Systems, Networks & Concurrency 2020

1

Introduction to Concurrency

Uwe R. Zimmer - The Australian National University

References for this chapter

[Ben-Ari06]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

Forms of concurrency

What is concurrency?

Working definitions:

- Literally 'concurrent' means:

  Adj.: Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint, associated [Oxfords English Dictionary]

Forms of concurrency

What is concurrency?

Working definitions:

- Literally 'concurrent' means:

  Adj.: Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint, associated [Oxfords English Dictionary]

- Technically 'concurrent' is usually defined negatively as:

  If there is no observer who can identify two events as being in strict temporal sequence (i.e. one event has fully terminated before the other one started) then these two events are considered concurrent.

Forms of concurrency

Why do we need/have concurrency?

- Physics, engineering, electronics, biology, …

☞ basically *every* real world system is **concurrent**!

- Sequential processing is suggested by most core computer architectures

  … *yet* (almost) *all* current processor architectures have **concurrent elements**

  … and *most* computer systems are part of a **concurrent network.**

- Strict sequential processing is suggested by widely used programming languages.

☞ Sequential programming delivers some *fundamental components* for concurrent programming

☞ *but we need to add a number of further crucial concepts*

Forms of concurrency

Why would a computer scientist consider concurrency?

☞ … to *be able* to connect computer systems with the **real world**

☞ … to *be able* to employ / design **concurrent parts of computer architectures**

☞ … to *construct* **complex software packages** (operating systems, compilers, databases, …)

☞ … to *understand* when sequential and/or concurrent programming is **required**

… or: to understand when sequential or concurrent programming can be **chosen freely**

☞ … to *enhance* the **reactivity** of a system

☞ … to enhance the **performance** of a system

☞ … to be able to design **embedded** systems

☞ …

Forms of concurrency

A computer scientist's view on concurrency

- Overlapped I/O and computation
  - ☞ Employ interrupt programming to handle I/O
- Multi-programming
  - ☞ Allow multiple independent programs to be executed on one CPU
- Multi-tasking
  - ☞ Allow multiple interacting processes to be executed on one CPU
- Multi-processor systems
  - ☞ Add physical/real concurrency
- Parallel Machines & distributed operating systems
  - ☞ Add (non-deterministic) communication channels
- General network architectures
  - ☞ Allow for any form of communicating, distributed entities

Forms of concurrency

A computer scientist's view on concurrency

Terminology for physically concurrent machines architectures:

- SISD
  [singe instruction, single data]
  ☞ Sequential processors
- SIMD
  [singe instruction, multiple data]
  ☞ Vector processors
- MISD
  [multiple instruction, single data]
  ☞ Pipelined processors
- MIMD
  [multiple instruction, multiple data]
  ☞ Multi-processors or computer networks

Forms of concurrency

An engineer's view on concurrency

☞ Multiple **physical, coupled, dynamical systems** form the actual environment and/or task at hand

☞ In order to model and control such a system, its **inherent concurrency** needs to be considered

☞ **Multiple less powerful processors** are often preferred over a single high-performance cpu

☞ The system design of usually strictly **based on the structure of the given physical system**.

Forms of concurrency

Does concurrency lead to chaos?

Concurrency often leads to the following features / issues / problems:

- **non-deterministic** phenomena
- **non-observable** system states
- results may depend on more than just the input parameters and states at start time (timing, throughput, load, available resources, signals … *throughout* the execution)
- **non-reproducible** ☞ debugging?

Forms of concurrency

Does concurrency lead to chaos?

Concurrency often leads to the following features / issues / problems:

- **non-deterministic** phenomena
- **non-observable** system states
- results may depend on more than just the input parameters and states at start time (timing, throughput, load, available resources, signals … *throughout* the execution)
- **non-reproducible** ☞ debugging?

Meaningful employment of concurrent systems features:

- non-determinism employed where the **underlying system is non-deterministic**
- non-determinism employed where the **actual execution sequence is meaningless**
- **synchronization** employed where adequate … but only there

☞ Control & monitor where required (and do it right), but not more …

Concurrency on different abstraction levels/perspectives

☞ **Networks**

- Large scale, high bandwidth interconnected nodes (“supercomputers”)
- Networked computing nodes
- Standalone computing nodes – including local buses & interfaces sub-systems
- Operating systems (& distributed operating systems)

☞ **Implicit concurrency**

☞ **Explicit concurrent programming (message passing and synchronization)**

☞ **Assembler level concurrent programming**

- Individual concurrent units inside one CPU
- Individual electronic circuits
- …

The concurrent programming abstraction

1. What appears sequential on a higher abstraction level, is usually concurrent at a lower abstraction level:

   ☞ e.g. Concurrent operating system or hardware components, which might not be visible at a higher programming level

2. What appears concurrent on a higher abstraction level, might be sequential at a lower abstraction level:

   ☞ e.g. Multi-processing system, which are executed on a single, sequential computing node

Models and Terminology

The concurrent programming abstraction

- '*concurrent*' is technically defined negatively as:

  If there is no observer who can identify two events as being in strict temporal sequence (i.e. one event has fully terminated before the other one starts up), then these two events are considered *concurrent*.

- '*concurrent*' in the context of programming and logic:

  "*Concurrent programming abstraction* is the study of interleaved execution sequences of the atomic instructions of sequential processes."
  (Ben-Ari)

Models and Terminology

The concurrent programming abstraction

Concurrent program ::=
Multiple sequential programs (processes or threads)
which are executed *concurrently*.

P.S. it is generally assumed that concurrent execution means that there is one execution unit (processor) per sequential program

- even though this is usually not technically correct, it is still an often valid, conservative assumption in the context of concurrent programming.

Models and Terminology

The concurrent programming abstraction

☞ No interaction between concurrent system parts means that we can analyze them individually as pure sequential programs [end of course].

Models and Terminology

The concurrent programming abstraction

☞ No interaction between concurrent system parts means that we can analyze them individually as pure sequential programs [end of course].

☞ **Interaction** occurs in form of:

- **Contention** (implicit interaction):

  Multiple concurrent execution units compete for one shared resource.

- **Communication** (explicit interaction):

  Explicit passing of information and/or explicit synchronization.

Models and Terminology

The concurrent programming abstraction

Time-line or Sequence?

Consider time (durations) explicitly:

☞ Real-time systems ☞ join the appropriate courses

Consider the sequence of interaction points only:

☞ Non-real-time systems ☞ stay in your seat

The concurrent programming abstraction

Correctness of concurrent non-real-time systems [logical correctness]:

- does *not* depend on clock speeds / execution times / delays
- does *not* depend on actual interleaving of concurrent processes

☞ holds true for **all possible sequences of interaction points (interleavings)**

Models and Terminology

The concurrent programming abstraction

Correctness vs. testing in concurrent systems:

Slight changes in external triggers may (and usually does) result in completely different schedules (interleaving):

☞ Concurrent programs which depend in any way on external influences cannot be tested without modelling and embedding those influences into the test process.

☞ Designs which are provably correct with respect to the specification and are **independent** of the *actual timing behavior* are essential.

P.S. some timing restrictions for the scheduling still persist in non-real-time systems, e.g. ‘fairness’

Models and Terminology

The concurrent programming abstraction

Atomic operations:

Correctness proofs / designs in concurrent systems rely on the assumptions of

‘Atomic operations’ [detailed discussion later]:

- Complex and powerful atomic operations ease the correctness proofs, but may limit flexibility in the design
- Simple atomic operations are theoretically sufficient, but may lead to complex systems which correctness cannot be proven in practice.

The concurrent programming abstraction

Standard concepts of correctness:

- **Partial correctness**:

$$(P(I) \wedge terminates(Program(I,O))) \Rightarrow Q(I,O)$$

- **Total correctness**:

$$P(I) \Rightarrow (terminates(Program(I,O)) \wedge Q(I,O))$$

where I, O are input and output sets,
P is a property on the input set,
and Q is a relation between input and output sets

☞ do these concepts apply to and are sufficient for concurrent systems?

Models and Terminology

The concurrent programming abstraction

Extended concepts of correctness in concurrent systems:

¬ Termination is often not intended or even considered a failure

Safety properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Box Q(I,S)$$

where $\Box Q$ means that Q does *always* hold

Liveness properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)
and S is the current state of the concurrent system

Models and Terminology

The concurrent programming abstraction

Safety properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Box Q(I,S)$$

where $\Box Q$ means that Q does *always* hold

Examples:

- Mutual exclusion (no resource collisions)
- Absence of deadlocks
  (and other forms of 'silent death' and 'freeze' conditions)
- Specified responsiveness or free capabilities
  (typical in real-time / embedded systems or server applications)

The concurrent programming abstraction

Liveness properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)
and S is the current state of the concurrent system

Examples:

- Requests need to complete eventually
- The state of the system needs to be displayed eventually
- No part of the system is to be delayed forever (fairness)

☞ Interesting *liveness* properties can be very hard to prove

Introduction to processes and threads

1 CPU per control-flow

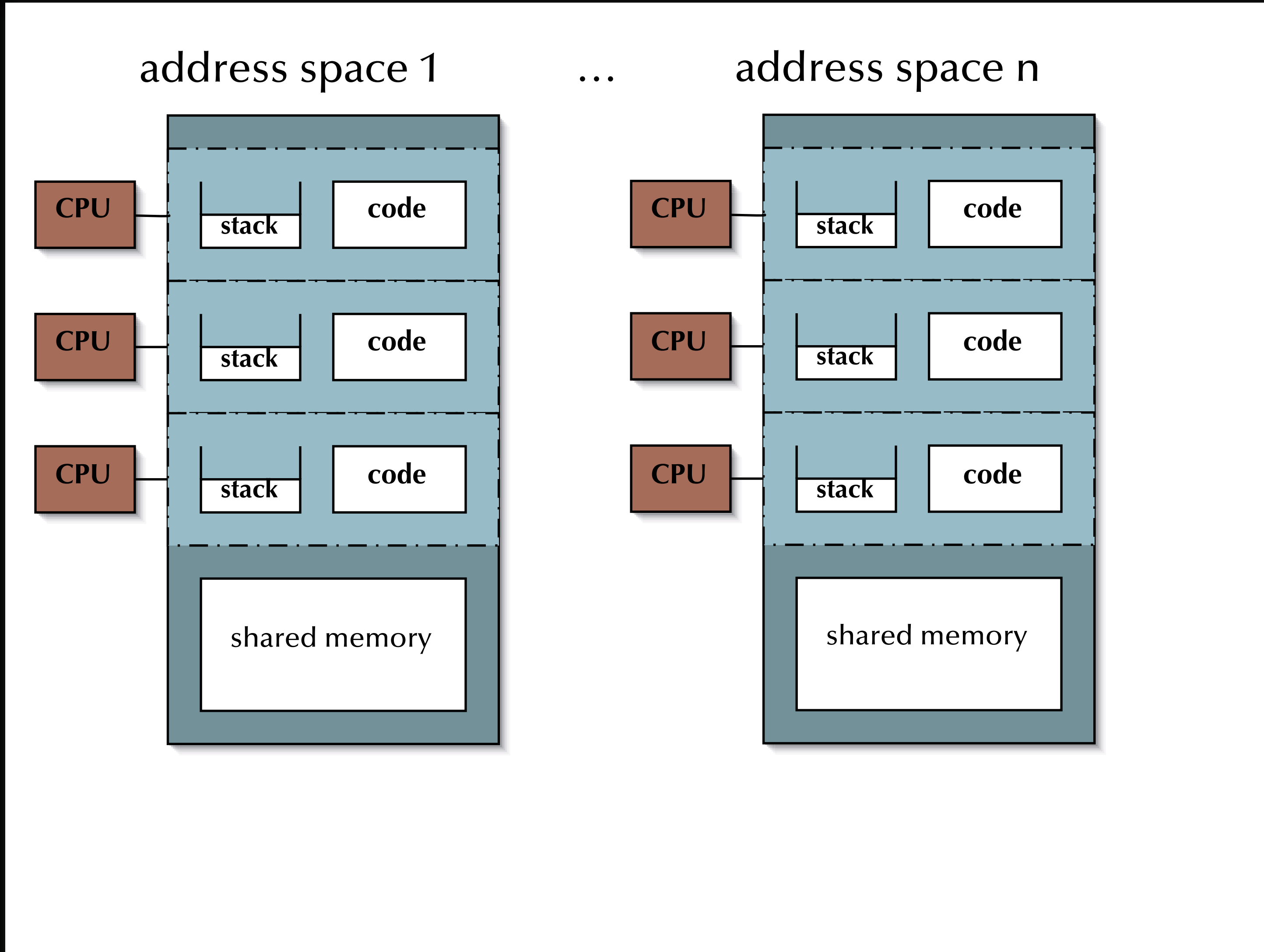
Specific configurations only, e.g.:

- Distributed µcontrollers.
- Physical process control systems:

  1 cpu per task, connected via a bus-system.

☞ **Process management** (scheduling) not required.

☞ **Shared memory access** need to be coordinated.



Introduction to processes and threads

1 CPU for all control-flows

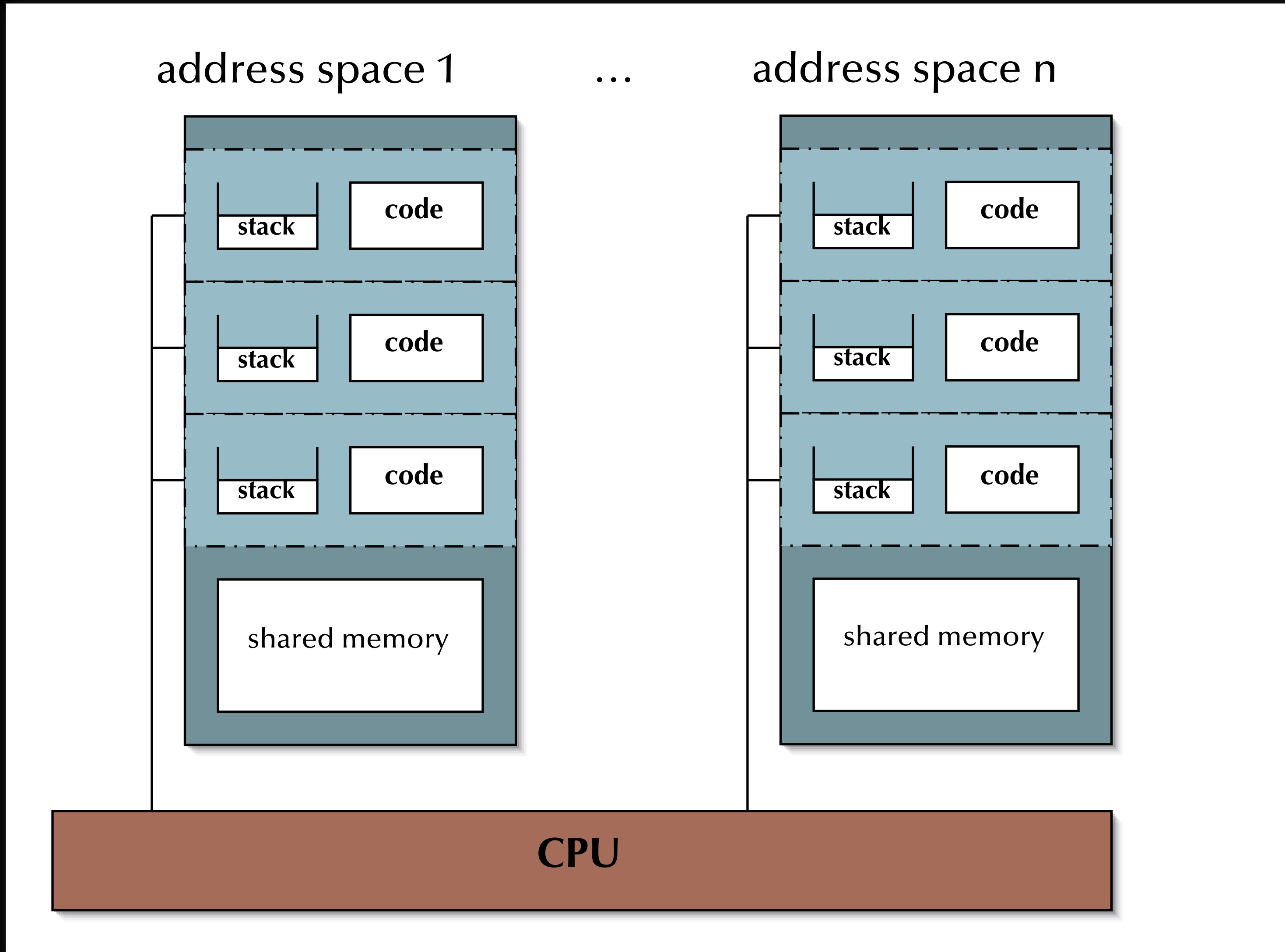
- OS: emulate one CPU for every control-flow:

  Multi-tasking operating system

☞ Support for **memory protection** essential.

☞ **Process management** (scheduling) required.

☞ **Shared memory access** need to be coordinated.



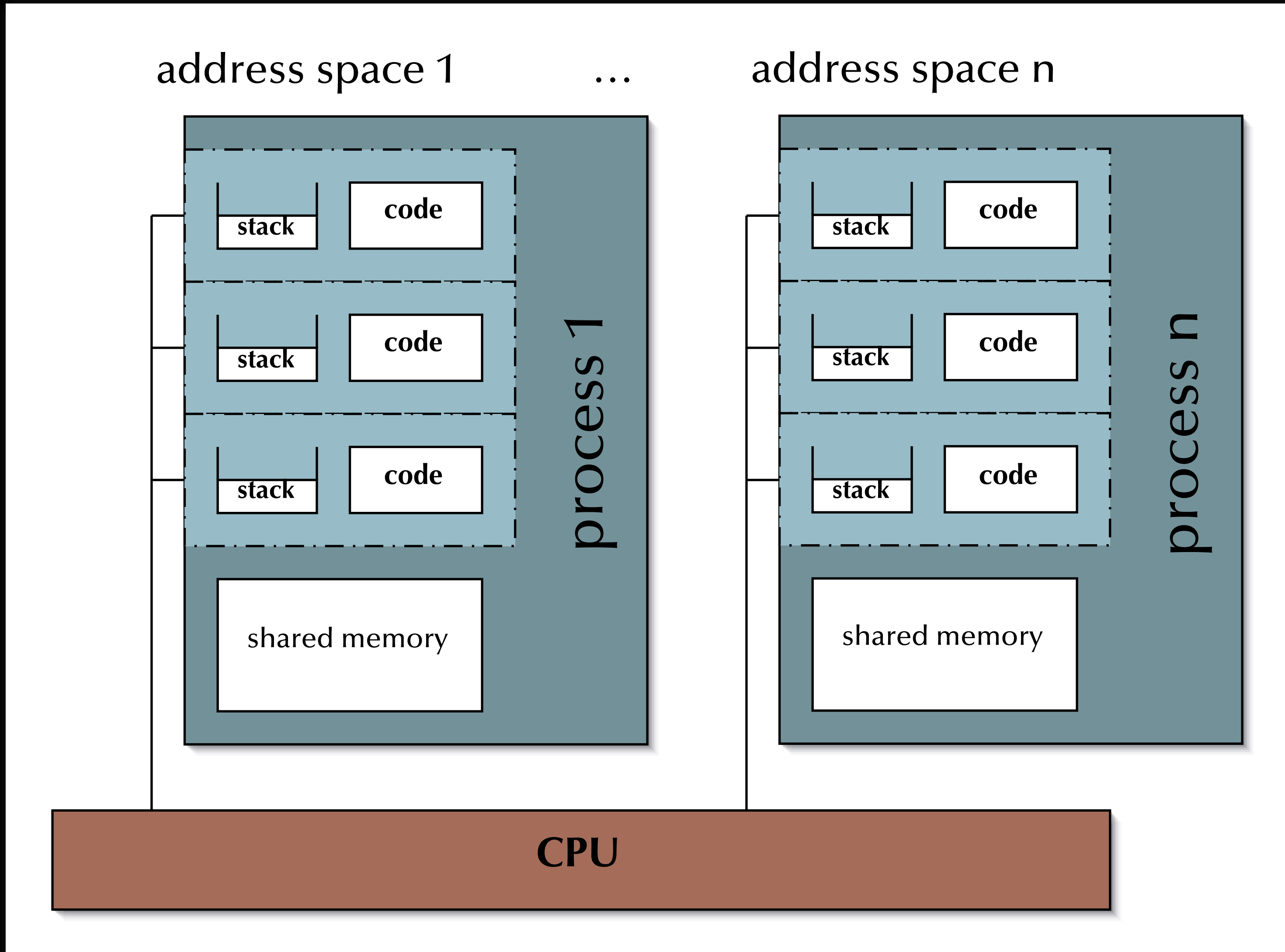
Introduction to processes and threads

Processes

Process ::=

Address space
+ Control flow(s)

☞ Kernel has full knowledge about all processes as well as their **states**, **requirements** and currently held **resources**.

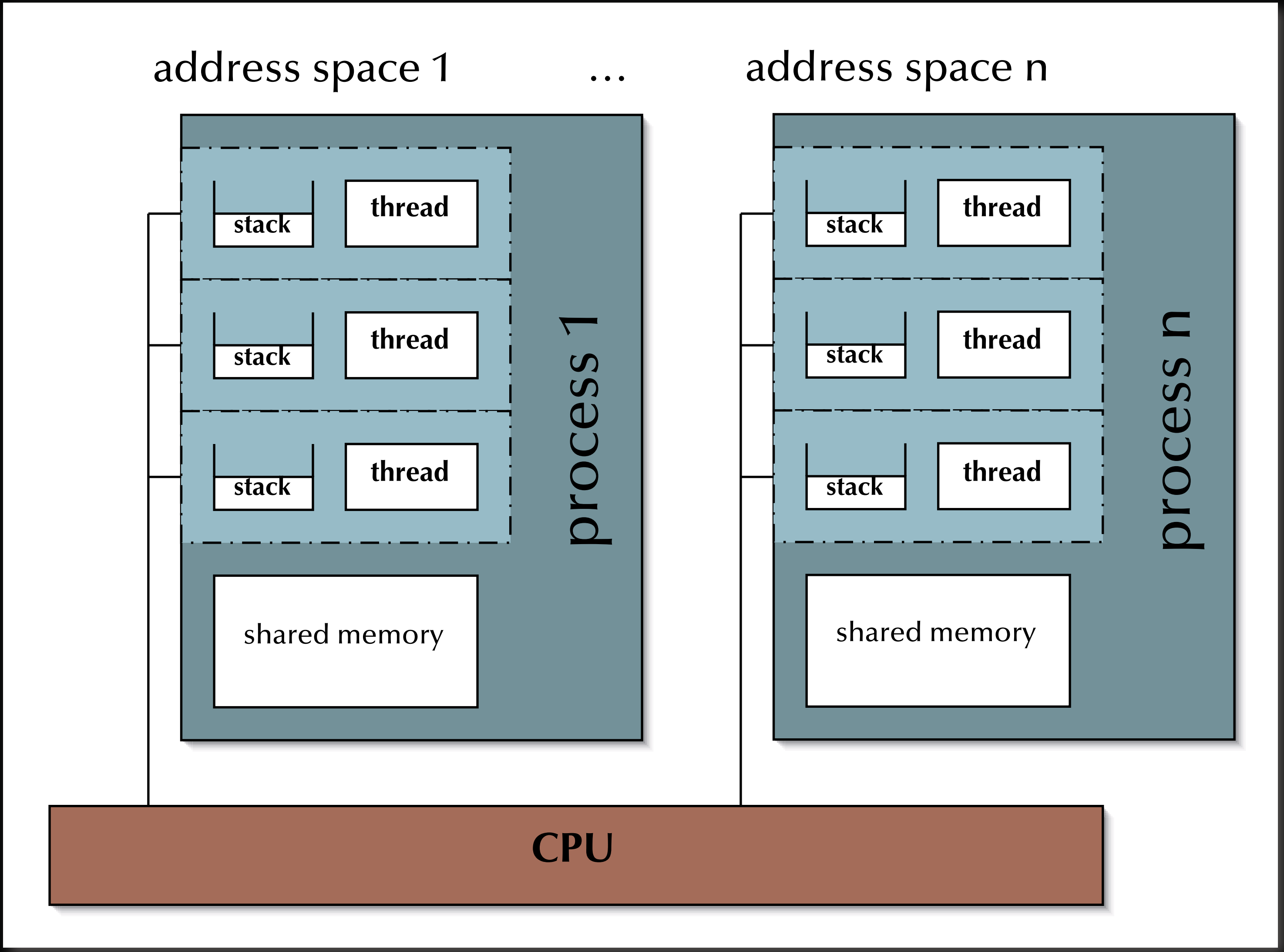


Introduction to processes and threads

Threads

Threads (individual control-flows) can be handled:

- *Inside* the OS:
  - ☞ Kernel scheduling.
  - Thread can easily be connected to external events (I/O).
- *Outside* the OS:
  - ☞ User-level scheduling.
  - Threads may need to go through their parent process to access I/O.

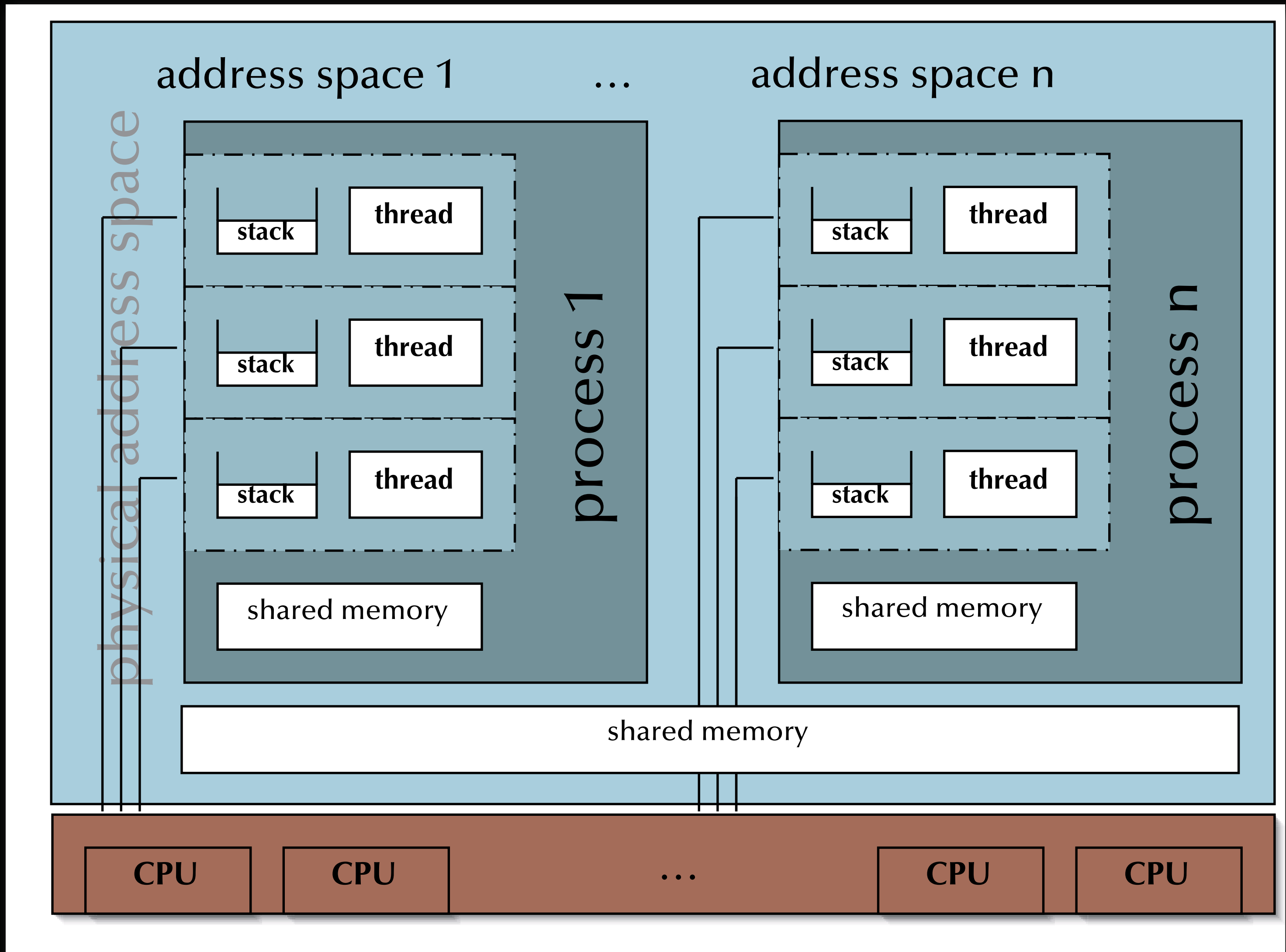


Introduction to processes and threads

Symmetric Multiprocessing (SMP)

All CPUs share the same physical address space (and access to resources).

☞ Any process / thread can be executed on any available CPU.



Processes ↔ Threads

Also processes can share memory and the specific definition of threads is different in different operating systems and contexts:

☞ Threads can be regarded as a group of processes, which share some resources (☞ process-hierarchy).

☞ Due to the overlap in resources, the attributes attached to threads are less than for 'first-class-citizen-processes'.

☞ Thread switching and inter-thread communication can be more efficient than switching on process level.

☞ Scheduling of threads depends on the actual thread implementations:

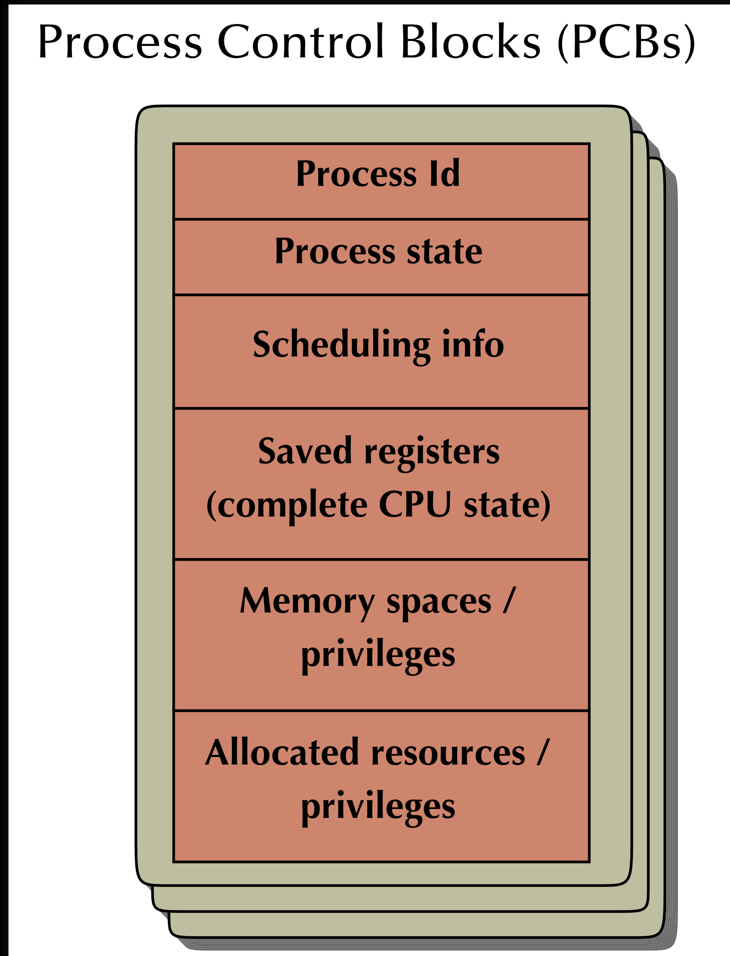
- e.g. *user-level control-flows*, which the kernel has no knowledge about at all.
- e.g. *kernel-level control-flows*, which are handled as processes with some restrictions.

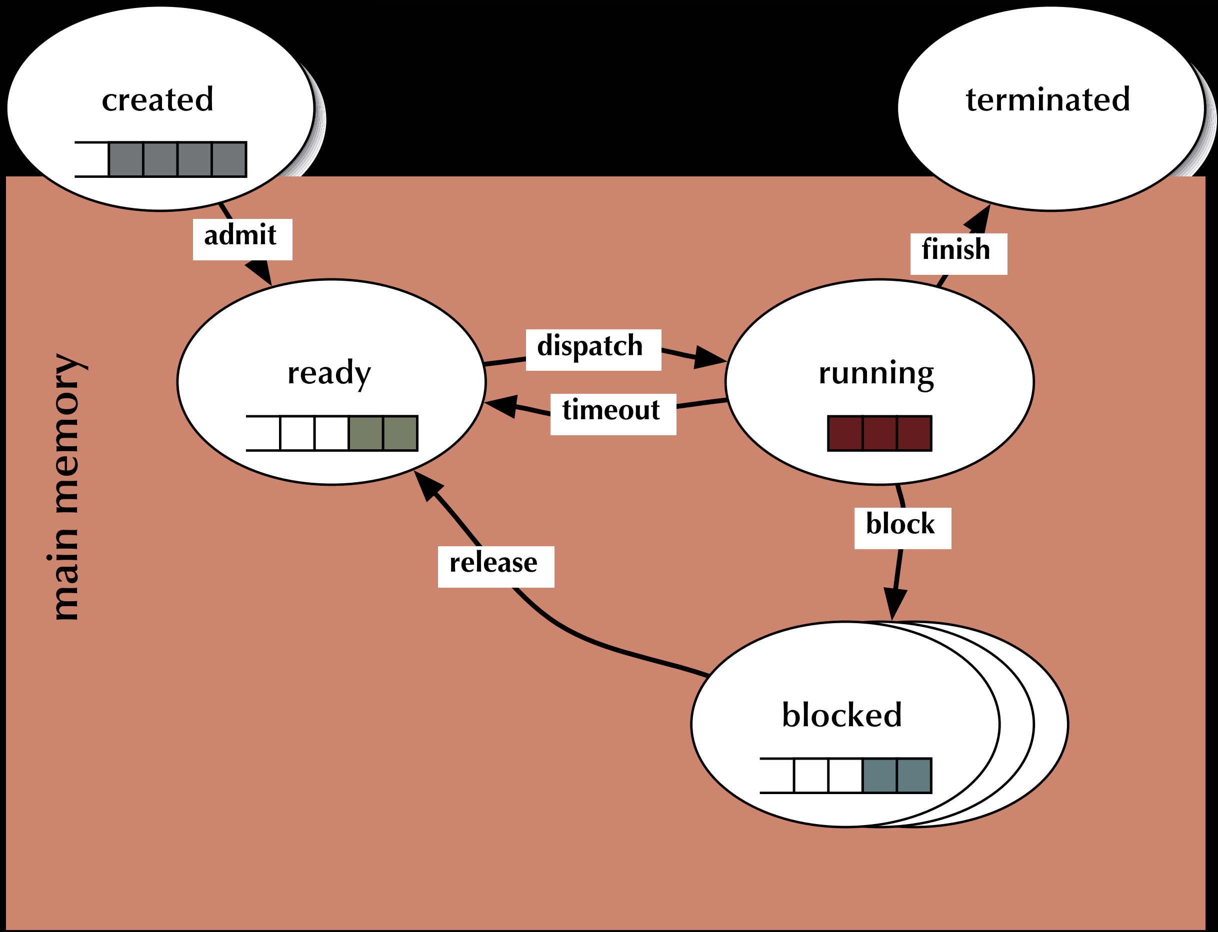
Introduction to processes and threads

Process Control Blocks

- **Process Id**
- **Process state**:
  {created, ready, executing, blocked, suspended, bored …}
- **Scheduling attributes**:
  Priorities, deadlines, consumed CPU-time, …
- **CPU state**: Saved/restored information while context switches (incl. the program counter, stack pointer, …)
- **Memory attributes / privileges**:
  Memory base, limits, shared areas, …
- **Allocated resources / privileges**:
  Open and requested devices and files, …

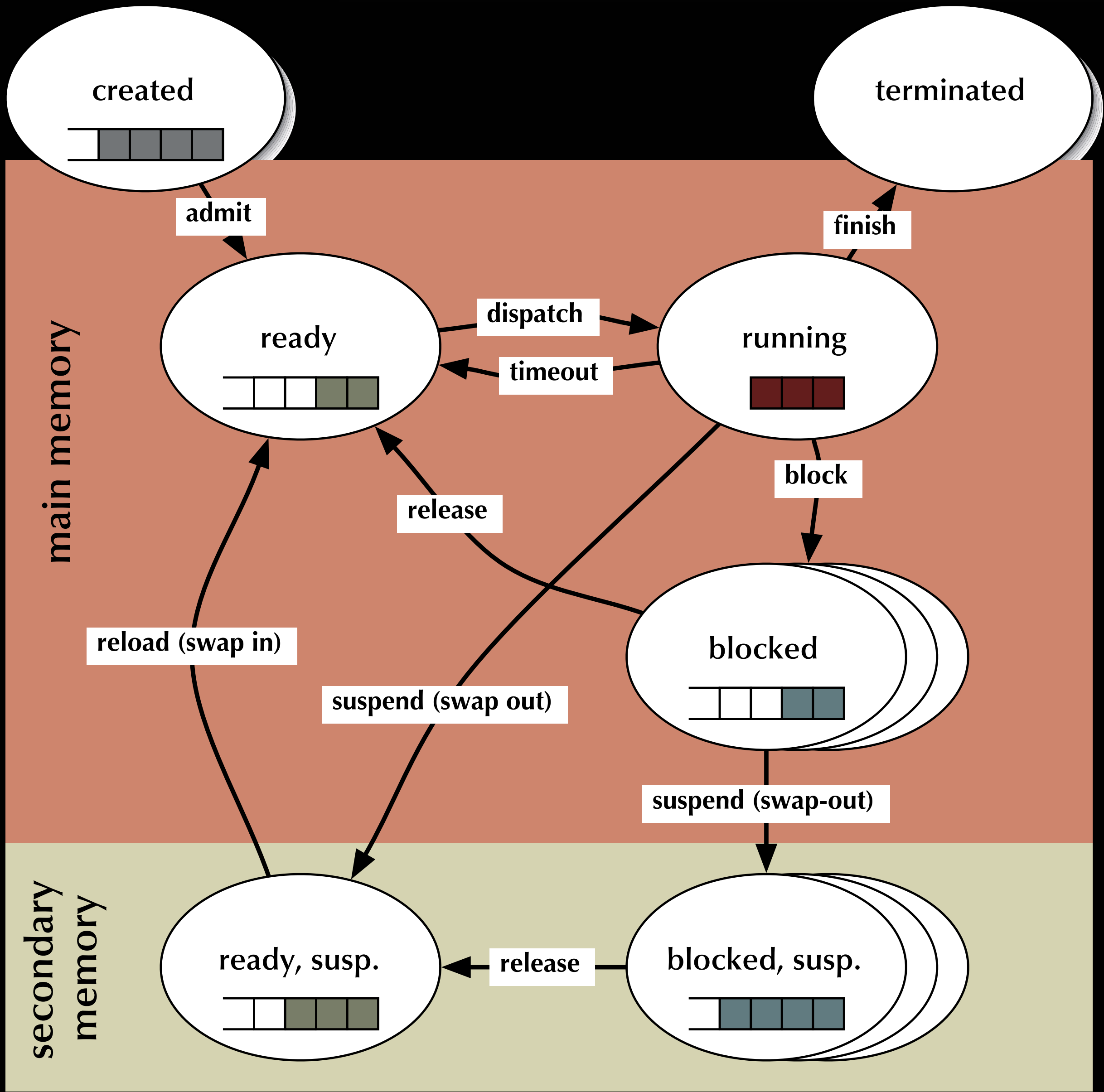
… PCBs (links thereof) are commonly enqueued at a certain state or condition (awaiting access or change in state)





Process states

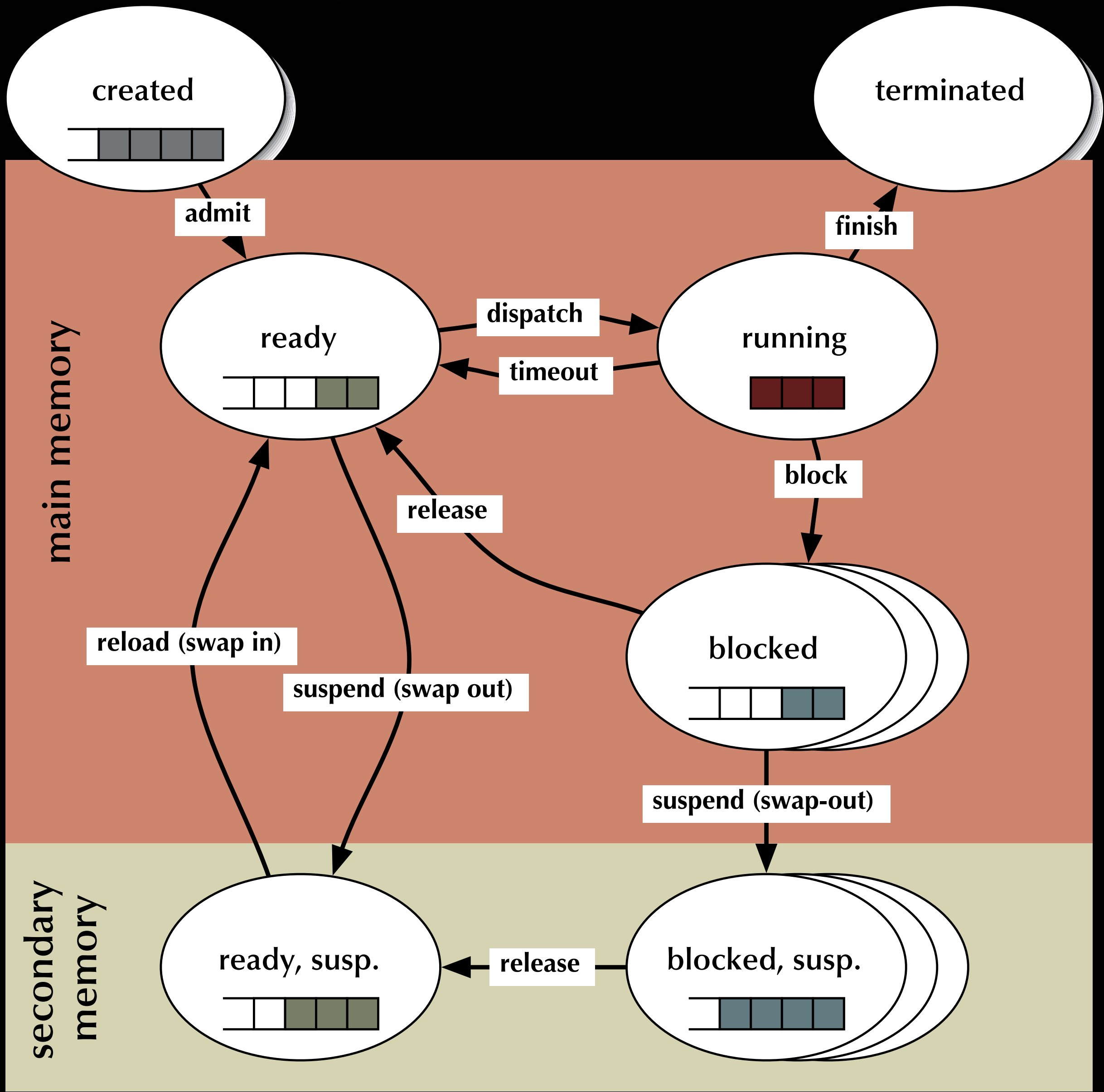
- **created**: the task is ready to run, but not yet considered by any dispatcher
  ☞ waiting for admission
- **ready**: ready to run
  ☞ waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
  ☞ waiting for a resource



Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
  ☞ waiting for admission
- **ready**: ready to run
  ☞ waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
  ☞ waiting for a resource
- **suspended** states: swapped out of main memory (none time critical processes)
  ☞ waiting for main memory space (and other resources)

Introduction to Concurrency

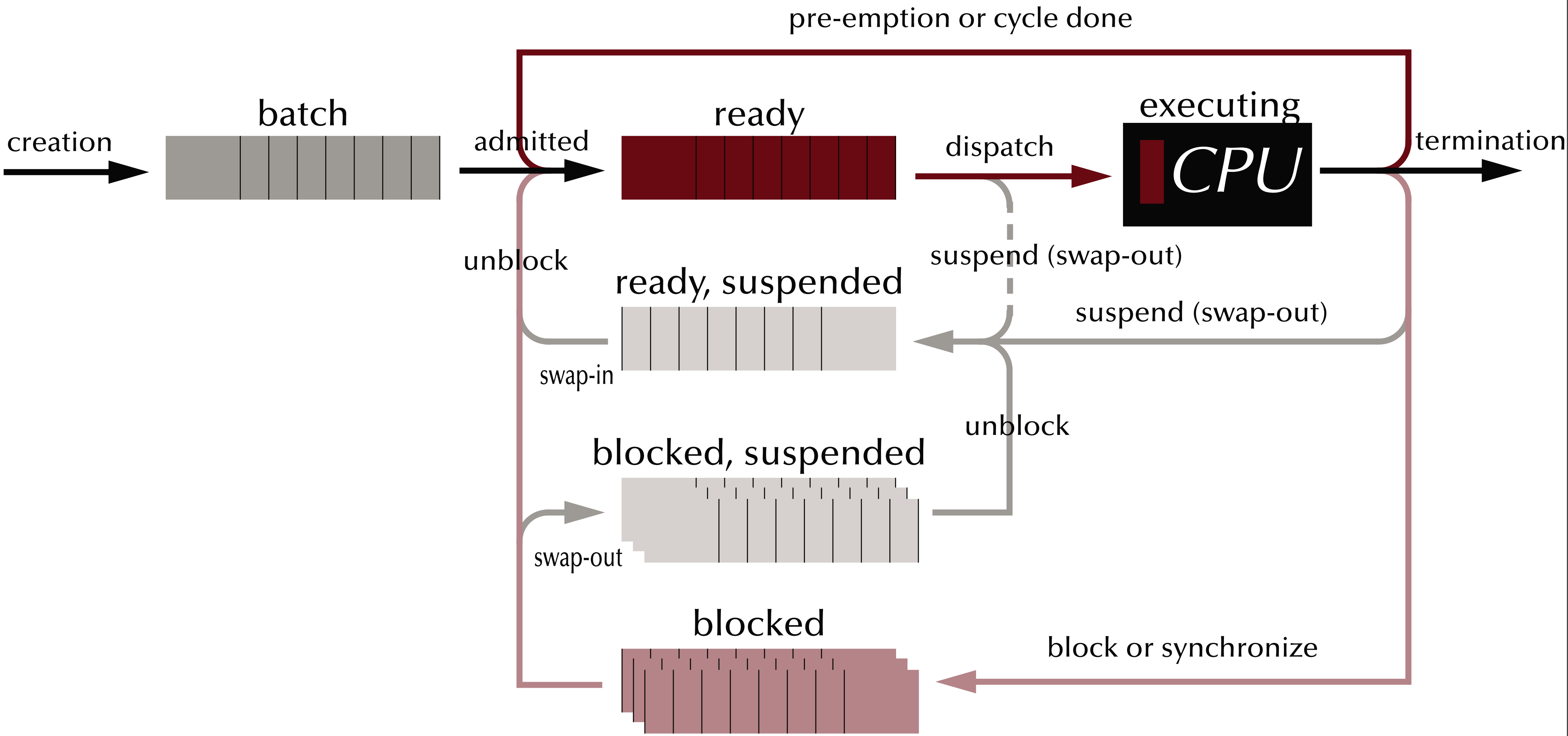


Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
  ☞ waiting for admission
- **ready**: ready to run
  ☞ waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
  ☞ waiting for a resource
- **suspended** states: swapped out of main memory
  (none time critical processes)
  ☞ waiting for main memory space (and other resources)

☞ dispatching and suspending can now be independent modules

Process states



UNIX processes

In UNIX systems tasks are created by 'cloning'

```
pid = fork ();
```

resulting in a *duplication* of the *current* process

… returning **'0'** to the newly created process (the 'child' process)

… returning the **process id** of the child process to the creating process (the 'parent' process)

… or returning **'-1'** as C-style indication of a failure (in void of actual exception handling)

Frequent usage:

```
if (fork () == 0) {
… the child's task …
… often implemented as: exec ("absolute path to executable file", "args");
exit (0); /* terminate child process */
} else {
… the parent's task …
pid = wait (); /* wait for the termination of one child process */
}
```

UNIX processes

Communication between UNIX tasks ('pipes')

```
int data_pipe [2], c, rc;
if (pipe (data_pipe) == -1) {
 perror ("no pipe"); exit (1);
}
if (fork () == 0) {
 close (data_pipe [1]);
 while ((rc = read
  (data_pipe [0], &c, 1)) > 0) {
   putchar (c);
 }
 if (rc == -1) {
  perror ("pipe broken");
  close (data_pipe [0]);
  exit (1);
 }
 close (data_pipe [0]); exit (0);
} else {
 close (data_pipe [0]);
 while ((c = getchar ()) > 0) {
  if (write(data_pipe[1], &c, 1)== -1) {
   perror ("pipe broken");
   close (data_pipe [1]);
   exit (1);
  };
 }
 close (data_pipe [1]);
 pid = wait ();
}
```

Concurrent programming languages

Requirement

- Concept of **tasks**, **threads** or other **potentially concurrent entities**

Frequently requested essential elements

- Support for **management** or concurrent entities (create, terminate, …)
- Support for **contention management** (mutual exclusion, …)
- Support for **synchronization** (semaphores, monitors, …)
- Support for **communication** (message passing, shared memory, rpc …)
- Support for **protection** (tasks, memory, devices, …)

Concurrent programming languages

Language candidates

☞ **Explicit concurrency**

- Ada, C++, Rust
- Chill
- Erlang
- Go
- Chapel, X10
- Occam, CSP
- All .net languages
- Java, Scala, Clojure
- Algol 68, Modula-2, Modula-3
- …

☞ **Implicit (potential) concurrency**

- Lisp, Haskell, Caml, Miranda, and any other functional language
- Smalltalk, Squeak
- Prolog
- Esterel, Lustre, Signal

☞ **Wannabe concurrency**

- Ruby, Python [mostly broken due to global interpreter locks]

☞ No support:

- Eiffel, Pascal
- C
- Fortran, Cobol, Basic…

☞ Libraries & interfaces (outside language definitions)

- POSIX
- MPI (Message Passing Interface)
- …

Languages with implicit concurrency: e.g. functional programming

Implicit concurrency in some programming schemes

Quicksort in a functional language (here: Haskell):

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

Pure functional programming is **side-effect free**

☞ Parameters can be evaluated independently ☞ *could* run concurrently

Some functional languages allow for **lazy evaluation**, i.e. sub-expressions are not necessarily evaluated completely:

```
borderline = (n /= 0) && (g (n) > h (n))
```

☞ If `n` equals zero then the evaluation of `g(n)` and `h(n)` can be stopped (or not even be started).

☞ Concurrent program parts **should be interruptible** in this case.

Short-circuit evaluations in imperative languages assume explicit sequential execution:

```
if Pointer /= nil and then Pointer.next = nil then …
```

Summary

Concurrency – The Basic Concepts

- **Forms of concurrency**
- **Models and terminology**
  - Abstractions and perspectives: computer science, physics & engineering
  - Observations: non-determinism, atomicity, interaction, interleaving
  - Correctness in concurrent systems
- **Processes and threads**
  - Basic concepts and notions
  - Process states
- **Concurrent programming languages:**
  - Explicit concurrency: e.g. Ada, Chapel
  - Implicit concurrency: functional programming – e.g. Haskell, Caml