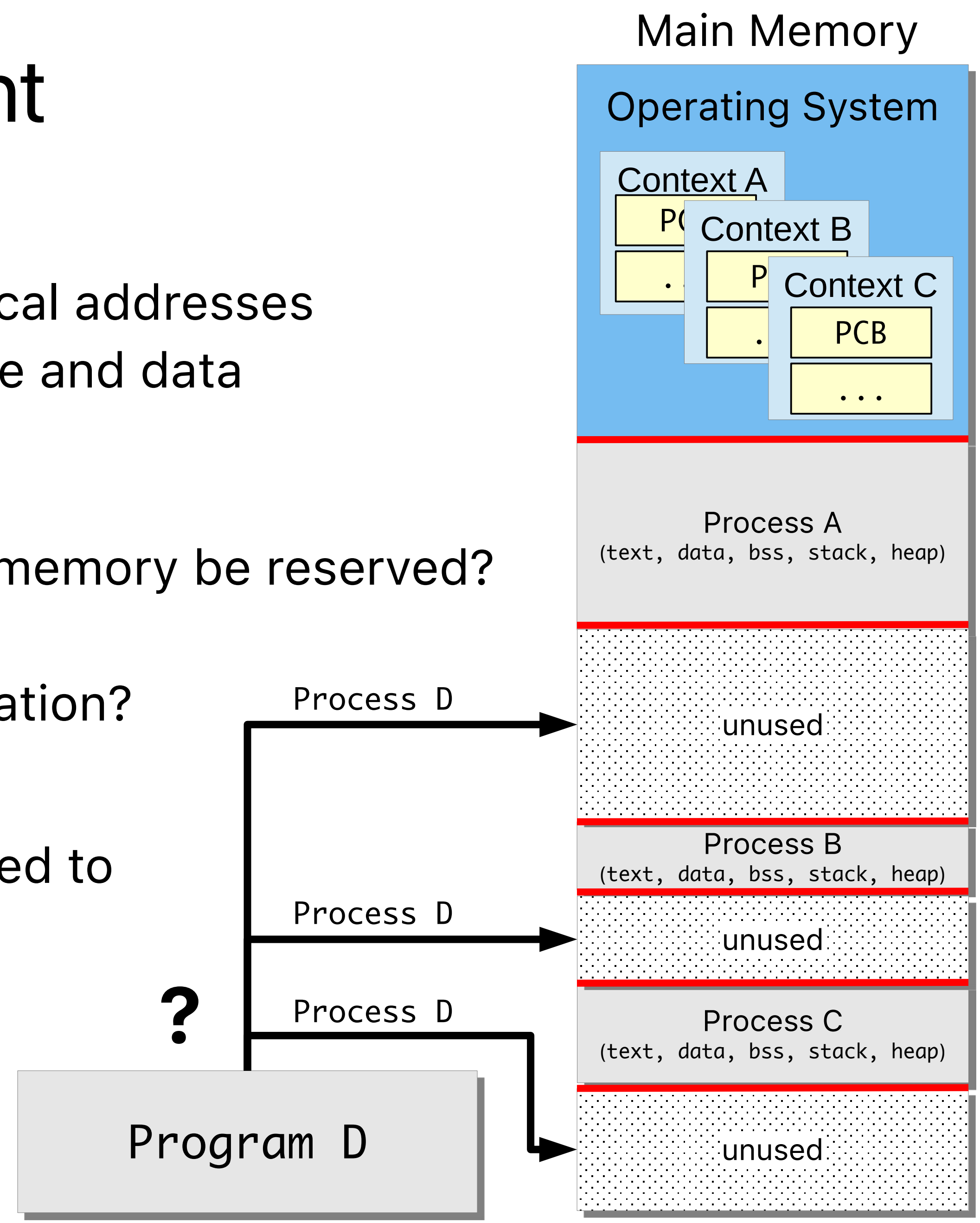
**storage properties**
- increasing access times
- increasing capacity
- decreasing price per bit

Very efficient and economical due to the **principle of locality**!

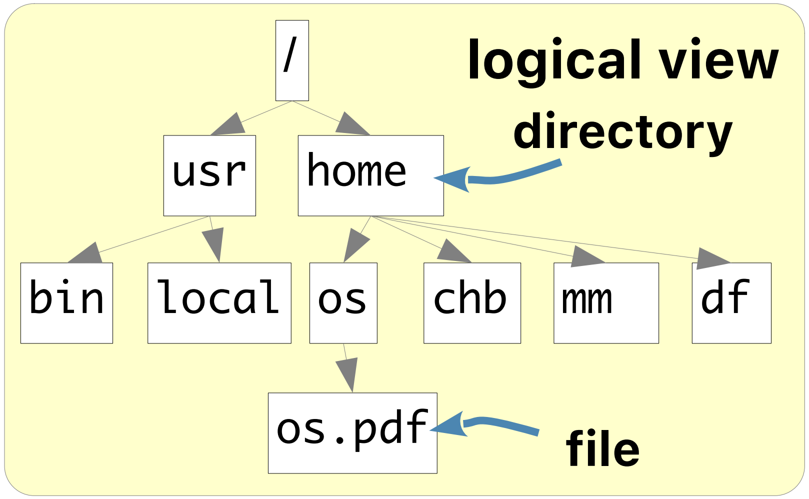


# Memory Management

- **address mapping**
  - logical addresses to physical addresses
  - enables **relocation** of code and data
- **placement strategies**
  - in **which location** should memory be reserved?
  - use **compactification?**
  - how to minimise fragmentation?
- **replacement strategies**
  - which memory area is suited to be swapped out?

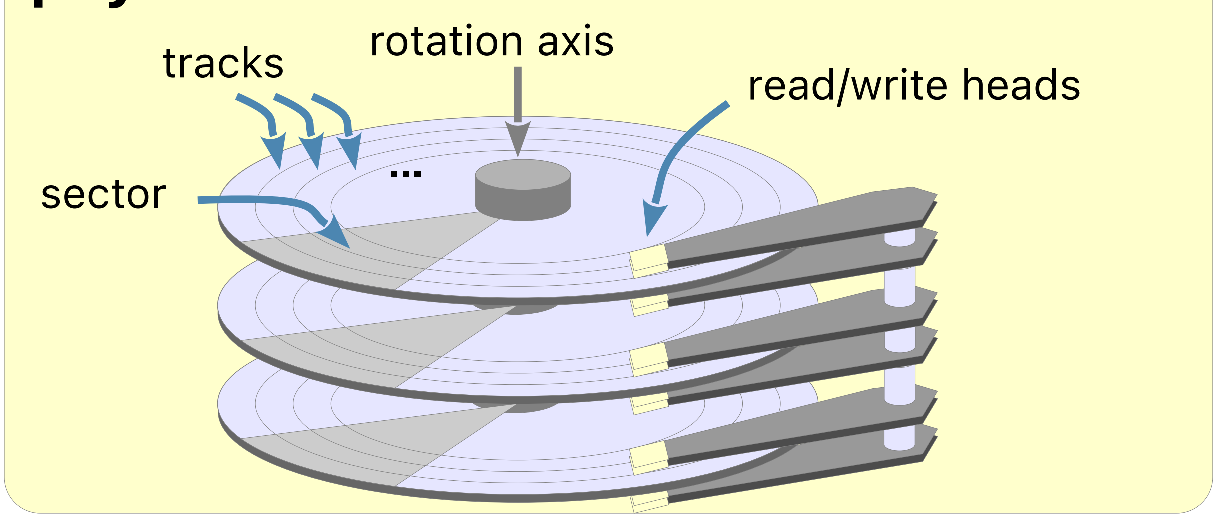


# Secondary Storage




**File systems** enable the permanent storage of large amounts of data.

mapping

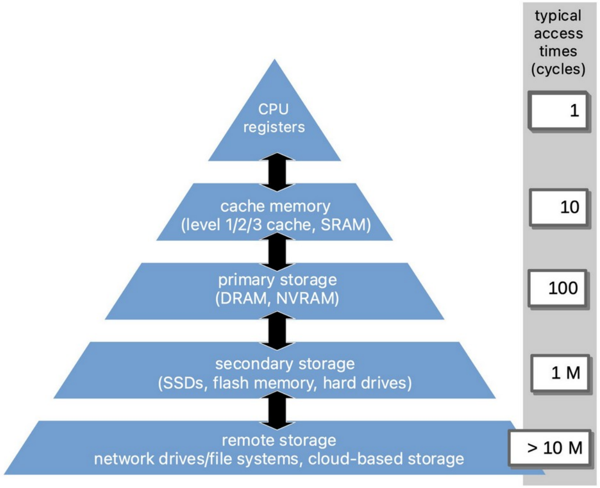



hard drive with a stack of 6 disks

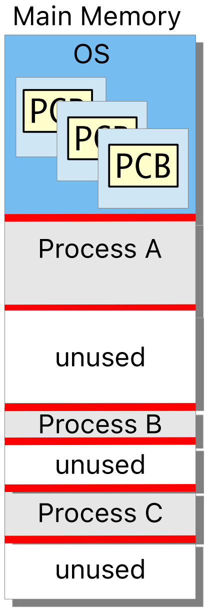
The operating system provides the **logical view** to the applications (processes) and must implement it efficiently.

# Memory Management: Interim Conclusion

- **primary and secondary storage**
  - memory hierarchy
  - operation: exploit **principle of locality**




- **manage memory efficiently**
  - address mapping: **logical → physical addresses**
  - {re,}placement strategies

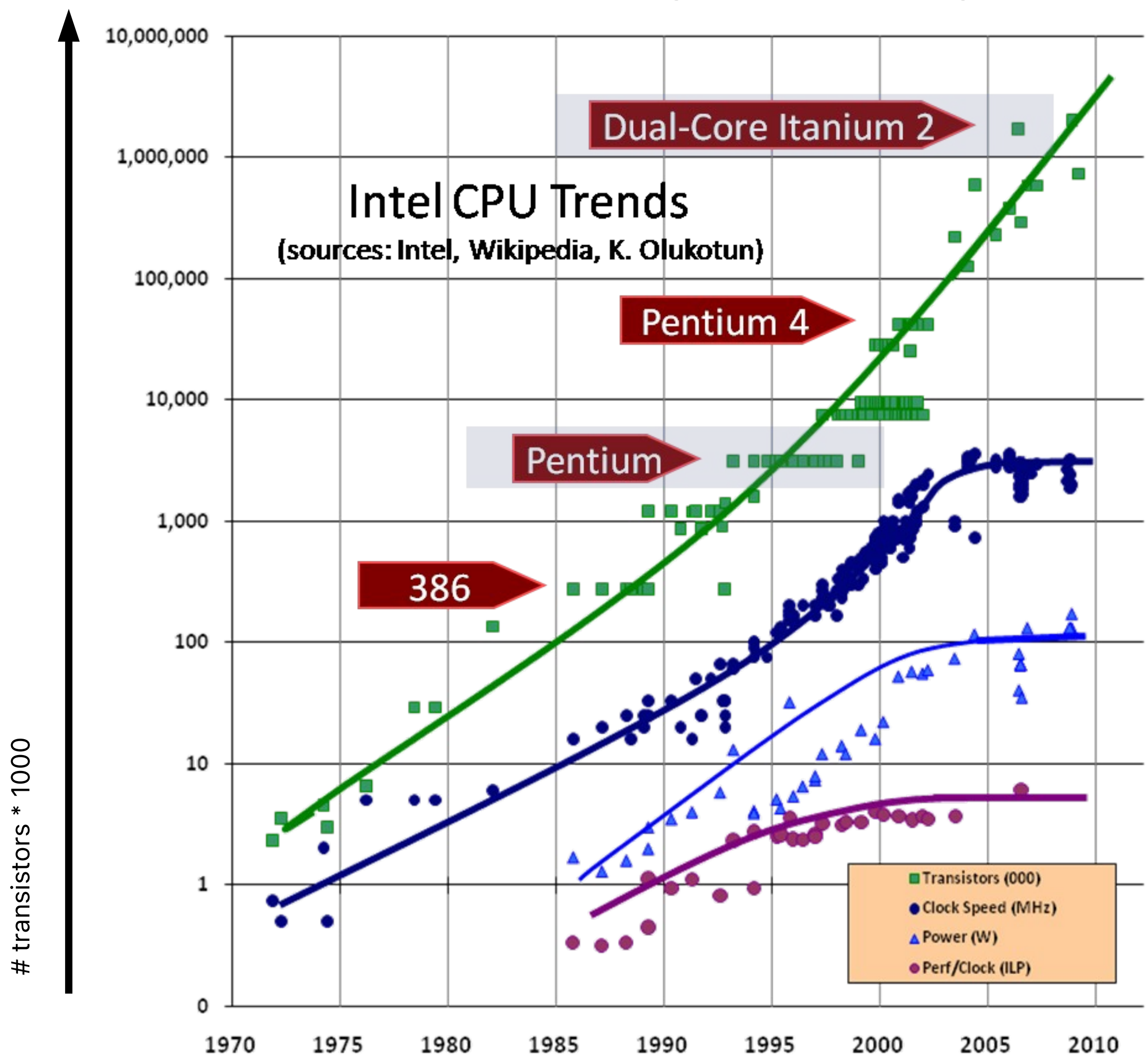



- **logical view**
  - file systems: operating system abstractions
  - implement efficient **abstraction layers** at OS level

# Agenda


- Recap
- Organizational Matters
- Processes
  - CPU Scheduling
  - Synchronisation and Deadlocks
  - Inter-process Communication
- Memory Management
  - Primary Storage
  - Secondary Storage
- **Multi- and Manycore Systems**
- Summary and Outlook

# Multi- and Manycore Systems




- # transistors
- clock speed
- power
- performance per clock cycle

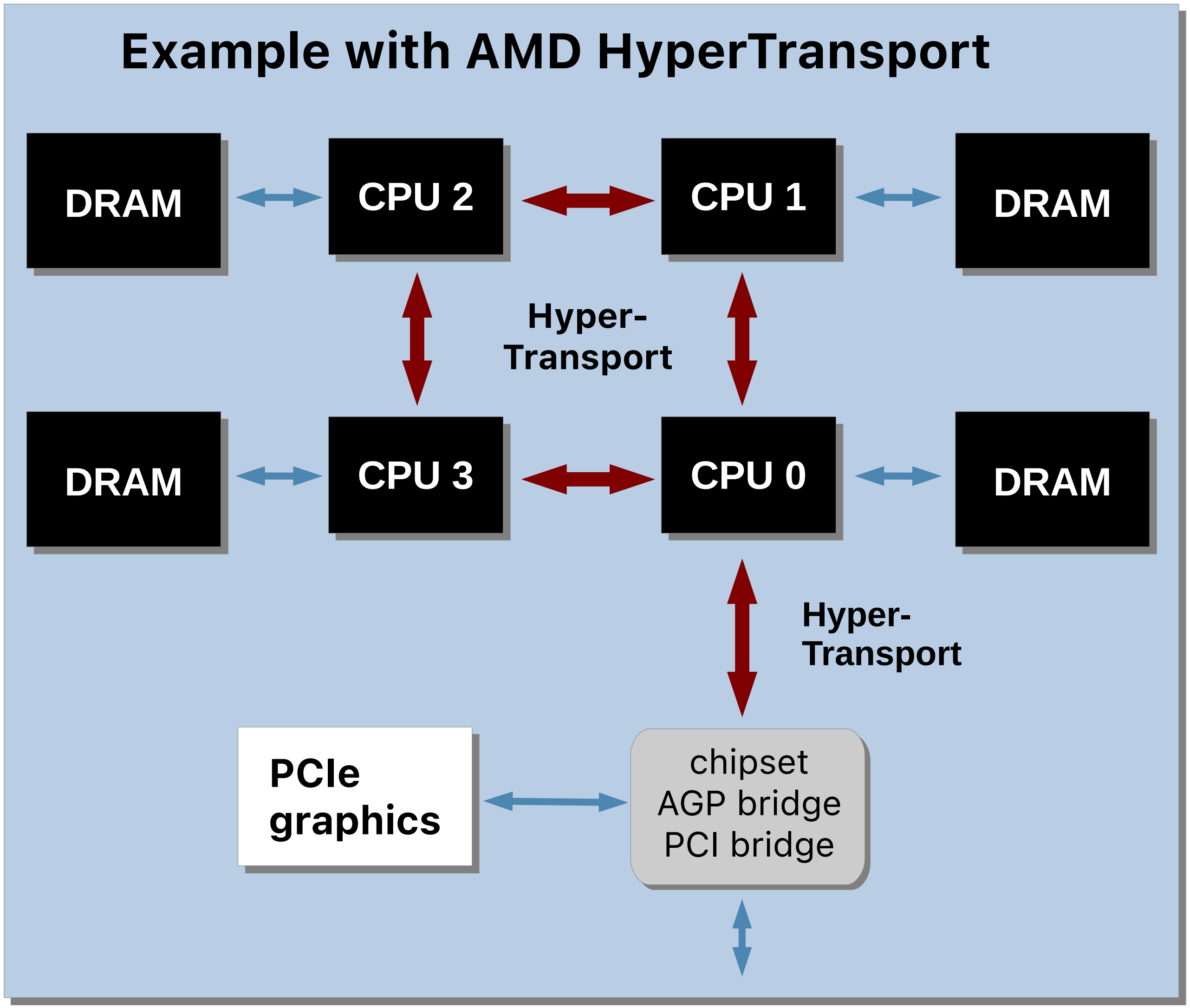
Sutter, H.:
*The Free Lunch Is Over*
`http://www.gotw.ca/publications/concurrency-ddj.htm`

# Non-Uniform Memory Architecture (NUMA)

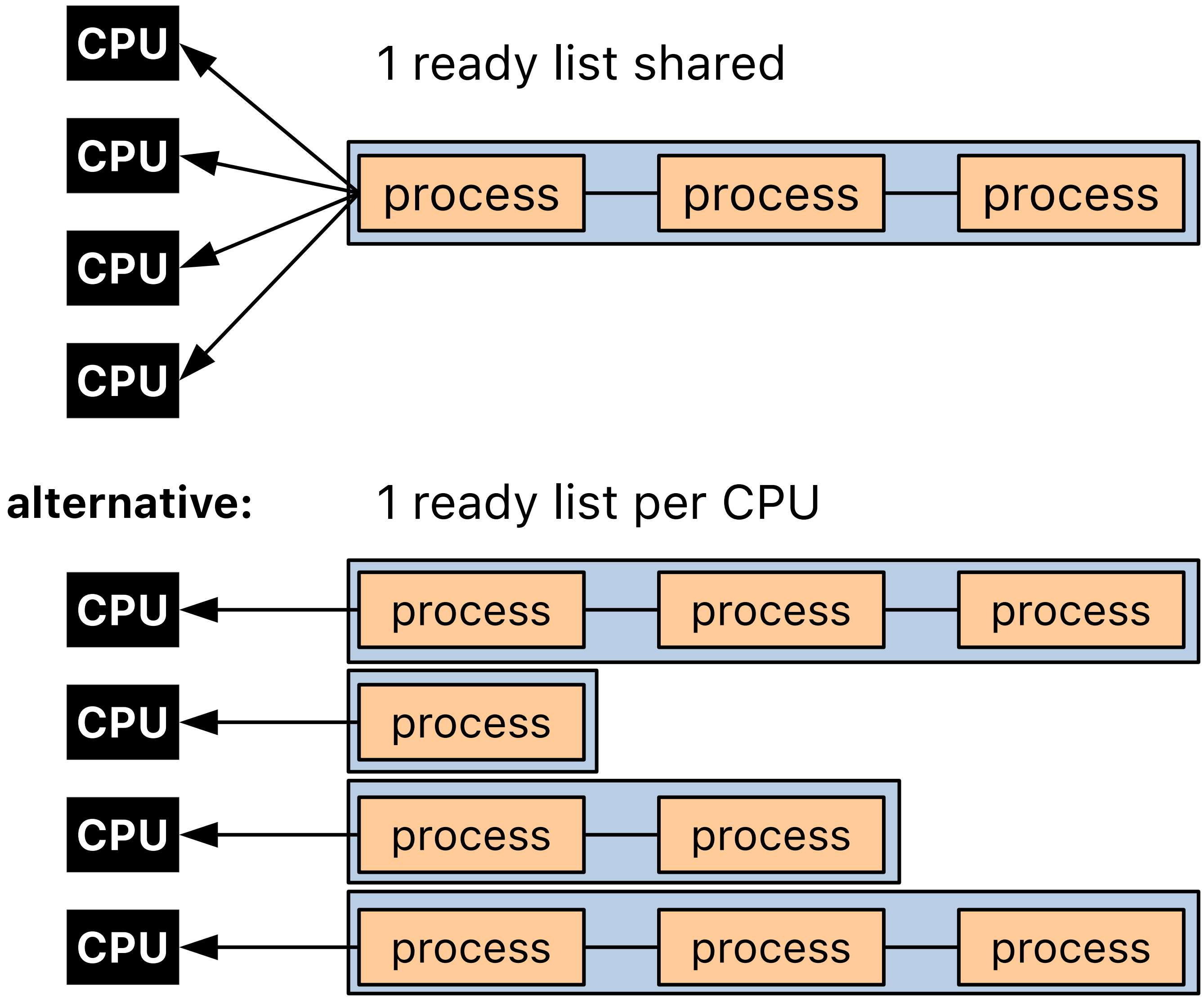
The CPUs (possibly with multiple cores) communicate with each other via HyperTransport.

Global address space: Main memory connected to other CPUs can be addressed, but the **latency is higher**.

Approach scales better, since parallel memory accesses are possible.



# CPU Scheduling on Multiprocessor Systems



# ⏭ Summary and Outlook

- **summary**
  - **OS resource management:** especially **CPU** and **memory**
  - **operating system abstractions**
    - **processes** → scheduling, synchronisation, inter-process communication
    - **files** → persistent input/output data, hierarchical data management
  - **memory** hierarchy
    - primary vs. secondary storage
    - **address mapping** (logical → physical)
- **outlook: processes and threads**
  - the **UNIX** process model
    - shells, I/O, UNIX process philosophy
    - process creation and states
  - **lightweight** process models
    - processes vs. threads
    - threads vs. user-level threads

# References and Acknowledgments

## Lecture

- Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

## Teaching Books and Reference Book

[1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons, *2018*.

[2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.

[3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.
https://www4.cs.fau.de/~wosch/glossar.pdf