Concurrent Systems

Nebenläufige Systeme

III. Processes

Wolfgang Schröder-Preikschat

November 8, 2016
Agenda

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
Outline

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
discussion on **abstract concepts** as to multiplexing machines:

- **program**
  - concretized form of an algorithm
  - static sequence of actions to be conducted by a processor
  - of sequential or non-sequential structure

- **process**
  - a program in execution
  - dynamic sequence of actions conducted by a processor
  - of parallel, concurrent, simultaneous, or interacting nature
discussion on **abstract concepts** as to multiplexing machines:

- **program**
  - concretized form of an algorithm
  - static sequence of actions to be conducted by a processor
  - of sequential or non-sequential structure

- **process**
  - a program in execution
  - dynamic sequence of actions conducted by a processor
  - of parallel, concurrent, simultaneous, or interacting nature

- explanation of **process characteristics** in physical and logical terms
  - appearance of a process as kernel thread and/or user thread
  - sequencing of processes, process states, and state transitions
discussion on **abstract concepts** as to multiplexing machines:

**program**
- concretized form of an algorithm
- static sequence of actions **to be** conducted by a processor
- of sequential or non-sequential structure

**process**
- a program in execution
- dynamic sequence of actions conducted by a processor
- of parallel, concurrent, simultaneous, or interacting nature

**process characteristics** in physical and logical terms
- appearance of a process as kernel thread and/or user thread
- sequencing of processes, process states, and state transitions

**bridging** of concurrency/simultaneity concepts and mechanisms
- on the one hand, program as the means of specifying a process
- on the other hand, process as medium to reflect simultaneous flows
Operating systems bring programs to execution by creation, releasing, controlling and timing of processes.
Operating systems bring programs to execution by creation, releasing, controlling and timing of processes

- in computer sciences, a process is unimaginable without a program
  - as coded representation of an algorithm, the program specifies a process
  - thereby, the program manifests and dictates a specific process
  - if so, it even causes, controls, or terminates other processes

1 Provided that the operating system offers all necessary commands.
Operating systems bring programs to execution by creation, releasing, controlling and timing of processes

- In computer sciences, a process is unimaginable without a program
  - As coded representation of an algorithm, the program specifies a process
  - Thereby, the program manifests and dictates a specific process
  - If so, it even causes, controls, or terminates other processes

- A program (also) describes the kind of flow (Ger. *Ablauf*) of a process
  - **Sequential**: A sequence of temporally non-overlapping actions
    - Proceeds deterministically, the result is determinate
  - **Parallel**: Non-sequential

---

1 Provided that the operating system offers all necessary commands.
Operating systems bring programs to execution by creation, releasing, controlling and timing of processes.

- In computer sciences, a process is unimaginable without a program.
- As coded representation of an algorithm, the program specifies a process.
- Thereby, the program manifests and dictates a specific process.
- If so, it even causes, controls, or terminates other processes.\(^1\)

- A program (also) describes the kind of flow (Ger. *Ablauf*) of a process.
  - **Sequential:** A sequence of temporally non-overlapping actions.
    - Proceeds deterministically, the result is determinate.
  - **Parallel:** Non-sequential.

In both kinds does the program flow consist of actions (p. 7 ff.).

Consider: Program Flow and Level of Abstraction

*One and the same program flow may be sequential on one level of abstraction and parallel on another.* [8, 10]

\(^1\)Provided that the operating system offers all necessary commands.
Outline

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
Definition
For a certain machine concretised form of an algorithm.
Program I

Problem-Oriented Language Level

Definition
For a certain machine concretised form of an algorithm.

virtual machine C
- after editing and
- before compilation

```c
#include <stdint.h>

void inc64(int64_t *i) {
    (*i)++;
}
```
Definition

For a certain machine concretised form of an algorithm.

- virtual machine C
  - after editing and
  - before compilation
  
```
#include <stdint.h>

void inc64(int64_t *i) {
    (*i)++;
}
```

- virtual machine ASM (x86)
  - after compilation
  - before assembly
  
```
inc64:
    movl 4(%esp), %eax
    addl $1, (%eax)
    adcl $0, 4(%eax)
    ret
```

---

\(^2\)gcc -O4 -m32 -static -fomit-frame-pointer -S, also below
Program I

Problem-Oriented/Assembly Language Level

Definition

For a certain machine concretised form of an algorithm.

- virtual machine C
  - after editing and
  - before compilation
  
  ```c
  #include <stdint.h>
  void inc64(int64_t *i) {
      (*i)++;
  }
  one action (line 4)
  ```

- virtual machine ASM (x86)
  - after compilation and
  - before assembly
  
  ```assembly
  inc64:
  movl 4(%esp), %eax
  addl $1, (%eax)
  addcl $0, 4(%eax)
  ret
  three actions (lines 7–9)
  ```

\(^2\)gcc -O4 -m32 -static -fomit-frame-pointer -S, also below
Program I

Problem-Oriented/Assembly Language Level

Definition

For a certain machine concretised form of an algorithm.

- virtual machine C
  - after editing and
  - before compilation

```c
#include <stdint.h>

void inc64 (int64_t *i) {
    (*i)++;
}
```

- virtual machine ASM (x86)
  - after compilation and
  - before assembly

```
inc64:
movl 4(%esp), %eax
addl $1, (%eax)
adc1l $0, 4(%eax)
ret
```

- one action (line 4)
- three actions (lines 7–9)

Definition (Action)

The execution of an instruction of a (virtual/real) machine.

\[^2\text{gcc -O4 -m32 -static -fomit-frame-pointer -S, also below}\]
address space and virtual machine SMC\textsuperscript{3}

- text segment
- Linux

after linking/binding and before loading

1. \texttt{0x080482f0:} \texttt{mov 0x4(%esp),%eax}
2. \texttt{0x080482f4:} \texttt{add $0x1,(%eax)}
3. \texttt{0x080482f7:} \texttt{adc $0x0,0x4(%eax)}
4. \texttt{0x080482fb:} \texttt{ret}

\textsuperscript{3}symbolic machine code: x86 + Linux.
address space and virtual machine SMC\(^3\)
- text segment
- Linux

1. \(0x080482f0\):
   - `mov 0x4(%esp),%eax`

2. \(0x080482f4\):
   - `add $0x1,(%eax)`

3. \(0x080482f7\):
   - `adc $0x0,0x4(%eax)`

4. \(0x080482fb\):
   - `ret`

real machine
- after loading
- executable

```assembly
8b 44 24 04
83 00 01
83 50 04 00
```

same number of actions (lines 1–3, resp.), but different forms of representation

\(^3\text{symbolic machine code}: x86 + \text{Linux.}\)
Program II

address space and virtual machine SMC

- text segment
- Linux

1 0x080482f0: mov 0x4(%esp),%eax
2 0x080482f4: add $0x1,(%eax)
3 0x080482f7: adc $0x0,0x4(%eax)
4 0x080482fb: ret

real machine

- after linking/binding and before loading
- after loading executable

CG 8b 44 24 04 83 00 01 83 50 04 00 c3

- same number of actions (lines 1–3, resp.), but different forms of representation

**Hint (ret or c3, resp.)**

The action for a subroutine return corresponds to the action of the corresponding subroutine call (gdb, disasm /rm main):

1 0x080481c9: c7 04 24 b0 37 0d 08 movl $0x80d37b0,(%esp)
2 0x080481d0: e8 1b 01 00 00 call 0x80482f0 <inc64>

symbolic machine code: x86 + Linux.
Non-Sequential Program I

**Definition**

A program $P$ specifying actions that allow for parallel flows in $P$ itself.

```c
pthread_t tid;
if (!pthread_create (&tid, NULL, thread, NULL)) {
    /* ... */
    pthread_join (tid, NULL);
}

void* thread (void* null) {
    /* ... */
    pthread_exit (NULL);
}
```
Non-Sequential Program I

Definition

A program $P$ specifying actions that allow for parallel flows in $P$ itself.

an excerpt of $P$ using the example of POSIX Threads [4]:

```c
#include <pthread.h>

int main(void) {
    pthread_t tid;

    if (!pthread_create(&tid, NULL, thread, NULL)) {
        /* ... */
        pthread_join(tid, NULL);
    }
    return 0;
}
```

void *thread(void *null) {
    /* ... */
    pthread_exit(NULL);
    return NULL;
}
Definition
A program $P$ specifying actions that allow for parallel flows in $P$ itself.

an excerpt of $P$ using the example of *POSIX Threads* [4]:

```c
pthread_t tid;

if (!pthread_create(&tid, NULL, thread, NULL)) {
    /* ... */
    pthread_join(tid, NULL);
}
```

the parallel flow allowed in $P$ itself:

```c
void *thread(void *null) {
    /* ... */
    pthread_exit(NULL);
}
```
Non-Sequential Program II

despite actions of parallelism, **sequential flows** of the same program:

```c
pid_t pid;

if (!(pid = fork())) {
    /* ... */
    exit(0);
}

wait(NULL);
```

- fork duplicates the address space $A$ of $P$, creates $A'$ as a copy of $A$
- within $A$ as source address space arises thereby no parallel flow, however
- independent of the degree of parallelism within $P$, fork sets it to 1 for $A'$
Non-Sequential Program II

despite actions of parallelism, **sequential flows** of the same program:

```c
pid_t pid;

if (!(pid = fork())) {
    /* ... */
    exit(0);
}
wait(NULL);
```

- fork duplicates the address space $A$ of $P$, creates $A'$ as a copy of $A$
- within $A$ as source address space arises thereby no parallel flow, however
- independent of the degree of parallelism within $P$, fork sets it to 1 for $A'$

- sequential flows can establish parallel ones within a domain that logically comprises those sequential flows
Non-Sequential Program II

despite actions of parallelism, sequential flows of the same program:

```c
pid_t pid;

if (!(pid = fork())) {
    /* ... */
    exit(0);
}
wait(NULL);
```

- fork duplicates the address space $A$ of $P$, creates $A'$ as a copy of $A$
- within $A$ as source address space arises thereby no parallel flow, however
- independent of the degree of parallelism within $P$, fork sets it to 1 for $A'$

- sequential flows can establish parallel ones within a domain that logically comprises those sequential flows

- the shown actions cause parallel flows within an operating system
  - multiprocessing (Ger. Simultanbetrieb) of sequential programs requires the operating system in the shape of a non-sequential program
  - serviceable characteristic is multithreading within the operating system
Non-Sequential Program II

despite actions of parallelism, sequential flows of the same program:

```c
pid_t pid;

if (!(pid = fork())) {
    /* ... */
    exit(0);
}
wait(NULL);
```

- fork duplicates the address space $A$ of $P$, creates $A'$ as a copy of $A$
- within $A$ as source address space arises thereby no parallel flow, however
- independent of the degree of parallelism within $P$, fork sets it to 1 for $A'$

- sequential flows can establish parallel ones within a domain that logically comprises those sequential flows

- the shown actions cause parallel flows within an operating system
  - multiprocessing (Ger. Simultanbetrieb) of sequential programs requires the operating system in the shape of a non-sequential program
  - serviceable characteristic is multithreading within the operating system

concept “operating system” is epitome of “non-sequential program”

4The exception (strictly cooperative systems) proves the rule.
address space A

```c
fork()
wait(NULL)
```

directions

processor (core) characteristic:
- Unioperated by a process-based operating system, namely:
  - pseudo-parallelism by means of processor (core) multiplexing
- Multi
ditto; but also event-based operating system, namely:
  - real parallelism by means of processor (core) multiplication
  
both cause parallel processes (p.16) within the operating system
Multiprocessing of Sequential Programs

address space A

parent

fork()

wait(NULL)

address space A'

child

/* ... */

exit(0)

© wosch CS (WS 2016, LEC 3) Fundamentals – Program
Multiprocessing of Sequential Programs

address space A

parent

fork()

wait(NULL)

address space A'

child

duplicate

sequential program flows

fork()

wait(NULL)

exit(0)

© wosch CS (WS 2016, LEC 3) Fundamentals–Program 11
Multiprocessing of Sequential Programs

address space A

**parent**

fork()

wait(NULL)

address space OS

**operating system**

duplicate

address space A'

**child**

exec(BAR)

/* ... */

exit(0)

sequential program flows

non-sequential program flow

processor (core) characteristic:

uni

operated by a process-based operating system, namely:
pseudo-parallelism by means of processor (core) multiplexing

ditto

multi

ditto; but also event-based operating system, namely:
real parallelism by means of processor (core) multiplication

both cause

parallel processes

(p.16)
Multiprocessing of Sequential Programs

processor (core) characteristic:

**Uni**
- operated by a process-based operating system, namely:
  - pseudo-parallelism by means of processor (core) multiplexing

**Multi**
- ditto
Multiprocessing of Sequential Programs

Address space A

- Parent
  - `fork()`
  - `wait(NULL)`

Address space A’

- Child
  - `exit(0)`

Address space OS

Operating system

Sequential program flows

- Duplicate

Non-sequential program flow

Processor (core) characteristic:

**Uni**
- Operated by a process-based operating system, namely:
  - Pseudo-parallelism by means of processor (core) multiplexing

**Multi**
- Ditto; but also event-based operating system, namely:
  - Real parallelism by means of processor (core) multiplication
Multiprocessing of Sequential Programs

- **processor (core) characteristic:**
  - **Uni** operated by a process-based operating system, namely:
    - pseudo-parallelism by means of processor (core) multiplexing
  - **Multi** ditto; but also event-based operating system, namely:
    - real parallelism by means of processor (core) multiplication

  both cause **parallel processes** (p. 16) within the operating system
## Process

### Definition (Program flow)

A program in execution.

<table>
<thead>
<tr>
<th>Level</th>
<th>Mapping</th>
<th>Assembly Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$\rightarrow$</td>
<td>program statement $\geq 1$ assembly mnemonics</td>
</tr>
<tr>
<td>4</td>
<td>$\rightarrow$</td>
<td>assembly mnemonic $\geq 1$ machine instructions</td>
</tr>
<tr>
<td>3</td>
<td>$\rightarrow$</td>
<td>machine instruction $\geq 1$ microprogram directives</td>
</tr>
<tr>
<td>2</td>
<td>$\rightarrow$</td>
<td>microprogram directive</td>
</tr>
</tbody>
</table>
A program in execution.

- the program specifies a sequence of actions that are to be executed
- its kind depends on the particular **level of abstraction** (cf. p. 34)
  - level$_5 \mapsto$ program statement
  - level$_4 \mapsto$ assembly mnemonic
  - level$_3 \mapsto$ machine instruction
  - level$_2 \mapsto$ microprogram directive

**Hint (Process $\neq$ Process instance)**

A process instance (Ger. *Exemplar*) is an incarnation of a process.

Just as an object is a "core image" of a class.
Process

Definition (Program flow)

A program in execution.

- the program specifies a sequence of actions that are to be executed
  - its kind depends on the particular **level of abstraction** (cf. p. 34)
    - level$_5$ $\mapsto$ program statement & $\geq 1$ assembly mnemonics
    - level$_4$ $\mapsto$ assembly mnemonic & $\geq 1$ machine instructions
    - level$_3$ $\mapsto$ machine instruction & $\geq 1$ microprogram directives
    - level$_2$ $\mapsto$ microprogram directive
- the actions of a processor thus are **not imperatively indivisible** (atomic)
  - this particularly holds both for the abstract (virtual) and real processor
A program in execution.

- the program specifies a sequence of actions that are to be executed
- its kind depends on the particular **level of abstraction** (cf. p. 34)
  - level \(_5\) $\mapsto$ program statement $\geq 1$ assembly mnemonics
  - level \(_4\) $\mapsto$ assembly mnemonic $\geq 1$ machine instructions
  - level \(_3\) $\mapsto$ machine instruction $\geq 1$ microprogram directives
  - level \(_2\) $\mapsto$ microprogram directive
- the actions of a processor thus are not imperatively indivisible (atomic)
  - this particularly holds both for the abstract (virtual) and real processor
- this sequence is static (passiv), while a process is dynamic (active)
Process

Definition (Program flow)

A program in execution.

- the program specifies a sequence of actions that are to be executed
  - its kind depends on the particular level of abstraction (cf. p. 34)
    - level 5 $\mapsto$ program statement $\geq 1$ assembly mnemonics
    - level 4 $\mapsto$ assembly mnemonic $\geq 1$ machine instructions
    - level 3 $\mapsto$ machine instruction $\geq 1$ microprogram directives
    - level 2 $\mapsto$ microprogram directive

- the actions of a processor thus are not imperatively indivisible (atomic)
  - this particularly holds both for the abstract (virtual) and real processor

- this sequence is static (passiv), while a process is dynamic (active)

Hint (Process $\neq$ Process instance)

A process instance (Ger. Exemplar) is incarnation of a process.\(^a\)

\(^a\)Just as an object is a “core image” of a class.
Indivisibility I

**Definition**

Being indivisible, to keep something appear as unit or entireness.

- a question of the “distance” of the viewer (subject) on an object
Indivisibility I

Definition

Being indivisible, to keep something appear as unit or entireness.

- a question of the “distance” of the viewer (subject) on an object
- **action** on higher, **sequence of actions** on lower level of abstraction

<table>
<thead>
<tr>
<th>level</th>
<th>action</th>
<th>sequence of actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>i++</td>
<td>*</td>
</tr>
<tr>
<td>4–3</td>
<td>incl i</td>
<td>movl i,%r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addl $1,%r*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl %r,i</td>
</tr>
<tr>
<td>2–1</td>
<td></td>
<td>* <strong>read</strong> from memory into accumulator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* <strong>modify</strong> contents of accumulator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* <strong>write</strong> from accumulator into memory</td>
</tr>
</tbody>
</table>

- typical for a complex instruction of an “abstract processor” (C, CISC)
Entireness or unit of a sequence of actions whose solo efforts all will happen apparently simultaneous (i.e., are synchronised)
Entireness or unit of a sequence of actions whose solo efforts all will happen \textit{apparently simultaneous} (i.e., are synchronised)

- an\textit{/the} essential non-functional property of an \textit{atomic operation} \textsuperscript{5}
  - logical togetherness of a sequence of actions in terms of time
  - by what that sequence appears as \textit{elementary operation} (ELOP)

\textsuperscript{5}from (Gr.) \textit{átomo} “indivisible”.
Entireness or unit of a sequence of actions whose solo efforts all will happen **apparently simultaneous** (i.e., are synchronised)

- an/the essential non-functional property of an **atomic operation**
  - logical togetherness of a sequence of actions in terms of time
  - by what that sequence appears as **elementary operation** (ELOP)

- examples of (critical) actions for incrementation of a counter variable:
  - level $5 \rightarrow 3$
    - **C/C++**
      ```
      i ++;
      ```
    - **ASM**
      ```
      movl i, % eax
      addl $1 , % eax
      movl % eax , i
      ```

---

$^5$ from (Gr.) átomo “indivisible”.
Entireness or unit of a sequence of actions whose solo efforts all will happen apparently simultaneous (i.e., are synchronised)

- an/the essential non-functional property of an atomic operation\(^5\)
- logical togetherness of a sequence of actions in terms of time
- by what that sequence appears as elementary operation (ELOP)

Examples of (critical) actions for incrementation of a counter variable:

- level \(_5 \rightarrow 3\)
  
  **C/C++**
  
  1 i++;

- level \(_3 \rightarrow 2\)
  
  **ASM**
  
  5 incl i
  6 read A from <i>
  7 modify A by 1
  8 write A to <i>

---

\(^5\)from (Gr.) átomo “indivisible.”
Indivisibility II

Entireness or unit of a sequence of actions whose solo efforts all will happen apparently simultaneous (i.e., are synchronised)

- the essential non-functional property of an atomic operation
  - logical togetherness of a sequence of actions in terms of time
  - by what that sequence appears as elementary operation (ELOP)

- examples of (critical) actions for incrementation of a counter variable:
  - level \( 5 \rightarrow 3 \)
    - C/C++
      1. `i++;`
    - ASM
      2. `movl i, %eax`
      3. `addl $1, %eax`
      4. `movl %eax, i`
  
  - level \( 3 \rightarrow 2 \)
    - ASM
      5. `incl i`
    - ISA
      6. `read A from <i>`
      7. `modify A by 1`
      8. `write A to <i>`

- points (i++, incl) in case of merely conditionally atomic execution
  - namely uninterruptible operation (level \( 5 \rightarrow 3 \)), uniprocessor (Ebene \( 3 \rightarrow 2 \))
  - problem: overlapping in time of the sequence of actions pointed here

\[^{5}\text{from (Gr.) átomo “indivisible”}\]
Definition

A process that is composed exclusively of a sequence of temporally non-overlapping actions.
Sequential Process

**Definition**

A process that is composed exclusively of a sequence of temporally non-overlapping actions.

- the sequence of actions forms a unique **execution thread**
  - of which always only a single one exists within a sequential process
  - but which may develop differently with each restart of that process
    - other input data, program change, ... , transient hardware errors
Sequential Process

**Definition**
A process that is composed exclusively of a sequence of temporally non-overlapping actions.

- the sequence of actions forms a unique **execution thread**
  - of which always only a single one exists within a sequential process
  - but which may develop differently with each restart of that process
    - other input data, program change, ..., transient hardware errors
- the sequence is defined by a **total order** of its actions
  - it is reproducible given unmodified original conditions
Sequential Process

**Definition**

A process that is composed exclusively of a sequence of temporally non-overlapping actions.

- the sequence of actions forms a unique **execution thread**
  - of which always only a single one exists within a sequential process
  - but which may develop differently with each restart of that process
    - other input data, program change, . . . , transient hardware errors
- the sequence is defined by a **total order** of its actions
  - it is reproducible given unmodified original conditions

**Hint (Execution Thread ≠ Thread)**

*Assumptions about the technical implementation of the sequence of actions are not met and are also irrelevant here. A thread is only one option to put the incarnation of a sequential process into effect.*
Non-Sequential Process

Definition

Also referred to as “parallel”, namely a process that is composed of a sequence of temporally overlapping actions.
Non-Sequential Process

Definition

Also referred to as “parallel”, namely a process that is composed of a sequence of temporally overlapping actions.

- requirement is a **non-sequential program** (cf. p. 9)
  - that allows for at least one more process incarnation (child process) or
  - that makes arrangements for the handling of events of external processes

---

6 Interrupt requests issued by some device (IRQ) oder process (signal).
Non-Sequential Process

Definition

Also referred to as “parallel”, namely a process that is composed of a sequence of temporally overlapping actions.

- requirement is a **non-sequential program** (cf. p. 9)
  - that allows for at least one more process incarnation (child process) or
  - that makes arrangements for the handling of events of external processes

whereby sequences of actions may overlap in the first place:

i. multithreading (Ger. *simultane Mehrfäigkeit*), in fact:
  - pseudo-parallel – multiplex mode of a single processor (core)
  - real parallel – parallel mode of a (multi-core) multiprocessor

ii. asynchronous program interrupts

---

*Interrupt requests issued by some device (IRQ) oder process (signal).*
Non-Sequential Process

Definition

Also referred to as “parallel”, namely a process that is composed of a sequence of temporally overlapping actions.

- requirement is a **non-sequential program** (cf. p. 9)
  - that allows for at least one more process incarnation (child process) *or*
  - that makes arrangements for the handling of events of external processes

- whereby sequences of actions may overlap in the first place:
  1. multithreading (Ger. *simultane Mehrfälgigkeit*), in fact:
     - pseudo-parallel – multiplex mode of a single processor (core)
     - real parallel – parallel mode of a (multi-core) multiprocessor
  2. asynchronous program interrupts

- consequently, the sequence of all actions is defined by a **partial order**
  - as external processes may enable temporal/causal independent actions

\[^6\]Interrupt requests issued by some device (IRQ) oder process (signal).\[^6\]
Definition (in a broader sense: “simultaneous processes”)

One or more (non-sequential) processes whose sequences of actions will overlap in time area by area (Ger. *bereichsweise*).
Definition (in a broader sense: “simultaneous processes”)

One or more (non-sequential) processes whose sequences of actions will overlap in time area by area (Ger. *bereichsweise*).

- areas are **concurrent** (Ger. *nebenläufig*) only if they are independent
  - none of these concurrent processes is cause or effect of the other
  - none of theses actions of these processes requires the result of any other
Concurrent Processes

Definition (in a broader sense: “simultaneous processes”)

One or more (non-sequential) processes whose sequences of actions will overlap in time area by area (Ger. *bereichsweise*).

- areas are **concurrent** (Ger. *nebenläufig*) only if they are independent
  - none of these concurrent processes is cause or effect of the other
  - none of these actions of these processes requires the result of any other
- to proceed, concurrent processes compete for **reusable resources**
  - they share the processor (core), cache (line), bus, or devices
  - outcome of this is **interference**⁷ (Ger. *Interferenz*) in process behaviour

---

⁷Derived from (Fre.) *s’entreferir* “to brawl each other”.
Concurrent Processes

Definition (in a broader sense: “simultaneous processes”)

One or more (non-sequential) processes whose sequences of actions will overlap in time area by area (Ger. *bereichsweise*).

- Areas are **concurrent** (Ger. *nebenläufig*) only if they are independent:
  - None of these concurrent processes is cause or effect of the other.
  - None of these actions of these processes requires the result of any other.

- To proceed, concurrent processes compete for **reusable resources**:
  - They share the processor (core), cache (line), bus, or devices.
  - Outcome of this is **interference**\(^7\) (Ger. *Interferenz*) in process behaviour.

- The effective degree of overlapping is irrelevant for the simultaneity:
  - Apart from time-dependent processes that have to keep deadlines.
  - Note that the larger the overlapping, the larger the time delay.
    - And the more likely will a delayed process miss its deadline.
  - Just as interference, which may also cause violation of timing constraints.

\(^7\)Derived from (Fre.) *s’entreferir* “to brawl each other.”
Definition (also: “depending processes”)

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource. Their actions get into conflict if at least one of these processes... will change the value of one of the shared variables (access pattern) or already acquires a shared non-preemptable resource (resource type). This may emerge as a race condition (Ger. Wettlaufsituation) for shared variables or (reusable/consumable) resources, resp. for starting or finishing an intended sequence of actions. Conflicts are eliminated by means of synchronisation methods: blocking prevent from executing an intended sequence of actions, non-blocking let a process abort and retry a started sequence of actions. Reducing finds coordination of cooperation and competition of processes.

8 printer, mouse, plotter, keyboard.
Definition (also: “depending processes”)

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

Their actions get into **conflict** if at least one of these processes...

- will change the value of one of the shared variables (**access pattern**) or
- already occupies a shared **non-preemptable** resource

---

8 printer, mouse, plotter, keyboard.
Definition (also: “depending processes”)

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

- their actions get into **conflict** if at least one of these processes...
  - will change the value of one of the shared variables (**access pattern**) or
  - already occupies a shared **non-preemptable resource**\(^8\) (**resource type**)
- this may emerge as a **race condition** (Ger. *Wettlaufs situation*)
  - for shared variables or (reusable/consumable) resources, resp.
  - for starting or finishing an intended sequence of actions

\(^8\) printer, mouse, plotter, keyboard.
Interacting Processes I  
(Ger.) *gekoppelte Prozesse* [3, p. 77]

---

**Definition (also: “depending processes”)**

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

- their actions get into **conflict** if at least one of these processes...
  - will change the value of one of the shared variables (**access pattern**) **or**
  - already occupies a shared **non-preemptable** resource[^8] (**resource type**)
- this may emerge as a **race condition** (Ger. *Wettlaufsituation*)
  - for shared variables or (reusable/consumable) resources, resp.
  - for starting or finishing an intended sequence of actions
- conflicts are eliminated by means of **synchronisation methods**:
  - **blocking**  ■ prevent from executing an intended sequence of actions
  - **non-blocking**  ■ let a process abort and retry a started sequence of actions

---

[^8]: printer, mouse, plotter, keyboard.
Definition (also: “depending processes”)

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

- their actions get into **conflict** if at least one of these processes...
  - will change the value of one of the shared variables (access pattern) or
  - already occupies a shared non-preemptable resource⁸ (resource type)

- this may emerge as a **race condition** (Ger. *Wettlaufsituation*)
  - for shared variables or (reusable/consumable) resources, resp.
  - for starting or finishing an intended sequence of actions

- conflicts are eliminated by means of **synchronisation methods**:
  - **blocking** prevent from executing an intended sequence of actions
  - **non-blocking** let a process abort and retry a started sequence of actions
  - **reducing** replace a sequence of actions by an atomic instruction

---

⁸ printer, mouse, plotter, keyboard.
Interacting Processes I (Ger.) *gekoppelte Prozesse* [3, p. 77]

**Definition (also: “depending processes”)**

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

- Their actions get into **conflict** if at least one of these processes... 
  - Will change the value of one of the shared variables (access pattern) or
  - Already occupies a shared non-preemptable resource\(^8\) (resource type)
- This may emerge as a **race condition** (Ger. *Wettlaufsituation*)
  - For shared variables or (reusable/consumable) resources, resp.
  - For starting or finishing an intended sequence of actions
- Conflicts are eliminated by means of **synchronisation methods**:
  - **Blocking**
    - Prevent from executing an intended sequence of actions
  - **Non-blocking**
    - Let a process abort and retry a started sequence of actions
  - **Reducing**
    - Replace a sequence of actions by an atomic instruction
- Founds **coordination** of cooperation and competition of processes

\(^8\) Printer, mouse, plotter, keyboard.
```c
int64_t cycle = 0;

void *thread_worker(void *null) {
    for (;;) {
        /* ... */
        inc64(&cycle);
    }
}

void *thread_minder(void *null) {
    for (;;) {
        printf("worker cycle %lld\n", cycle);
        pthread_yield();
    }
}
```

- inc64: see p. 7
```c
int64_t cycle = 0;

void *thread_worker(void *null) {
    for (;;) {
        /* ... */
        inc64(&cycle);
    }
}

void *thread_minder(void *null) {
    for (;;) {
        printf("worker cycle %lld\n", cycle);
        pthread_yield();
    }
}
```

Which cycle values prints the minder thread (Ger. *Aufpasserfaden*)?
```c
int64_t cycle = 0;

void *thread_worker(void *null) {
  for (;;) {
    /* ... */
    inc64(&cycle);
  }
}

void *thread_minder(void *null) {
  for (;;) {
    printf("worker cycle %lld\n", cycle);
    pthread_yield();
  }
}
```

Which cycle values prints the minder thread (Ger. *Aufpasserfaden*)?

Which are produced by multiple worker threads (Ger. *Arbeiterfäden*)?

In case thread_worker exists in several identical incarnations
assuming that the non-sequential program runs on a 32-bit machine

- instances of `int64_t` then form a pair of 32-bit words: double word
- operations on instances of `int64_t` cease to be solo efforts
Interacting Processes III

1. Race Condition

- assuming that the non-sequential program runs on a 32-bit machine
  - instances of int64_t then form a pair of 32-bit words: double word
  - operations on instances of int64_t cease to be solo efforts

worker thread

```
1 inc64:
2    movl 4(%esp), %eax
3    addl $1, (%eax)
4    adcl $0, 4(%eax)
5    ret

6 .L6:
7    movl $cycle, (%esp)
8    call inc64
9    jmp .L6
```
assuming that the non-sequential program runs on a 32-bit machine

- instances of `int64_t` then form a pair of 32-bit words: **double word**
- operations on instances of `int64_t` **cease to be solo efforts**

**worker thread**

1. `inc64:`
2. `movl 4(%esp), %eax`
3. `addl $1, (%eax)`
4. `adcl $0, 4(%eax)`
5. `ret`

6. `.L6:`
7. `movl $cycle, (%esp)`
8. `call inc64`
9. `jmp .L6`

**minder thread**

10. `movl cycle+4, %edx ; high &`
11. `movl cycle, %eax ; low word`
12. `movl $.LC0 , (%esp)`
13. `movl %edx, 8(%esp)`
14. `movl %eax, 4(%esp)`
15. `call printf`

**Assume**

- `cycle = 2^{32} - 1`
- `inc64` overlaps actions 10–11 then, `edx = 0` and `eax = 0`
- effect is, `printf` displays 0 not $2^{32}$, as would have been right
assuming that the non-sequential program runs on a 32-bit machine

- instances of int64_t then form a pair of 32-bit words: double word
- operations on instances of int64_t cease to be solo efforts

worker thread

```
inc64:
  movl 4(%esp), %eax
  addl $1, (%eax)
  adcl $0, 4(%eax)
  ret

.L6:
  movl $cycle, (%esp)
  call inc64
  jmp .L6
```

minder thread

```
10  movl  cycle+4, %edx ; high &
11  movl  cycle,  %eax  ; low word
12  movl  $.LC0,  (%esp)
13  movl  %edx,  8(%esp)
14  movl  %eax,  4(%esp)
15  call printf

• assume cycle = 2^{32} − 1
```
Interacting Processes III

1. Race Condition

assuming that the non-sequential program runs on a 32-bit machine
- instances of int64_t then form a pair of 32-bit words: double word
- operations on instances of int64_t cease to be solo efforts

worker thread

1. inc64:
2. movl 4(%esp), %eax
3. addl $1, (%eax)
4. adcl $0, 4(%eax)
5. ret

.mL6:
6. movl $cycle, (%esp)
7. call inc64
8. jmp .L6

minder thread

10. movl cycle+4, %edx ; high &
11. movl cycle, %eax ; low word
12. movl $.LC0, (%esp)
13. movl %edx, 8(%esp)
14. movl %eax, 4(%esp)
15. call printf

- assume cycle = $2^{32} - 1
- inc64 overlaps actions 10–11

© wosch  CS (WS 2016, LEC 3)  Fundamentals – Process
Interacting Processes III

1. Race Condition

assuming that the non-sequential program runs on a 32-bit machine

- instances of `int64_t` then form a pair of 32-bit words: **double word**
- operations on instances of `int64_t` *cease to be solo efforts*

**worker thread**

```assembly
inc64:
1  movl 4(%esp), %eax
2  addl $1, (%eax)
3  adcl $0, 4(%eax)
4  ret

.L6:
5  movl $cycle, (%esp)
6  call inc64
7  jmp .L6
```

**minder thread**

```assembly
10  movl cycle +4, %edx ; high &
11  movl cycle, %eax ; low word
12  movl $.LC0, (%esp)
13  movl %edx, 8(%esp)
14  movl %eax, 4(%esp)
15  call printf

16  assume cycle = 2^{32} − 1
17  inc64 overlaps actions 10–11
18  then, edx = 0 and eax = 0
19  effect is, printf displays 0
20  – not 2^{32}, as would have been right
```
assuming that the development or run-time environment varies

- different compilers, assemblers, linker, or loaders
- different operating systems—but the same real processor (x86)
assuming that the development or run-time environment varies
- different compilers, assemblers, linker, or loaders
- different operating systems—but the same real processor (x86)

GCC 4.7.2, Linux

```plaintext
inc64:
    movl 4(%esp), %eax
    addl $1, (%eax)
    adcl $0, 4(%eax)
    ret
```

real parallel actions: (multi-core) multiprocessor
the actions in lines 3–4 are critical as well: divisible
read-modify-write
a classical error: as the case may be, ineffective numeration
assuming that the development or run-time environment varies
- different compilers, assemblers, linker, or loaders
- different operating systems—but the same real processor (x86)

GCC 4.7.2, Linux

```c
inc64:
  movl 4(%esp), %eax
  addl $1, (%eax)
  adcl $0, 4(%eax)
  ret
```

GCC 4.2.1, MacOSX

```c
_inc64:
  movl 4(%esp), %eax
  movl (%eax), %ecx
  movl 4(%eax), %edx
  addl $1, %ecx
  adcl $0, %edx
  movl %edx, 4(%eax)
  movl %ecx, (%eax)
  ret
```
assuming that the development or run-time environment varies
- different compilers, assemblers, linker, or loaders
- different operating systems—but the same real processor (x86)

GCC 4.7.2, Linux

```c
inc64:
  movl 4(%esp), %eax
  addl $1, (%eax)
  adcl $0, 4(%eax)
  ret
```

GCC 4.2.1, MacOSX

```c
_inc64:
  movl 4(%esp), %eax
  movl (%eax), %ecx
  movl 4(%eax), %edx
  addl $1, %ecx
  adcl $0, %edx
  movl %edx, 4(%eax)
  movl %ecx, (%eax)
  ret
```

pseudo-parallel actions (case 4.2.1)
- (UNIX-) signal
- asynchronous program interrupt
assuming that the development or run-time environment varies
- different compilers, assemblers, linker, or loaders
- different operating systems—but the same real processor (x86)

GCC 4.7.2, Linux

```
1 inc64:
2    movl 4(%esp), %eax
3    addl $1, (%eax)
4    adcl $0, 4(%eax)
5    ret
```

GCC 4.2.1, MacOSX

```
6 _inc64:
7    movl 4(%esp), %eax
8    movl (%eax), %ecx
9    movl 4(%eax), %edx
10   addl $1, %ecx
11   adcl $0, %edx
12   movl %edx, 4(%eax)
13   movl %ecx, (%eax)
14   ret
```

pseudo-parallel actions (case 4.2.1)
- (UNIX-) signal
- asynchronous program interrupt

real parallel actions: (multi-core) multiprocessor
- the actions in lines 3–4 are critical as well: divisible read-modify-write
assuming that the development or run-time environment varies
- different compilers, assemblers, linker, or loaders
- different operating systems—but the same real processor (x86)

<table>
<thead>
<tr>
<th>GCC 4.7.2, Linux</th>
<th>GCC 4.2.1, MacOSX</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc64:</td>
<td>_inc64:</td>
</tr>
<tr>
<td>movl 4(%esp), %eax</td>
<td>movl 4(%esp), %eax</td>
</tr>
<tr>
<td>addl $1, (%eax)</td>
<td>movl (%eax), %ecx</td>
</tr>
<tr>
<td>adcl $0, 4(%eax)</td>
<td>movl 4(%eax), %edx</td>
</tr>
<tr>
<td>ret</td>
<td>addl $1, %ecx</td>
</tr>
<tr>
<td></td>
<td>adcl $0, %edx</td>
</tr>
<tr>
<td></td>
<td>movl %edx, 4(%eax)</td>
</tr>
<tr>
<td></td>
<td>movl %ecx, (%eax)</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

pseudo-parallel actions (case 4.2.1)
- (UNIX-) signal
- asynchronous program interrupt

real parallel actions: (multi-core) multiprocessor
- the actions in lines 3–4 are critical as well: divisible read-modify-write

a classical error: as the case may be, ineffective numeration
Outline

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
Consistency

Coordination of Interacting Processes

- prevention of race conditions by the **protection of critical sections**
- transfer a non-sequential process into a temporary sequential process
  - strictly: the shorter the sequential time span, the better the solution
- or, if applicable, rewrite conflict-prone program sequences as a transaction
Consistency

prevention of race conditions by the **protection of critical sections**

- transfer a non-sequential process into a temporary sequential process
  - strictly: the shorter the sequential time span, the better the solution
- or, if applicable, rewrite conflict-prone program sequences as a transaction

**Lookahead**: prevent overlapping by means of **mutual exclusion**

blocking of interacting processes

```c
void mutex_inc64(int64_t *i, pthread_mutex_t *lock) {
    pthread_mutex_lock(lock);    /* indivisible, now */
    inc64(i);                     /* reuse code @ p. 7 */
    pthread_mutex_unlock(lock);  /* divisible, again */
}
```
Consistency

Coordination of Interacting Processes

- prevention of race conditions by the **protection of critical sections**
  - transfer a non-sequential process into a temporary sequential process
    - strictly: the shorter the sequential time span, the better the solution
  - or, if applicable, rewrite conflict-prone program sequences as a transaction

**Lookahead:** prevent overlapping by means of **mutual exclusion**

blocking of interacting processes: **comparatively long time span**

```c
void mutex_inc64(int64_t *i, pthread_mutex_t *lock) {
    pthread_mutex_lock(lock);    /* indivisible, now */
    inc64(i);                    /* reuse code @ p.7 */
    pthread_mutex_unlock(lock); /* divisible, again */
}
```

© wosch    CS (WS 2016, LEC 3)    Characteristics – Physical
Consistency

Coordinated Interacting Processes

- Prevention of race conditions by the **protection of critical sections**
  - Transfer a non-sequential process into a temporary sequential process
    - Strictly: the shorter the sequential time span, the better the solution
  - Or, if applicable, rewrite conflict-prone program sequences as a transaction

**Lookahead**: prevent overlapping by means of **mutual exclusion**

- Blocking of interacting processes: comparably long time span

```c
void mutex_inc64(int64_t *i, pthread_mutex_t *lock) {
    pthread_mutex_lock(lock); /* indivisible, now */
    inc64(i); /* reuse code @ p.7 */
    pthread_mutex_unlock(lock); /* divisible, again */
}
```

- Reducing to a 64-bit ELOP of the real processor

```c
void inc64(int64_t *i) { /* renew code @ p.7 */
    asm ("lock incq %0": : "m" (*i): "memory");
}
```

- Anywhere applicable and by orders of magnitude more efficient solution
anchoring of processes can be different within a computing system
Localisation

Operating-System v. Application Context

- **anchoring** of processes can be different within a computing system
  - namely inside or outside the operating-system machine level:
    - **inside** – originally, within the operating system or its kernel
      - incarnation of the process is root of possibly other processes
      - partial virtualisation of the CPU as the real processor (core)
        ⇩ “kernel thread”, in computer science folklore
    - **outside** – optional, within run-time or even application system
      - incarnation of the process as leaf or inner node (of a graph)
      - partial virtualisation of the root process as an abstract processor
        ⇩ “user thread”, in computer science folklore
Localisation

Operating-System v. Application Context

- anchoring of processes can be different within a computing system
  - namely inside or outside the operating-system machine level:
    - **inside** - originally, within the operating system or its kernel
      - incarnation of the process is root of possibly other processes
      - partial virtualisation of the CPU as the real processor (core)
      - "kernel thread", in computer science folklore
    - **outside** - optional, within run-time or even application system
      - incarnation of the process as leaf or inner node (of a graph)
      - partial virtualisation of the root process as an abstract processor
      - "user thread", in computer science folklore
  - usually, a processor (core) is entirely unaware of being multiplexed
    - threads evolve from time sharing their underlying processor (core)
      - a kernel thread may serve as an **abstract processor** for user threads
Localisation

Operating-System v. Application Context

- anchoring of processes can be different within a computing system
  - namely inside or outside the operating-system machine level:
    - inside - originally, within the operating system or its kernel
      - incarnation of the process is root of possibly other processes
      - partial virtualisation of the CPU as the real processor (core)
        ↦ “kernel thread”, in computer science folklore
    - outside - optional, within run-time or even application system
      - incarnation of the process as leaf or inner node (of a graph)
      - partial virtualisation of the root process as an abstract processor
        ↦ “user thread”, in computer science folklore

- usually, a processor (core) is entirely unaware of being multiplexed
  - threads evolve from time sharing their underlying processor (core)
    - a kernel thread may serve as an abstract processor for user threads
  - no nowadays known (real) processor is aware of what it is processing
    - particularly, a kernel thread does not know about potential user threads
    - when it gets switched or delayed, all of its user threads will as well
Localisation

**anchoring** of processes can be different within a computing system

- **inside** originally, within the operating system or its kernel
  - incarnation of the process is root of possibly other processes
  - partial virtualisation of the CPU as the real processor (core)
  - \(\rightarrow\) "kernel thread", in computer science folklore
- **outside** optional, within run-time or even application system
  - incarnation of the process as leaf or inner node (of a graph)
  - partial virtualisation of the root process as an abstract processor
  - \(\rightarrow\) "user thread", in computer science folklore

usually, a processor (core) is entirely unaware of being multiplexed

- threads evolve from time sharing their underlying processor (core)
  - a kernel thread may serve as an **abstract processor** for user threads
- no nowadays known (real) processor is aware of what it is processing
  - particularly, a kernel thread does not know about potential user threads
  - when it gets switched or delayed, **all** of its user threads will as well

operating systems are aware only of their own "first-class citizens"
Weight Category

• feather–, ** lightweight
  • partial virtualization

modes of process switches as to partial processor virtualisation:
  * inside the same (user/kernel) address space, ibidem\(^9\) continuing
  ** inside kernel address space, same user address space sharing

\(^9\) (Lat.), “at the same place”
modes of **process switches** as to partial processor virtualisation:

* inside the same (user/kernel) address space, *ibidem*\textsuperscript{9} continuing
** inside kernel address space, same user address space sharing
*** inside kernel address space, at other user address space landing

\textsuperscript{9}(Lat.), “at the same place”
modes of **process switches** as to partial processor virtualisation:

* inside the same (user/kernel) address space, *ibidem*\(^9\) continuing

** inside kernel address space, same user address space sharing

*** inside kernel address space, at other user address space landing

\(^9\)(Lat.), “at the same place”
modes of **process switches** as to partial processor virtualisation:

* inside the same (user/kernel) address space, *ibidem*\(^9\) continuing

** inside kernel address space, same user address space sharing

*** inside kernel address space, at other user address space landing

\(^9\)(Lat.), “at the same place”
scheduling (Ger. *Ablaufplanung*) the *dispatching* (Ger. *Einlastung*) of processes or, to be precise, process incarnations.

- Scheduling involves planning the order in which processes are executed.
- Dispatching refers to the mechanism by which a process is selected for execution when its turn arrives.

These concepts are part of the larger field of operating systems, and they can be challenging to implement due to unpredictable dynamic system behavior and resource sharing, which can lead to asynchronism.

Sequencing of Processes

Scheduling v. Synchronisation
sequencing (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations

- a big theoretical/mathematical side of operating systems [2, 1, 6, 7]
- but enforcing the scheduling policies faces several practical challenges
Sequencing of Processes

- **scheduling** (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations
  - a big theoretical/mathematical side of operating systems [2, 1, 6, 7]
  - but enforcing the scheduling policies faces several practical challenges

- unpredictable dynamic system behaviour at run-time dashes hopes
  - on the one hand interrupts, on the other hand resource sharing
  - breeds **asynchronism** and, as a result, foregrounds **heuristic**
Sequencing of Processes

- **scheduling** (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations
  - a big theoretical/mathematical side of operating systems [2, 1, 6, 7]
  - but enforcing the scheduling policies faces several practical challenges
- unpredictable dynamic system behaviour at run-time dashes hopes
  - on the one hand interrupts, on the other hand resource sharing
  - breeds **asynchronism** and, as a result, foregrounds **heuristic**
- process **synchronisation** is notorious for producing interference
  - once it comes to *contention resolution*, which **implies sequencing**
    - blocking – in matters of allocating consumable and/or reusable resources
    - non-blocking – pertaining to indivisible machine (CPU) instructions
  - especially susceptible for inducing interference is **blocking synchronisation**
Sequencing of Processes

- **scheduling** (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations
  - a big theoretical/mathematical side of operating systems [2, 1, 6, 7]
  - but enforcing the scheduling policies faces several practical challenges
- unpredictable dynamic system behaviour at run-time dashes hopes
  - on the one hand interrupts, on the other hand resource sharing
  - breeds **asynchronism** and, as a result, foregrounds **heuristic**
- process **synchronisation** is notorious for producing interference
  - once it comes to **contention resolution**, which **implies sequencing**
    - blocking – in matters of allocating consumable and/or reusable resources
    - non-blocking – pertaining to indivisible machine (CPU) instructions
  - especially susceptible for inducing interference is **blocking synchronisation**
- to **control resource usage**, processes pass through logical states
  - whereby synchronisation emerges jointly responsible for state transitions
  - taken together, scheduling **and** synchronisation are **cross-cutting concerns**
Process States and State Transitions

- **typical life time cycle** of processes:
  - **ready**: ready to run, but still waiting for a processor (core)
  - **running**: executing on a processor (core), performing a CPU burst
  - **blocked**: waiting for an event (being in sync), performing an I/O burst
Process States and State Transitions

- *expecting resource allocation*

- **ready**
  - start
  - yield
  - cause

- **running**
  - seize
  - yield

- **blocked**
  - await

**typical life time cycle of processes:**

- **ready**
  - ready to run, but still waiting for a processor (core)

- **running**
  - executing on a processor (core), performing a CPU burst

- **blocked**
  - waiting for an event (being in sync), performing an I/O burst
Process States and State Transitions

- **ready**: ready to run, but still waiting for a processor (core)
- **running**: executing on a processor (core), performing a CPU burst
- **blocked**: waiting for an event (being in sync), performing an I/O burst

**(typical life time cycle)** of processes:

- **start**: initiates the process
- **seize**: acquires resources
- **yield**: releases resources
- **cause**: triggers state change
- **await**: waits for an event
- **cease**: terminates the process

**using allocated resources**

---

© wosch  CS (WS 2016, LEC 3)  Characteristics – Logical
Process States and State Transitions

- relevant resources:
  - processor
    - start
    - seize
    - yield
    - cease
  - signal
    - await
    - cause

- typical **life time cycle** of processes:
  - ready
    - ready to run, but still waiting for a processor (core)
  - running
    - executing on a processor (core), performing a CPU burst
  - blocked
    - waiting for an event (being in sync), performing an I/O burst
Process States and State Transitions

- **relevant resources:**
  - processor
    - start
    - seize
    - yield
    - cease
  - signal
    - await
    - cause
- **waitlists involved:**
  - ready list of runnable processes
  - blocked list of processes unable to run

- **typical life time cycle** of processes:
  - **ready**
    - ready to run, but still waiting for a processor (core)
  - **running**
    - executing on a processor (core), performing a CPU burst
  - **blocked**
    - waiting for an event (being in sync), performing an I/O burst
Outline

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
Résumé

- a process is **predetermined by a program** that is to be executed
  - the process inherits the static characteristics of its program
  - when being existent, the process adds dynamic characteristics
    - as a function of data processing and interaction with the environment
- a process may be **sequential or non-sequential** (as to its program)
  - that is to say, composed of non-overlapping or overlapping actions
  - whereby overlapping is caused by multiprocessing in a wider sense
    - real parallelism, but also pseudo-parallelism in its various forms
- processes are **parallel, concurrent, simultaneous, or interacting**
  - simultaneous processes comprise concurrent and interacting periods
  - each of these can be parallel on their part, i.e., if their actions overlap
    - by either multiplexing or multiplication of the necessary processing units
- as to implementation, processes may be **kernel or user threads**
  - regardless of which, logical states report on the life time cycle of a process
  - whereby synchronisation emerges jointly responsible for state transitions
    - taken together, scheduling **and** synchronisation need to be complementary
Résumé

- a process is **predetermined by a program** that is to be executed
  - the process inherits the static characteristics of its program
  - when being existent, the process adds dynamic characteristics
    - as a function of data processing and interaction with the environment

- a process may be **sequential or non-sequential** (as to its program)
  - that is to say, composed of non-overlapping or overlapping actions
  - whereby overlapping is caused by multiprocessing in a wider sense
    - real parallelism, but also pseudo-parallelism in its various forms

- processes are **parallel, concurrent, simultaneous, or interacting**
  - simultaneous processes comprise concurrent and interacting periods
  - each of these can be parallel on their part, i.e., if their actions overlap
    - by either multiplexing or multiplication of the necessary processing units

- as to implementation, processes may be **kernel or user threads**
  - regardless of which, logical states report on the life time cycle of a process
  - whereby synchronisation emerges jointly responsible for state transitions
    - taken together, scheduling and synchronisation need to be complementary
Résumé

- Un processus est **prédéterminé par un programme** qui doit être exécuté.
  - Le processus hérite des caractéristiques statiques de son programme.
  - Lorsqu’il existe, le processus ajoute des caractéristiques dynamiques
  - en fonction du traitement de données et de l’interaction avec l’environnement.

- Un processus peut être **séquentiel ou non-séquentiel** (selon son programme).
  - Cela signifie qu’il est composé de tâches non-ouvrantes ou superposées.
  - Cette superposition est due au multitâche dans un sens plus large.
    - L’parallelisme réel, mais aussi l’**pseudo-parallelisme** dans ses diverses formes.

- Les processus sont **parallèles, concurrents, simultanés, ou interactifs**.
  - Les processus simultanés comprennent des périodes concurrentes et interactives.
  - Chacun de ces processus peut être parallèle sur leur propre compte, c’est-à-dire, si leurs actions se superposent.
    - Par exemple, par multiplexage ou multiplication des unités de traitement nécessaires.

- En ce qui concerne l’implémentation, les processus peuvent être **kernels ou threads utilisateur**.
  - Indépendamment de ce qui se passe, les états logiques rapportent le cycle de vie d’un processus.
  - La synchronisation est conjointement responsable des transitions d’état.
    - Ensemble, le **scheduling** et la **synchronisation** doivent être complémentaires.
Résumé

- A process is **predetermined by a program** that is to be executed.
  - The process inherits the static characteristics of its program.
  - When being existent, the process adds dynamic characteristics:
    - As a function of data processing and interaction with the environment.

- A process may be **sequential or non-sequential** (as to its program):
  - That is to say, composed of non-overlapping or overlapping actions.
  - Whereby overlapping is caused by multiprocessing in a wider sense:
    - Real parallelism, but also pseudo-parallelism in its various forms.

- Processes are **parallel, concurrent, simultaneous, or interacting**:
  - Simultaneous processes comprise concurrent and interacting periods.
  - Each of these can be parallel on their part, i.e., if their actions overlap:
    - By either multiplexing or multiplication of the necessary processing units.

- As to implementation, processes may be **kernel or user threads**:
  - Regardless of which, logical states report on the life time cycle of a process.
  - Whereby synchronisation emerges jointly responsible for state transitions:
    - Taken together, scheduling **and** synchronisation need to be complementary.
a process is **predetermined by a program** that is to be executed
- the process inherits the static characteristics of its program
- when being existent, the process adds dynamic characteristics
  - as a function of data processing and interaction with the environment

a process may be **sequential or non-sequential** (as to its program)
- that is to say, composed of non-overlapping or overlapping actions
- whereby overlapping is caused by multiprocessing in a wider sense
  - real parallelism, but also pseudo-parallelism in its various forms

processes are **parallel, concurrent, simultaneous, or interacting**
- simultaneous processes comprise concurrent and interacting periods
- each of these can be parallel on their part, i.e., if their actions overlap
  - by either multiplexing or multiplication of the necessary processing units

as to implementation, processes may be **kernel or user threads**
- regardless of which, logical states report on the life time cycle of a process
- whereby synchronisation emerges jointly responsible for state transitions
  - taken together, scheduling *and* synchronisation need to be complementary
Reference List I


Multilevel Machines.
In: Structured Computer Organization.
Prentice-Hall, Inc., 1979. –

[12] Wikipedia:
Process.
Process “particularly, describes the formal notice or writ used by a court to exercise jurisdiction over a person or property”

- analogy in computer science or operating-system concepts, resp.:
  - **writ**: order to abandon rivalry\(^\text{10}\) in the claiming of resources
  - direction to resolve competition of resource contenders
  - **court**: incarnation of the function of scheduling or coordination
  - point of synchronisation in a program
  - **jurisdiction**: sphere of authority of contention resolution
  - zone of influence of the synchronisation policy
  - **property**: occupancy/ownership of resources, ability to proceed
  - functional or non-functional attribute

- generally, the action or trial, resp., follows a hierarchical jurisdiction
  - thereby, the process step related to a certain level is denoted as *instance*
    - in informatics, translation to (Ger.) “Instanz” however was rather unapt
  - operating systems often command a multi-level processing of processes

\(^\text{10}\)Lat. *rivalis* “in the use of a watercourse co-authored by a neighbour”
refinement of [11, p. 5]: levels present on today's computers
- right, the method and (bracketed) program that supports each level