Concurrent Systems
*Nebenläufige Systeme*

III. Processes

Wolfgang Schröder-Preikschat

October 23, 2014

Subject Matter

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

- 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

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

1Provided that the operating system offers all necessary commands.

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]

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

```c
#include <stdint.h>

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

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

**Definition (Action)**

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

```c
#include <stdint.h>

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

Program II

Operating-System Machine Level

- address space and virtual machine SMC
  - text segment
  - Linux
- real machine
  - after linking/binding and
  - before loading
- virtual machine SMC
  - after linking/binding and
  - before loading

```c
mov 0x4(%esp),%eax
add $0x1,(%eax)
adc $0x0,0x4(%eax)
ret
```

- 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, disas /rm main*):

```c
mov 0x804837b0,(%esp)
call 0x8048f40
```

© wosch CS (WS 2014, LEC 3) Preface 5–34
© wosch CS (WS 2014, LEC 3) Preface 6–34
© wosch CS (WS 2014, LEC 3) Fundamentals 7–34
© wosch CS (WS 2014, LEC 3) Fundamentals 8–34
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
1 pthread_t tid;
2 if (!pthread_create(&tid, NULL, thread, NULL)) {
3     /* ... */
4     pthread_join(tid, NULL);
5 }
6
7 void *thread(void *null) {
8     /* ... */
9     pthread_exit(NULL);
10 }
```

the parallel flow allowed in $P$ itself:

```c
7 void *thread(void *null) {
8     /* ... */
9     pthread_exit(NULL);
10 }
```

Non-Sequential Program II

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

```c
1 pid_t pid;
2 if(!(pid = fork())) {
3     /* ... */
4     exit(0);
5 }
6
7 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.

Multiprocessing of Sequential Programs

address space $A$

parent

```
fork()
wait(NULL)
/* ... */
exit(0)
```

child

address space $A'$

duplicate

sequential program flows

address space OS

operating system

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

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

Process

Definition (Program flow)
A program in execution on and through a processor.

- 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.$^3$

$^3$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
- an 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>addl $1, %r*</td>
<td>movl %r, i</td>
</tr>
</tbody>
</table>

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

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.

- 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 Mehrfädigkeit), in fact:
      - pseudo-parallel – multiplex mode of a single processor (core)
      - real parallel – parallel mode of a (multi-core) multiprocessor
    - ii asynchronous program interrupts
  - consequently, the sequence of all actions is defined by a partial order
    - as external processes may enable temporal/causal independent actions

5 from (Gr.) átomo “indivisible”.

6 Interrupt requests issued by some device (IRQ) oder process (signal).
**Concurrent Processes (Ger.) gleichzeitige Prozesse [3]**

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

One or more non-sequential processes in which at least two 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 to proceed, concurrent processes compete for **reusable resources**
  - they share the processor (core), cache (line), bus, or devices
  - outcome of this is **interference** (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

**Interacting Processes II**

**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*

- 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

**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
  - 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 non-blocking reducing
    - prevent from executing an intended sequence of actions
    - let a process abort and retry a started sequence of actions
    - replace a sequence of actions by an atomic instruction

- founds **coordination** of cooperation and competition of processes

**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**
  ```c
  inc64:
  movl 4(%esp), %eax
  addl $1, (%eax)
  addcl $0, 4(%eax)
  ret
  
  .L6:
  movl $cycle, (%esp)
  call inc64
  jmp .L6
  ```

- **minder thread**
  ```c
  movl cycle+4, %edx ; high &
  movl cycle, %eax ; low word
  movl $.L00, (%esp)
  movl %edx, 8(%esp)
  movl %eax, 4(%esp)
  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
Interacting Processes IV

2. Race Condition

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

- **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

**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 */
}
```

- 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");
}
```

Localisation

Operating-System v. Application Context

- **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)
    - "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
  
- operating systems are aware only of their own “first-class citizens”
Weight Category

Interruption and Resumption Overhead

* feather-, ** light-, *** heavy-weight

* partial virtualization

---

Sequencing of Processes

Scheduling vs. 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
  - 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**

---

Outline

Preface
Fundamentals
Program
Process
Characteristics
Physical
Logical
Summary

---

Process States and State Transitions

<table>
<thead>
<tr>
<th>State</th>
<th>Relevant Resources:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>processor: start, seize, yield, cease</td>
</tr>
<tr>
<td>running</td>
<td>signal: yield, cease, await, cause</td>
</tr>
<tr>
<td>blocked</td>
<td>waitlists involved: ready list of runnable processes, blocked list of processes unable to run</td>
</tr>
</tbody>
</table>

- 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
- 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

Summary

- **Résumé**
- **Reference List I**
  - [1] **Coffman, E. G. ; Denning, P. J.:** Operating System Theory.
    Prentice Hall, Inc., 1973
    Addison-Wesley, 1967
  - [3] **Hansen, P. B.:** Concurrent Processes.
    In: Operating System Principles.
  - [4] **IEEE:**
    POSIX.1c Threads Extensions / Institute of Electrical and Electronics Engineers.
  - [5] **Kleinöder, J. ; Schröder-Preikschat, W. :**
    Rechnerorganisation.
    In: Lehrstuhl Informatik 4 (Hrsg.): Systemprogrammierung.
    FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien), Kapitel 5
- **Reference List II**
  - [6] **Kleinrock, L. :**
    John Wiley & Sons, 1975
  - [7] **Liu, J. W. S. :**
    Real-Time Systems.
  - [8] **Löhr, K.-P. :**
    Nichtsequentielle Programmierung.
    In: Institut für Informatik (Hrsg.): Algorithmen und Programmierung IV.
    Freie Universität Berlin, 2006 (Vorlesungsfolien)
  - [9] **Neufeldt, V. (Hrsg.) ; Guralnik, D. A. (Hrsg.):**
    Webster’s New World Dictionary.
    Simon & Schuster, Inc., 1988
  - [10] **Schröder-Preikschat, W. :**
    Concurrency.
    In: Lehrstuhl Informatik 4 (Hrsg.): Concurrent Systems.
    FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 2
- **Reference List III**
  - [11] **Tanenbaum, A. S. :**
    Multilevel Machines.
    In: Structured Computer Organization.
  - [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
  - court: incarnation of the function of scheduling or coordination
  - jurisdiction: sphere of authority of contention resolution
  - property: occupancy/ownership of resources, ability to proceed
- 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

\(^\text{10}\)Lat. rivalis “in the use of a watercourse co-authored by a neighbour”

---

Structured Computer Organisation

Multilevel Machines [5]

- refinement of [11, p.5]: levels present on todays computers
- right, the method and (bracketed) program that supports each level

---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>digital logic level</td>
<td>microarchitecture level</td>
<td>instruction set architecture level</td>
<td>operating-system machine level</td>
<td>assembly language level</td>
<td>problem-oriented language level</td>
</tr>
</tbody>
</table>

- partial interpretation (operating system)
- translation (assembler) and binding (linker)
- translation (compiler)
- interpretation (microprogram) or execution
- execution

---

© wosch CS (WS 2014, LEC 3) Addendum – Process