Concurrent Systems

Nebenläufige Systeme

III. Processes

Wolfgang Schröder-Preikschat

November 19, 2020
Agenda

Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
Preface

Fundamentals
  Program
  Process

Characteristics
  Physical
  Logical

Summary
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.)

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]

---

1Provided that the operating system offers all necessary commands.
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
- before assembly

```asm
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.

\[\text{gcc -O4 -m32 -static -fomit-frame-pointer -S, also below}\]
address space and virtual machine SMC³

text segment

Linux

after linking/binding and
before loading

real machine

after loading
executable

moving 0x4(%esp),%eax
adding $0x1,(%eax)
adding $0x0,0x4(%eax)
returning

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):

moving $0x80d37b0,(%esp)
calling 0x80482f0 <inc64>

³symbolic machine code: x86 + Linux.
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]:

1. `pthread_t tid;

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

the parallel flow allowed in $P$ itself:

4. `void *thread(void *null) {
   /* ... */
   5. pthread_exit(NULL);
   }`  

Hint

It is not mandatory that these threads of execution must take place simultaneously!
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 $A'$
- 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”

---

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

- 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 (passive), while a process is dynamic (active)

**Hint (Process \(\neq\) 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
- 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>
<tr>
<td></td>
<td></td>
<td>modify contents of accumulator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write 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)

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:

\[ \text{level}_5 \rightarrow 3 \quad \text{level}_3 \rightarrow 2 \]

<table>
<thead>
<tr>
<th>C/C++</th>
<th>ASM</th>
<th>ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 i++; 2 movl i, %eax</td>
<td>5 incl i</td>
<td>6 read A from &lt;i&gt;</td>
</tr>
<tr>
<td>3 addl $1, %eax</td>
<td>7 modify A by 1</td>
<td></td>
</tr>
<tr>
<td>4 movl %eax, i</td>
<td>8 write A to &lt;i&gt;</td>
<td></td>
</tr>
</tbody>
</table>

points \((i++, \text{incl})\) in case of merely conditionally atomic execution

- namely uninterruptible operation \((\text{level}_5 \rightarrow 3)\), uniprocessor \((\text{level}_3 \rightarrow 2)\)
- problem: overlapping in time of the sequence of actions pointed here

\(^5\text{from (Gr.) átomo “indivisible”}\).
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**
  - 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
    - same input data, no program changes, . . . , no transient hardware errors

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 (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\(^6\)

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

- 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) or process (signal).
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]
  - 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
  - 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
- 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 occupies 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** replace a sequence of actions by an atomic instruction

- Founds **coordination** of cooperation and competition of processes
```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

inc64: see p. 7
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

```
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
  - 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)

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

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

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");
}
```

anywhere applicable and by orders of magnitude more efficient solution
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

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

\(^9\)(Lat.), “at the same place”
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

- **Expecting resource allocation**
  - **start**
  - **seize**
  - **cause**
  - **await**

- **Using allocated resources**
  - **yield**
  - **cease**

**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.
[1] **Coffman, E. G. ; Denning, P. J.**:
*Operating System Theory.*
Prentice Hall, Inc., 1973

[2] **Conway, R. W. ; Maxwell, L. W. ; Millner, L. W.**:
*Theory of Scheduling.*
Addison-Wesley, 1967

[3] **Hansen, P. B.**:
Concurrent Processes.
*In: Operating System Principles.*
ISBN 0–13–637843–9, Kapitel 3, S. 55–131

[4] **IEEE**:
POSIX.1c Threads Extensions / Institute of Electrical and Electronics Engineers.
Standardddokument

[5] **Kleinöder, J. ; Schröder-Prekischat, W.**:
Rechnerorganisation.
*In: Lehrstuhl Informatik 4 (Hrsg.): Systemprogrammierung.*
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien), Kapitel 5
John Wiley & Sons, 1975


In: Institut für Informatik (Hrsg.): *Algorithmen und Programmierung IV*. Freie Universität Berlin, 2006 (Vorlesungsfolien)

Simon & Schuster, Inc., 1988

In: Lehrstuhl Informatik 4 (Hrsg.): *Concurrent Systems*. FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 2
In: Structured Computer Organization.

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 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