

Concurrent Systems

Nebenläufige Systeme

III. Processes

Wolfgang Schröder-Preikschat

November 13, 2019



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
- 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 processes¹
- 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.



Outline

Preface

Fundamentals

Program

Process

Characteristics

Physical

Logical

Summary



Definition

For a certain machine concretised form of an algorithm.

- virtual machine C
 - after editing and
 - before compilation
- virtual machine ASM (x86)
 - after compilation²and
 - before assembly

```

1 #include <stdint.h>
2
3 void inc64(int64_t *i) {
4     (*i)++;
5 }
```

```

6 inc64:
7     movl 4(%esp), %eax
8     addl $1, (%eax)
9     adcl $0, 4(%eax)
10    ret
```

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

Definition (Action)

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

²gcc -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
- | | | | | |
|---|-------------|----------------------|--|-------------|
| 1 | 0x080482f0: | mov 0x4(%esp),%eax | | 8b 44 24 04 |
| 2 | 0x080482f4: | add \$0x1, (%eax) | | 83 00 01 |
| 3 | 0x080482f7: | adc \$0x0, 0x4(%eax) | | 83 50 04 00 |
| 4 | 0x080482fb: | ret | | c3 |
- same number of actions (lines 1–3, resp.), but different forms of representation

Hint (ret or c3, resp.)

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

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

³symbolic machine code: x86 + Linux.

Non-Sequential Program I

Definition

A program P that allows **several execution threads**^a in P itself.

^aAny kind of program thread, coroutines, signal/interrupt handlers.

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

```
1 pthread_t tid;
2
3 if (!pthread_create(&tid, NULL, thread, NULL)) {
4     /* ... */
5     pthread_join(tid, NULL);
6 }
```

- the parallel flow allowed in P itself:

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

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:

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

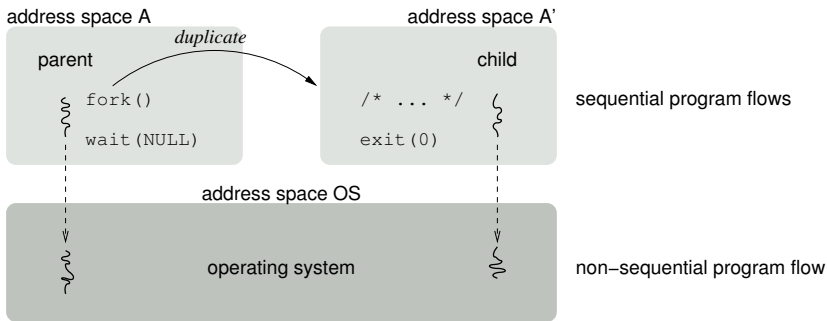
- sequential flows can establish parallel ones within a domain that logically comprises those sequential flows

- 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”⁴

⁴The 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



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)
 - level₅ \mapsto program statement ≥ 1 assembly mnemonics
 - level₄ \mapsto assembly mnemonic ≥ 1 machine instructions
 - level₃ \mapsto machine instruction ≥ 1 microprogram directives
 - level₂ \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

^aJust as an object is a “core image” of a class.



Indivisibility I

Definition

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

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

level	action	sequence of actions
5	<code>i++</code>	
4-3	<code>incl i*</code>	<code>movl i,%r</code> <code>addl \$1,%r*</code>
	<code>addl \$1,i*</code>	<code>movl %r,i</code>
2-1		* <i>read</i> from memory into accumulator <i>modify</i> contents of accumulator <i>write</i> from accumulator into memory

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



Indivisibility II

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

- an/the essential non-functional property of an **atomic operation**⁵
 - logical togetherness of a sequence of actions in terms of time
 - by what that sequence appears as **elementary operation** (ELOP)
- examples of (critical) actions for incrementation of a counter variable:
 - level $5 \mapsto 3$

C/C++	ASM
1 <code>i++;</code>	2 <code>movl i, %eax</code>
	3 <code>addl \$1, %eax</code>
	4 <code>movl %eax, i</code>
 - level $3 \mapsto 2$

ASM	ISA
5 <code>incl i</code>	6 <i>read A from <i></i>
	7 <i>modify A by 1</i>
	8 <i>write A to <i></i>
- 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

⁵from (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 \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.



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 processes⁶
- whereby sequences of actions may overlap in the first place:
 - i asynchronous program interrupts
 - ii multithreading (Ger. *simultane Mehrfädigkeit*), 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

⁶Interrupt requests issued by some device (IRQ) or process (signal).

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

⁷Derived 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 resource⁸ (**resource type**)
- this may emerge as a **race condition** (Ger. *Wettlaufsituation*)
 - for shared variables or (reusable/consumable) resources, resp.
 - for starting or finishing an intended sequence of actions
- conflicts are eliminated by means of **synchronisation methods**:
 - blocking** ■ prevent from executing an intended sequence of actions
 - non-blocking** ■ let a process abort and retry a started sequence of actions
 - reducing** ■ replace a sequence of actions by an atomic instruction
- founds **coordination** of cooperation and competition of processes

⁸printer, mouse, plotter, keyboard.



```
1 int64_t cycle = 0;
2
3 void *thread_worker(void *null) {
4     for (;;) {
5         /* ... */
6         inc64(&cycle);
7     }
8 }
9
10 void *thread_minder(void *null) {
11     for (;;) {
12         printf("worker cycle %lld\n", cycle);
13         pthread_yield();
14     }
15 }
```

■ 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



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

- worker thread

```

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

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

```

- minder thread

```

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

```

- assume $cycle = 2^{32} - 1$

- `inc64` overlaps actions 10–11
- then, `edx = 0` and `eax = 0`
- effect is, `printf` displays 0
 - not 2^{32} , as would have been right



- assuming that the 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

```

- **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

- 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

```



Outline

Preface

Fundamentals

Program

Process

Characteristics

Physical

Logical

Summary



- 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**

```
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 }
```

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

```
6 void inc64(int64_t *i) {          /* renew code @ p.7 */
7     asm ("lock incq %0" : : "m" (*i) : "memory");
8 }
```

- anywhere applicable and by orders of magnitude more efficient solution

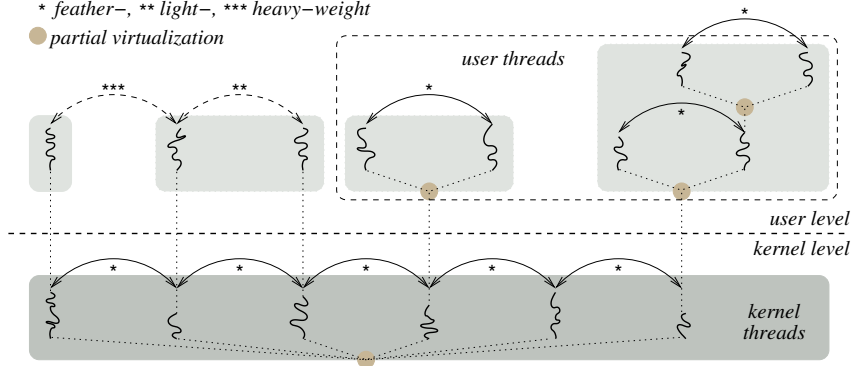


- **anchoring** of processes can be different within a computing system
 - namely inside or outside the operating-system machine level:
 - inside** – originally, within the operating system or its kernel
 - incarnation of the process is root of possibly other processes
 - partial virtualisation of the 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”



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

● partial virtualization



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

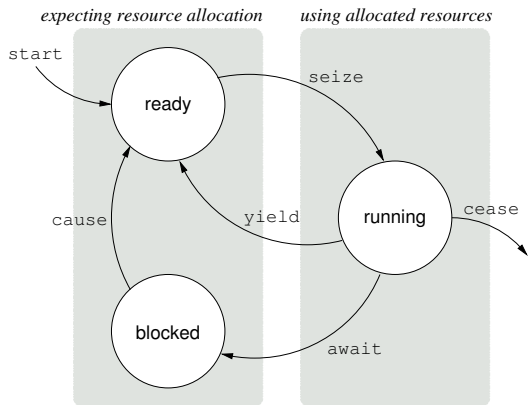
⁹(Lat.), "at the same place"



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



■ 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



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



Reference List I

- [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*.
Englewood Cliffs, N.J., USA : Prentice-Hall, Inc., 1973. –
ISBN 0-13-637843-9, Kapitel 3, S. 55-131
- [4] IEEE:
POSIX.1c Threads Extensions / Institute of Electrical and Electronics Engineers.
New York, NY, USA, 1995 (IEEE Std 1003.1c-1995). –
Standarddokument
- [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. :
Queuing Systems. Bd. I: Theory.
John Wiley & Sons, 1975
- [7] LIU, J. W. S.:
Real-Time Systems.
Prentice-Hall, Inc., 2000. –
ISBN 0-13-099651-3
- [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



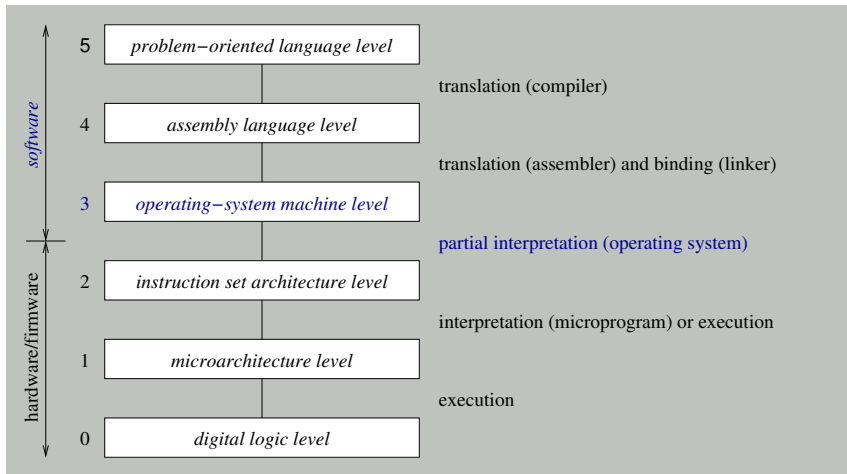
- [11] TANENBAUM, A. S.:
Multilevel Machines.
In: *Structured Computer Organization.*
Prentice-Hall, Inc., 1979. –
ISBN 0-130-95990-1, Kapitel Introduction, S. 1-17
- [12] WIKIPEDIA:
Process.
http://en.wikipedia.org/wiki/Legal_process, Apr. 2014



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 rivalry¹⁰ 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 unepit !!!
 - operating systems often command a multi-level processing of processes

¹⁰Lat. *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

