Concurrent Systems

Nebenläufige Systeme

III. Processes

Wolfgang Schröder-Preikschat

November 19, 2020
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

process ■ a program in execution

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
Subject Matter

- discussion on **abstract concepts** as to multiplexing machines:
  - **program** - concretized form of an algorithm

  - **process** - a program in execution

- explanation of **process characteristics** in physical and logical terms

- a **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.

1 A program (also) describes the kind of flow (Ger. *Ablauf*) of a process. Sequential actions proceed deterministically, the result is determinate. Parallel actions are 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] 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\(^1\)

\(^1\)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.

- 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.
Operating systems bring programs to execution by creation, releasing, controlling and timing of processes

- in computer sciences, a process is unimaginable without a program

- A program (also) describes the kind of flow (Ger. *Ablauf*) of a process
  - sequential: a sequence of temporally non-overlapping actions
  - parallel: non-sequential

- in both kinds does the program flow consist of **actions** (p. 7 ff.)
Operating systems bring programs to execution by creation, releasing, controlling and timing of processes.

- In computer sciences, a process is unimaginable without a program.

- A program (also) describes the kind of flow (Ger. *Ablauf*) of a process.

---

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]
Outline

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
Program I

Definition
For a certain machine concretised form of an algorithm.
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)++;
}
```
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
- before assembly

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

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

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

- three actions (lines 7–9)

---

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

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

three actions (lines 7–9)

Definition (Action)

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

\(^2\)gcc -O4 -m32 -static -fomit-frame-pointer -S, also below
address space and virtual machine SMC\(^3\)
- text segment
- Linux

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x080482f0:</td>
<td>mov 0x4(%esp),%eax</td>
</tr>
<tr>
<td>0x080482f4:</td>
<td>add $0x1,(%eax)</td>
</tr>
<tr>
<td>0x080482f7:</td>
<td>adc $0x0,0x4(%eax)</td>
</tr>
<tr>
<td>0x080482fb:</td>
<td>ret</td>
</tr>
</tbody>
</table>

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

- text segment
- Linux

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

- after linking/binding and before loading

real machine

- after loading
- executable

\begin{align*}
\texttt{8b 44 24 04} \\
\texttt{83 00 01} \\
\texttt{83 50 04 00} \\
\texttt{c3}
\end{align*}

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

\textsuperscript{3}symbolic machine code: x86 + Linux.
Program II

address space and virtual machine SMC\(^3\)

- text segment
- Linux

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

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

real machine

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

\begin{verbatim}
8b 44 24 04
83 00 01
83 50 04 00
c3
\end{verbatim}

Hint (\texttt{ret} or \texttt{c3}, resp.)

The action for a subroutine return corresponds to the action of the corresponding subroutine call (\texttt{gdb, disas /rm main}):

1. \texttt{movl $0x80d37b0,(\%esp)}
2. \texttt{call 0x80482f0 <inc64>}

\(^3\)symbolic machine code: x86 + Linux.
Definition

A program $P$ that allows several execution threads\(^a\) in $P$ itself.

\(^a\)Any kind of program thread, coroutines, signal/interrupt handlers.
A program $P$ that allows several execution threads\textsuperscript{a} in $P$ itself.

\textsuperscript{a}Any kind of program thread, coroutines, signal/interrupt handlers.

an excerpt of $P$ using the example of POSIX Threads\cite{4}:

\begin{verbatim}
pthread_t tid;

if (!pthread_create(&tid, NULL, thread, NULL)) {
    /* ... */
    pthread_join(tid, NULL);
}
\end{verbatim}
Non-Sequential Program I

Definition

A program $P$ that allows several execution threads\(^a\) in $P$ itself.

\(^a\)Any kind of program thread, coroutines, signal/interrupt handlers.

an excerpt of $P$ using the example of \textit{POSIX Threads} [4]:

```
1 pthread_t tid;

2 if (!pthread_create(&tid, NULL, thread, NULL)) {
3    /* ... */
4        pthread_join(tid, NULL);
5    }
```

the parallel flow allowed in $P$ itself:

```
7 void *thread(void *null) {
8    /* ... */
9        pthread_exit(NULL);
10    }
```
Non-Sequential Program I

Definition

A program $P$ that allows several execution threads\(^a\) in $P$ itself.

\(^a\)Any kind of program thread, coroutines, signal/interrupt handlers.

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);
}
```

Hint

*It is not mandatory that these threads of execution must take place simultaneously!*)
actions of parallelism—but *sequential flows* of the same program:

```c
pid_t pid;
if (!(pid = fork())) {
    /* ... */
    exit(0);
}
wait(NULL);
```
Non-Sequential Program II

actions of parallelism—but **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 $A'$
actions of parallelism—but **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

actions of parallelism—but **sequential flows** of the same program:

```c
pid_t pid;

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

wait(NULL);
```

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

actions of parallelism—but **sequential flows** of the same program:

```c
pid_t pid;

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

the shown actions cause parallel flows within an operating system

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

4The exception (strictly cooperative systems) proves the rule.
Multiprocessing of Sequential Programs

address space A

directions

  fork()

  wait(NULL)
Multiprocessing of Sequential Programs

address space A

parent

fork()

wait(NULL)

address space A’

child

duplicate

/* ... */

exit(0)
Multiprocessing of Sequential Programs

address space A

parent

duplicate

fork()
wait(NULL)

address space A’

child

/* ... */
exit(0)

sequential program flows

processor (core) characteristic:

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

Multi; 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 OS

operating system

duplicate

address space A'

child

/* ... */
exit(0)

sequential program flows

non-sequential program flow
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:
  - sequential program flows

- Non-sequential program flow:
  - 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
Multiprocessing of Sequential Programs

- **address space A**
  - **parent**
  - `fork()`
  - `wait(NULL)`

- **address space A'**
  - **child**
  - `/* ... */`
  - `exit(0)`

- **address space OS**
  - operating system

Non-sequential program flow

Sequential program flows

**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
  - **Multi**: ditto; but also event-based operating system

- Both cause **parallel processes** (p. 16) within the operating system
Definition (Program flow)

A program in execution.
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$_5$ → program statement
    - level$_4$ → assembly mnemonic
    - level$_3$ → machine instruction
    - level$_2$ → microprogram directive

- its actions are not imperatively indivisible (atomic)
- this particularly holds both for the abstract (virtual) and real processor
- this sequence is static (passive), while a process is dynamic (active)

**Hint (Process ≠ 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)
    - $\text{level}_5 \rightarrow \text{program statement} \geq 1 \text{assembly mnemonics}$
    - $\text{level}_4 \rightarrow \text{assembly mnemonic} \geq 1 \text{machine instructions}$
    - $\text{level}_3 \rightarrow \text{machine instruction} \geq 1 \text{microprogram directives}$
    - $\text{level}_2 \rightarrow \text{microprogram directive}$

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

**Definition (Program flow)**

A program in execution.

- the program specifies a sequence of actions that are to be executed
- this sequence is static (passive), 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

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

Hint (Process ≠ Process instance)

A process instance (Ger. Exemplar) is an 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>(^*) read from memory into accumulator (</td>
</tr>
</tbody>
</table>

- typical for a complex instruction of an “abstract processor” (C, CISC)

© wosch, thoenig CS (WS 2020/21, LEC 3) Fundamentals – Process 13
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 *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)

---

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

- **Indivisibility II**

- The essential non-functional property of an \textit{atomic operation}\(^5\)

- Examples of (critical) actions for incrementation of a counter variable:
  - Level\(^5\)\(\rightarrow\)3

<table>
<thead>
<tr>
<th>C/C++</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i++;</code></td>
<td><code>movl i, %eax</code></td>
</tr>
<tr>
<td><code>addl $1, %eax</code></td>
<td></td>
</tr>
<tr>
<td><code>movl %eax, i</code></td>
<td></td>
</tr>
</tbody>
</table>

---

\(^5\) from (Gr.) \textit{átomo} “indivisible”.

© wosch, thoenig CS (WS 2020/21, LEC 3) Fundamentals – Process
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*\(^5\)

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

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

- **level\(_3\) \rightarrow 2**
  - **ASM**
  
  ```
  incl i
  ```
  - **ISA**
  
  ```
  read A from <i>
  ```
  ```
  modify A by 1
  ```
  ```
  write A to <i>
  ```

\(^5\) from (Gr.) *átomo* “indivisble”.

© wosch, thoenig  CS (WS 2020/21, LEC 3)  Fundamentals – Process 14
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

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

<table>
<thead>
<tr>
<th>C/C++</th>
<th>ASM</th>
<th>ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>i++;</td>
<td>movl i, %eax</td>
<td>incl i, 0</td>
</tr>
<tr>
<td></td>
<td>addl $1, %eax</td>
<td>read A from &lt;i&gt;</td>
</tr>
<tr>
<td></td>
<td>movl %eax, i</td>
<td>modify A by 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write A to &lt;i&gt;</td>
</tr>
</tbody>
</table>

Points (i++, incl) in case of merely conditionally atomic execution namely uninterruptible operation (level $5 \mapsto 3$), uniprocessor (level $3 \mapsto 2$)

Problem: overlapping in time of the sequence of actions pointed here

\[^5\text{from (Gr.) átomo “indivisible”}\]
Definition (Sequential program in execution)
A process with only a single thread of execution.
Definition (Sequential program in execution)

A process with only a **single thread of execution**.

- A sequence of actions that forms a **unique execution thread**
  - but which may develop differently with each restart of that process
    - other input data, program change, ..., transient hardware errors
Sequential Process

Definition (Sequential program in execution)
A process with only a single thread of execution.

- a sequence of actions that forms a **unique execution thread**
- the sequence is defined by a **total order** of its actions
  - it is reproducible given unmodified original conditions
    - same input data, no program changes, ..., no transient hardware errors
Sequential Process

Definition (Sequential program in execution)
A process with only a single thread of execution.

- a sequence of actions that forms a unique execution thread
- the sequence is defined by a total order of its actions

Hint (Execution Thread $\neq$ 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.
Definition (Non-sequential program in execution)

A process consisting of several threads of execution, which may take place simultaneously (in parallel).
Non-Sequential Process

**Definition (Non-sequential program in execution)**

A process consisting of several threads of execution, which may take place simultaneously (in parallel).

- 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) or process (signal).
Non-Sequential Process

Definition (Non-sequential program in execution)
A process consisting of several threads of execution, which may take place simultaneously (in parallel).

- requirement is a non-sequential program (cf. p. 9)

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

Definition (Non-sequential program in execution)
A process consisting of several threads of execution, which may take place simultaneously (in parallel).

- requirement is a **non-sequential program** (cf. p. 9)
- whereby sequences of actions may overlap in the first place:
  - i. asynchronous program interrupts
  - ii. multithreading (Ger. *simultane Mehrfäigkeit*)
- consequently, the sequence of **all** actions is defined by a **partial order**
  - as external processes may enable temporal/causal independent actions
Concurrent Processes

Definition (in a broader sense: “simultaneous processes”)
Several threads of execution of the same non-sequential process or of multiple sequential processes taking place simultaneously.
Concurrent Processes

(Ger.) gleichzeitige Prozesse [3]

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

Several threads of execution of the same non-sequential process or of multiple sequential processes taking place simultaneously.

“concurrent” only with respect to the same level of abstraction [10]

- 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

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

© wosch, thoenig
CS (WS 2020/21, LEC 3) Fundamentals – Process
Concurrent Processes

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

Several threads of execution of the same non-sequential process or of multiple sequential processes taking place simultaneously.

- “**concurrent**” only with respect to the same level of abstraction [10]

- however, to proceed, these processes compete for **reusable resources**
  - they share the processor (core), cache (line), bus, or devices
  - this also results in **interference**\(^7\) (Ger. *Interferenz*) in process behaviour

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

© wosch, thoenig  CS (WS 2020/21, LEC 3)  Fundamentals – Process
Concurrent Processes

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

Several threads of execution of the same non-sequential process or of multiple sequential processes taking place simultaneously.

- "concurrent" only with respect to the same level of abstraction [10]

- however, to proceed, these processes compete for reusable resources

- 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

[Ger.] gleichzeitige Prozesse [3]
Definition (also: “depending processes”)

Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared 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 printer, mouse, plotter, keyboard.
Interacting Processes I

Definition (also: “depending processes”)

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

- 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

8 printer, mouse, plotter, keyboard.
Interacting Processes I

Definition (also: “depending processes”)
Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

- this may emerge as a race condition (Ger. Wettlaufsitation)
- 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
Definition (also: “depending processes”)  
Simultaneous processes that, directly or indirectly, interact with each other through a shared variable or by accessing a shared resource.

- this may emerge as a **race condition** (Ger. *Wettlaufsituation*)

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

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

- this may emerge as a **race condition** (Ger. *Wettlaufsituation*)
- conflicts are eliminated by means of **synchronisation methods**
- founds **coordination** of cooperation and competition of processes

---

8 printer, mouse, plotter, keyboard.
# Race Conditions

```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

In case `thread_worker` exists in several identical incarnations, the value printed by `thread_minder` will not be the same, as the cycle variable is incremented by each worker thread independently.
```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)?
```

- 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*)?
- 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
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:
    movl 4(%esp), %eax
    addl $1, (%eax)
    adcl $0, 4(%eax)
    ret

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

`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

```c
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

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

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

Assume \( cycle = 2^{32} - 1 \).
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

```
movl cycle+4, %edx ; high &
movl cycle, %eax ; low word
movl $.LC0, (%esp)
movl %edx, 8(%esp)
movl %eax, 4(%esp)
call printf
```

- assume `cycle = 2^{32} - 1`
- `inc64` overlaps actions 10–11
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:
   movl 4(%esp), %eax
   addl $1, (%eax)
   adcl $0, 4(%eax)
   ret

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

- 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 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

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

real parallel actions: (multi-core) multiprocessor

case 4.2.1
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
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**

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

**GCC 4.2.1, MacOSX**

```assembly
_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

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

GCC 4.2.1, MacOSX

```assembly
_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)

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)

**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)

**real parallel actions:** (multi-core) multiprocessor

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

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary

© wosch, thoenig CS (WS 2020/21, LEC 3) Characteristics
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

```c
void mutex_inc64(int64_t *i, pthread_mutex_t *lock) {
    pthread_mutex_lock(lock);
    /* indivisible, now */
    inc64(i);
    /* reuse code @ p. */
    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. */
    asm("lock incq %0" : : "m" (*i) : "memory");
}
```

anywhere applicable and by orders of magnitude more efficient solution
Consistency

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

**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**

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

- blocking of interacting processes: comparatively long time span

```c
1 void mutex_inc64(int64_t *i, pthread_mutex_t *lock) {
2    pthread_mutex_lock(lock);   /* indivisible, now */
3    inc64(i);                   /* reuse code @ p. 7 */
4    pthread_mutex_unlock(lock);  /* divisible, again */
5 }
```
Consistency

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

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

reducing to a 64-bit ELOP of the real processor

```c
void inc64(int64_t *i) {
    asm ("lock incq %0" : : "m" (*i) : "memory");
}
```

anywhere applicable and by orders of magnitude more efficient solution
Localisation

- anchoring of processes can be different within a computing system
Localisation

- 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
    - outside – optional, within run-time or even application 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
  
- 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”
anchoring of processes can be different within a computing system

- **inside** originally, within the operating system or its kernel

- **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

- anchoring of processes can be different within a computing system

- 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

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

- 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
  - namely inside or outside the operating-system machine level:
    - **inside** – originally, within the operating system or its kernel
      
      $\rightarrow$ "kernel thread", in computer science folklore

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

- operating systems are aware only of their own “first-class citizens”
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*\(^9\) 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”
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”
Sequencing of Processes

Scheduling v. Synchronisation

- **scheduling** (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations.
Sequencing of Processes

Scheduling v. Synchronisation

- **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
Sequencing of Processes

Scheduling v. Synchronisation

- **scheduling** (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations

- 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

- unpredictable dynamic system behaviour at run-time dashes hopes

- 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 v. Synchronisation

- **scheduling** (Ger. *Ablaufplanung*) the **dispatching** (Ger. *Einlastung*) of processes or, to be precise, process incarnations

- unpredictable dynamic system behaviour at run-time dashes hopes

- process **synchronisation** is notorious for producing interference

- **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

#### Relevant resources:
- Processor

#### States and Transitions:
- **Start**
- **Seize**
- **Yield**
- **Cease**
- **Cause**
- **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
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
typical \textbf{life time cycle} of processes:

- \textbf{ready}  
  - ready to run, but still waiting for a processor (core)

- \textbf{running}  
  - executing on a processor (core), performing a CPU burst

- \textbf{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

- 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
Résumé

A process is predetermined by a program that is to be executed. When being existent, the process inherits the static characteristics of its program. 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
A process is predetermined by a program that is to be executed. The process inherits the static characteristics of its program when being existent, and 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.

© wosch, thoenig  CS (WS 2020/21, LEC 3)  Summary
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.

- A process may be **sequential or non-sequential** (as to its program).

- Processes are **parallel, concurrent, simultaneous, or interacting**.

- As to implementation, processes may be **kernel or user threads**.


Originally as a Concept of Law

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[^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 unept
    - operating systems often command a multi-level processing of processes

[^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

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>digital logic level</td>
<td>execution</td>
</tr>
<tr>
<td>1</td>
<td>microarchitecture level</td>
<td>interpretation (microprogram) or execution</td>
</tr>
<tr>
<td>2</td>
<td>instruction set architecture level</td>
<td>translation (microprogram) or execution</td>
</tr>
<tr>
<td>3</td>
<td>operating-system machine level</td>
<td>partial interpretation (operating system)</td>
</tr>
<tr>
<td>4</td>
<td>assembly language level</td>
<td>translation (assembler) and binding (linker)</td>
</tr>
<tr>
<td>5</td>
<td>problem-oriented language level</td>
<td>translation (compiler)</td>
</tr>
</tbody>
</table>

structured computer organisation

multilevel machines [5]

© wosch, thoenig CS (WS 2020/21, LEC 3) Addendum – Process