Course
TÖL401G: Stýrikerfi / Operating Systems
6. Process Synchronisation

Chapter Objectives

- Illustrate a race condition and describe the critical-section problem.
- Present software, hardware and higher-level solutions to the critical-section problem:
  - Peterson’s Algorithm,
  - Atomic instructions, including spinlocks,
  - Semaphores,
  - Monitors and condition variables,
  - Message passing.
  - Java API for semaphores and monitors with conditions.
- Present classical synchronisation problems.
- Have fun with the Deadlock Empire computer game.

Contents

Note for users of the Silberschatz et al. book: For didactical reasons, I will present the material of this chapter slightly differently.

6.1 Introduction: Shared Resources(samnýttar auðlind

- Processes/threads running in parallel, may share resources:
  - Devices: Printer, hard disk, etc.
  - Data: Files, memory shared between processes, global variables shared by threads of the same process, etc.

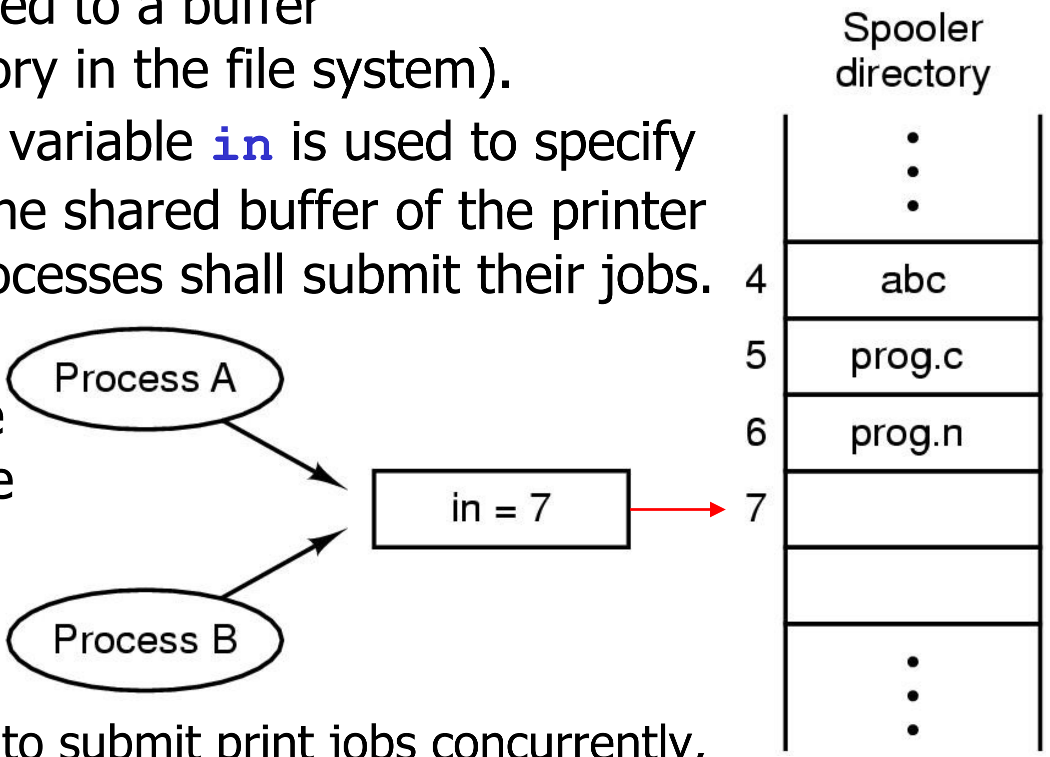
- If possible, shared resources shall be avoided (because they may lead to inconsistencies due to race conditions →next slide), but this is not always possible, e.g.
  - resource exists only once (because that hardware resource is expensive),
  - processes/threads need exactly to share data to work together and to communicate.

Race Conditions

- Access to shared resource must be coordinated (synchronised).
  - Otherwise, produced results depend on the schedule (which may differ from run to run) of the concurrent processes instead of being deterministic (i.e. independent from the possible schedules): "race condition" (or "race").
    - Race condition = Result depends on timing
      i.e. order in which scheduler decides to execute concurrent processes.
      - Because a user does not see the underlying schedules, the result seems to be non-deterministic (óákveðinn): sometimes a software works, sometimes not.
    - Consequences of a race condition:
      - Data may get lost and even corrupted. If operating system data structures are subject of race conditions, the whole system might crash.
    - To avoid race conditions, the possible schedules of concurrent processes need to be restricted to those that do not lead to problems.
      - Essentially, this whole chapter 6 is about how to achieve this!

Example for Race Conditions (1)

- Two processes A and B submit concurrently their print job to a printer spooler:
  - Print jobs are submitted to a buffer (e.g. RAM or a directory in the file system).
  - The global (=shared) variable `in` is used to specify the next free slot in the shared buffer of the printer spooler where the processes shall submit their jobs.
  - Each submitting process is responsible for updating the value of variable `in` after submitting a job: `in=in+1`.
    - If the processes try to submit print jobs concurrently, a race condition may occur (→next slide).



Example for Race Conditions (2)

(Single processor/single core – concurrency by scheduler context switches)

- **Process A**
  - Reads value of variable **`in`** (= 7) into CPU register.
  - Writes job into the slot that is specified by CPU register (= 7).
  - Gets interrupted by scheduler timer interrupt before updating variable **`in`**.
    - Dispatcher saves CPU register in PCB of process A. Process B selected by scheduler as next process.
- **Process B**
  - Reads value of variable **`in`** (which is still 7) into CPU register.
  - (Over-)Writes job into the slot that is specified by CPU register (= 7).
  - Increments CPU register, updates variable **`in`** using value in CPU register (new value: 8).
  - Terminates.
- **Process A**
  - CPU registers are restored by dispatcher, i.e. process A resumes execution where it was interrupted.
  - Increments CPU register, updates variable **`in`** using value in CPU register (new value: 8).
  - Terminates.
- Job from process A gets lost, because access to variable **`in`** was not synchronised.

Example for Non-deterministic Results Due to Race Conditions

Pseudo code construct to specify execution of P1 and P2 as concurrent processes/threads.

```
int a = 1, b = 1;
parallel {
    P1() {
      a = a + 1;
      b = b + 1;
    }

    P2() {
      b = 2*b;
      a = 2*a;
    }
}
```

a and **b** are shared variables.

Schedule 1

Time

```
a = a + 1;
b = b + 1;
b = 2 * b;
a = 2 * a;
```

Result:

```
a=2*(1+1)=4
b=2*(1+1)=4
```

Schedule 2

Time

```
a = a + 1;
b = 2 * b;
b = b + 1;
a = 2 * a;
```

Result:

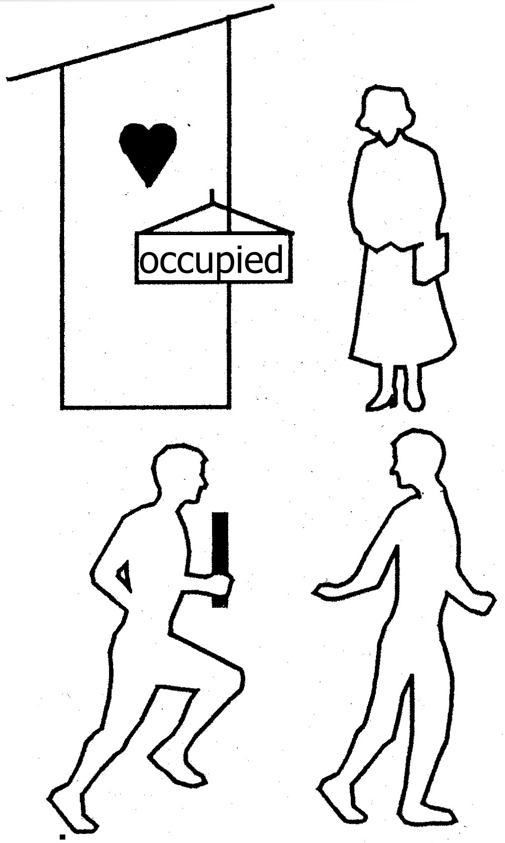
```
a=2*(1+1)=4
b=(2*1)+1=3
```

Note: These are just two possible schedules that the CPU scheduler of the operating system may create due to context switches while executing P1 and P2. In fact, even more different schedules exist. On multi-core/-processor systems, P1 and P2 may even run in parallel.

- Non-deterministic results due to race conditions!

Types of Synchronisation Between Interacting Processes

- Mutual Exclusion:
  - Prevent simultaneous access to shared resources.
    - (Order of access does not matter as long as it is mutual exclusive.)
  - Goal: Maintain consistency of data.
- Condition Synchronisation:
  - Wait for conditions/events.
  - Goal: Ordering of actions.

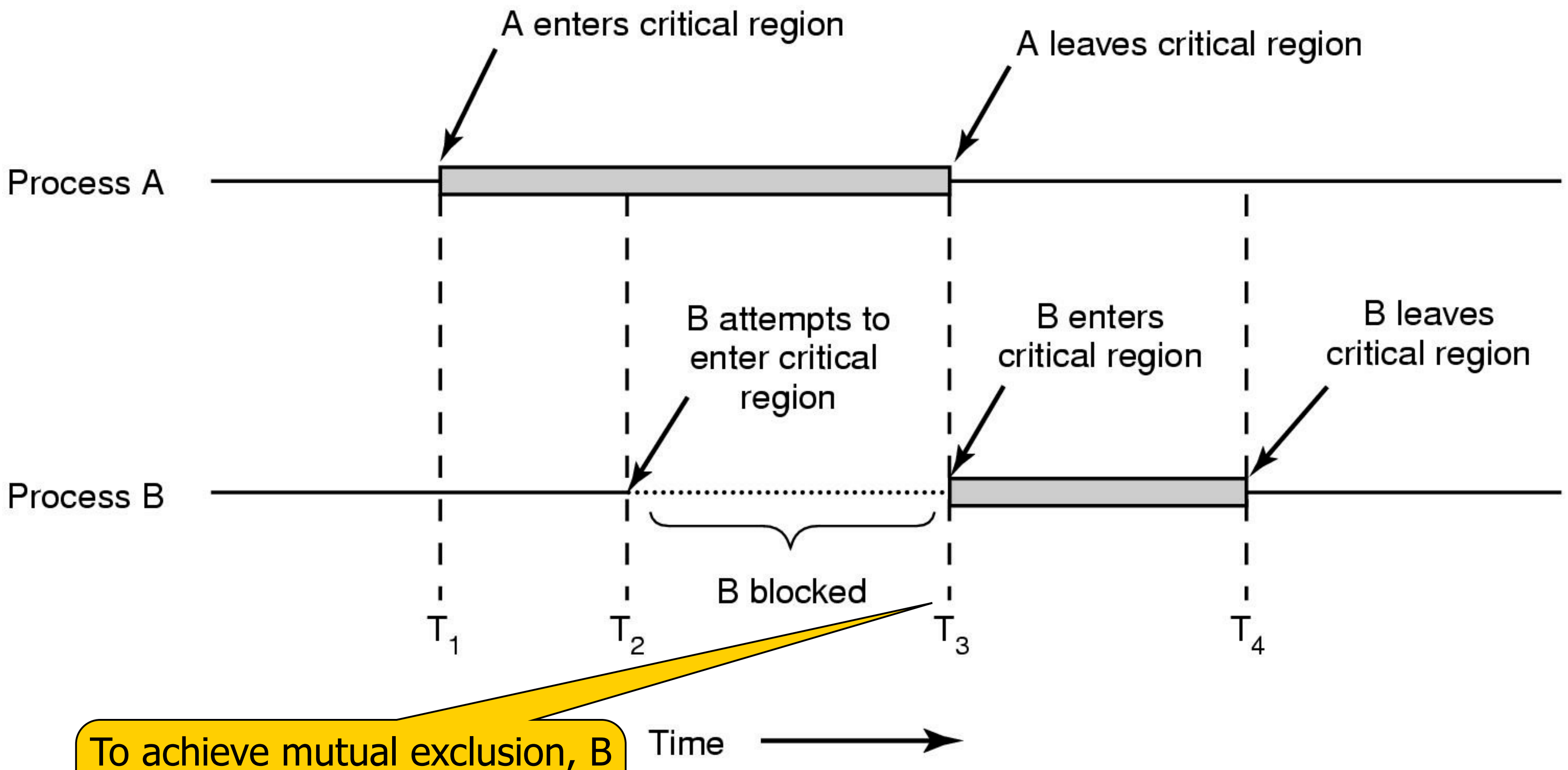


- Note: often, but not necessarily always, condition synchronisation includes mutual exclusion.
  - See, e.g. slide “Semaphores (10): Example Producer-Consumer Problem”:
    - There, conditions synchronisation does not include mutual inclusion – mutual exclusion rather needs to be added separately.

6.2 The Critical Section Problem

- A critical section (also called: critical region) of a process contains code that accesses shared resources.
  - Different sets of resources may be shared between processes. For each set of shared resources, a different critical section may exist.
    - E.g. critical section accessing resource A, another critical section accessing resource B: accessing A and B simultaneously is no problem.
- Mutual Exclusion (Gagnkvæm útilokun):
  - If a process is executing in its critical section, no other process can execute its critical section (concerning the same shared resources).
    - I.e. no process may enter its critical section, if another process is already in its critical section.
- By achieving mutual exclusion, the critical section problem (no accesses to the same shared resource at the same time) is solved.
  - A solution for orderly entering and leaving the critical section is required!

Entering & Leaving the Critical Section



To achieve mutual exclusion, B needs to be blocked until A leaves critical section

Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (mutual exclusion).
- No process that is outside its critical section may block other processes (progress).
- No process waits infinitely to enter its critical section (bounded waiting).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs/cores are made.

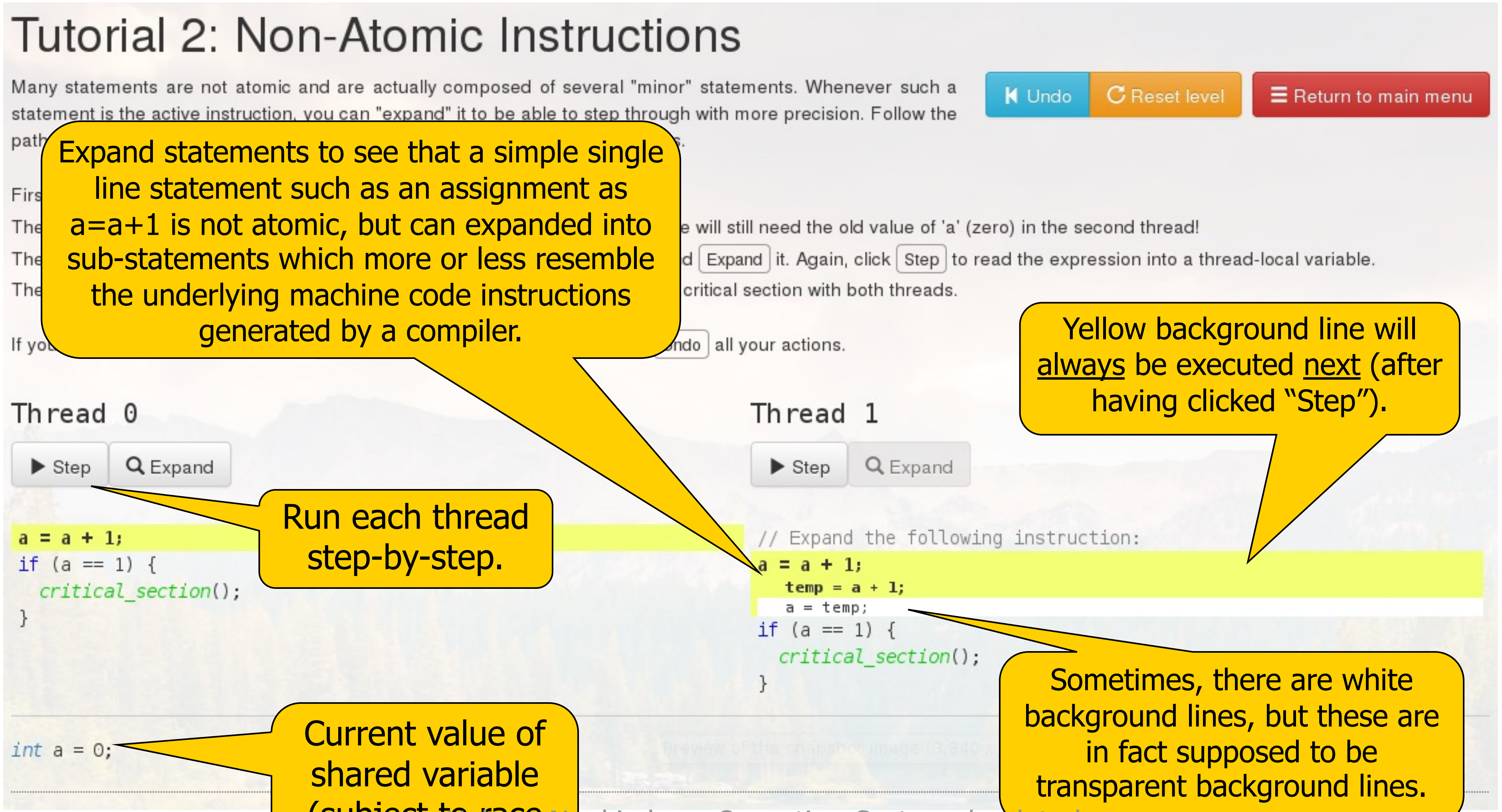
Atomicity/Atomic Instructions

- An operation or instruction (or a region of several instructions) is atomic, if it appears to the rest of the system (=other processes) to occur instantaneously, i.e. it cannot be divided or interrupted. E.g.:
  - The high-level programming language operation **`a=a+1`** is not atomic:
    - In fact, the compiler generates multiple machine code instructions:
      - Reading variable **`a`** from RAM into a CPU register,
      - Adding 1 to that that CPU register,
      - Finally writing that CPU register back to variable **`a`** in RAM.
    - Each of these machine code instructions can be interrupted (e.g. by a timer interrupt that activates the scheduler which might switch to a different process) or on a multi-core/multi-processor system another core/processor may execute other instructions of other processes in parallel.
  - The parallel regions of P1 and P2 on slide 6-8 are not atomic:
    - A scheduler may interrupt P1 or P2 in-between and switch to the other process or on a multi-core/multi-processor system another core/processor may execute the other processes in parallel.
- Atomicity is one possibility to achieve mutual exclusion.

Want to play a computer game? The Deadlock Empire!

- Browser game: `http://deadlockempire.github.io/`
  - While a real programmer has of course to avoid race conditions, you win the Dead-lock Empire games if you are able to provoke a race condition (or deadlock →ch. 7)
    - =Learning what can go wrong, teaches how avoid that things go wrong.
- Allows you to run step-by-step through various code examples using a simulator.
- You play being the scheduler, i.e. decide on which thread shall execute next its next instruction.
  - By this you can create "good" and "bad" schedules, e.g. circumventing race conditions or provoking deadlocks.
- Slay dragons, master concurrency!

The Deadlock Empire
User Interface of Simulator



Tutorial 2: Non-Atomic Instructions

Many statements are not atomic and are actually composed of several "minor" statements. Whenever such a statement is the active instruction, you can "expand" it to be able to step through with more precision. Follow the path

Firs

The e will still need the old value of 'a' (zero) in the second thread!

The d Expand it. Again, click Step to read the expression into a thread-local variable.

The critical section with both threads.

If yo ndo all your actions.

Thread 0

```
a = a + 1;
if (a == 1) {
  critical_section();
}
```

Thread 1

```
// Expand the following instruction:
a = a + 1;
  temp = a + 1;
  a = temp;
if (a == 1) {
  critical_section();
}
```

```
int a = 0;
```

Expand statements to see that a simple single line statement such as an assignment as a=a+1 is not atomic, but can expanded into sub-statements which more or less resemble the underlying machine code instructions generated by a compiler.

Yellow background line will <u>always</u> be executed <u>next</u> (after having clicked "Step").

Run each thread step-by-step.

Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.

Current value of shared variable (subject to race condition).

The Deadlock Empire
Take a break: Play now!

- Now:
  - Open in your web browser: **http://deadlockempire.github.io/**
  - Play from the above web page the "Tutorial" consisting of two games:
    - Tutorial 1: Interface
    - Tutorial 2: Non-Atomic Instructions
      - Play this game not only once, but try all the possible schedules.
    - The further games refer to topics that you'll learn later, so do not play them now, but later.
    - There are also videos in Panopto explaining each of the covered games.

If you are stuck, you can always watch
Panopto video with solutions for the Deadlock Empire: Starting from video
"Ch. 6 Deadlock Empire Tutorial 1: Interface"
ending at video
"Ch. 6 Deadlock Empire Condition Variables"

Types of Solutions for the Critical Section Problem

- Software solutions:
  - Solution on application/user level: No additional support by hardware or the operating system.
    - Busy waiting.
- Hardware-supported solutions:
  - Solution based on CPU support for special machine code instructions.
    - Atomic execution of instructions, Disable interrupts.
- Operating system-supported solutions:
  - Operating system provides special system calls.
    - Semaphore, Monitor, Message Passing, etc.
      - (OS may block process by putting into waiting state on scheduler level.)
- (We will discuss these three types in the following sections…)

6.3 Software Solutions

- Software solutions use shared global variables for synchronisation of processes.
- Software solutions use Busy Waiting:
  - Wait for a certain condition/event (e.g. global variable changes value) by continuously probing. E.g.

```
while (condition != true) { /* do nothing */ };
```

  - Busy Waiting consumes CPU time for doing nothing: inefficient!
    - Use software solutions only if no hardware or OS-supported solution are available.
- In the following, we consider only a solution for two processes/threads.

Software Solutions: Structure Used in Examples

- In the following examples we use C/Java-like pseudocode to investigate potential software solutions.
  All programs have the same structure:

```
    while (true) {
          Prologue critical section (Entry section)
          critical section
          Epilogue critical section (Exit section)
          Remainder section
    };
```

  - Prologue and epilogue of critical section are responsible for orderly entering and leaving critical section.
    - On the following slides, our main focus will be these prologues and epilogues of the critical sections. (It does not matter how the critical section looks like.)
    - For didactic reasons, we will first start with some (unsuccessful) tries that nearly solve the problem until we come up with the final solution.

Software Solutions: First Try (1)

- Global variable for synchronisation:

```
int turn = 1;
```

Specifies which process is allowed to enter the critical section.

- Two processes running in parallel:

Process P0:

```
while (true) {
    while (turn!=0){ /* wait */ };
    <critical section>
    turn = 1;
    /* Remainder */
  };
```

Process P1:

```
while (true) {
    while (turn!=1){ /* wait */ };
    <critical section>
    turn = 0;
    /* Remainder */
  };
```

Software Solutions: First Try (2)

- Discussion:
  - Advantage:
    - Mutual exclusion is achieved.
  - Disadvantage:
    - Processes may only enter critical section in an alternating manner.
    - Slower process dictates speed.
    - If one process terminates, the other process is blocked infinitely.
    - This violates the requirements on the solution (→reminder on next slide).

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (mutual exclusion).
- No process waits infinitely to enter its critical section (bounded waiting).
- No process that is outside its critical section may block other processes (progress).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.

Software Solutions: Second Try (1)

- Problem of first try:
  - Only that process whose ID is stored in the synchronisation variable, may enter the critical section next.
- Better:
  - Each process has its own “key” for the critical section to assure that
    - a process may still enter the critical section, even if the other process fails.
    - a process may enter the critical section independent from the other process that currently does not want to access the shared resource.

Software Solutions: Second Try (2)

- Global variables for synchronisation:

```
boolean    flag[2] = {false, false};
```

`flag[n]==true`, if process n is in critical section.

Process P0:

```
while (true) {
    while (flag[1]){ /* wait */ };
    flag[0] = true;
    <critical section>
    flag[0] = false;
    /* Remainder */
};
```

Process P1:

```
while (true) {
    while (flag[0]){ /* wait */ };
    flag[1] = true;
    <critical section>
    flag[1] = false;
    /* Remainder */
};
```

Software Solutions: Second Try (3)

- Discussion
  - Advantage:
    - If one of the processes fails (or terminates) outside its critical section, the other process may still enter its critical region.
    - Processes may enter their critical region in a non-alternating style independent from the speed of the other process.
  - Disadvantage:
    - Not even the requirement of mutual exclusion is achieved. Counter example:

Schedule

Assumption: `flag[2] = {`**`false`**`, `**`false`**`}`
P0: executes **`while`** `(flag[1]);` and notices that `flag[1]` is **`false`**.
P1: executes **`while`** `(flag[0]);` and notices that `flag[0]` is **`false`**.
P0: executes `flag[0] = `**`true`**`;`
P1: executes `flag[1] = `**`true`**`;`
****** Both processes enter their critical section. ******

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (mutual exclusion).
- No process waits infinitely to enter its critical section (bounded waiting).
- No process that is outside its critical section may block other processes (progress).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.

Software Solutions: Third Try (1)

- Problem of second try:
  - Process may change its flag after another process has probed that flag.
- Maybe, this problem can be solved by swapping the order of the two problematic instructions?
  - I.e. first change flag, then probe flag.

Software Solutions: Third Try (2)

- Global variables for synchronisation:

```
boolean    flag[2] = {false, false};
```

Process P0:

```
while (true) {
   flag[0] = true;
   while (flag[1]){ /* wait */ };
   <critical section>
   flag[0] = false;
   /* Remainder */
 };
```

Process P1:

```
while (true) {
   flag[1] = true;
   while (flag[0]){ /* wait */ };
   <critical section>
   flag[1] = false;
   /* Remainder */
 };
```

Software Solutions: Third Try (3)

- Is this third try the solution of the critical section problem?
  - Viewpoint of P0 – Possible cases:
    - P0 sets `flag[0]` to **`true`**, P1 is not in the critical section:
      - P1 may only enter critical section after P0 entered it (because P1 is blocked by **`while`** `(flag[0]);`) and left it again.
    - P0 sets `flag[0]` to **`true`**, P1 is in the critical section:
      - P0 is blocked by **`while`** `(flag[1]);` until P1 has left the critical section.
  - The same considerations are valid for P1, hence this third try achieves mutual exclusion.
  - **But:**

Software Solutions: Third Try (4)

- ... a deadlock (=each process is waiting for the other process →ch. 7) may occur:

Assumption: `flag[2] = {false, false}`
P0: executes `flag[0] = true;`
P1: executes `flag[1] = true;`
P0: executes `while (flag[1]);` and **waits** because `flag[1] == true`.
P1: executes `while (flag[0]);` and **waits** because `flag[0] == true`.
****** A deadlock occurs: Each process waits infinitely that the other process leaves the critical section. ******

  - Of course, this violates one of our requirements...

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (mutual exclusion).
- No process waits infinitely to enter its critical section (bounded waiting).
- No process that is outside its critical section may block other processes (progress).
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.

Software Solutions: Excursion into History

- For long time (in the early days of computer science), researchers thought that there is no software solution for the critical section problem.
- In 1965, the Dutch mathematician Theodorus J. Dekker published a correct software solution for the critical section problem!
  - However, Dekker's algorithm is somewhat hard to understand and the correctness is hard to prove.
- In 1981, the American mathematician Gary L. Peterson published a more simple and more elegant solution of the critical section problem.
  - This algorithms uses both a **`flag`** array (to indicate the state of each process concerning the mutual exclusion) and a variable **`turn`** (to resolve problems of identical relative speed). (→next slide)

Software Solutions: Peterson's Algorithm (1)

- Global variables for synchronisation:

```
boolean     flag[2] = {false, false};
int         turn;
```

In principle, the third try was good. Now, the **turn** variable is used to introduce some asymmetry so that not both processes wait for each other if they progress in parallel.

Process P0:

```
 while (true) {
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
                  { /* wait */ };
   <critical section>
   flag[0] = false;
   /* Remainder */
 };
```

Process P1:

```
 while (true) {
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
                  { /* wait */ };
   <critical section>
   flag[1] = false;
   /* Remainder */
 };
```

Software Solutions: Peterson's Algorithm (2) – Proof

=Proofing that the four requirements on solutions for the critical section problem are not violated.

- 1. No process waits infinitely to enter its critical section (bounded waiting):
  - Assumption (indirect proof, i.e. proof by contradiction):
    P0 is blocked infinitely in its inner **`while`** loop (=waits infinitely).
  - Distinguish three cases concerning what P1 might be doing:
    - Case 1:
      - P1 does not want to enter the critical section (=is outside).
    - Case 2:
      - P1 is waiting as well in its **`while`** loop to enter the critical section.
    - Case 3:
      - P1 enters the critical section over and over again, therefore infinitely blocking P0.
  - In the following: Proof by contradiction.

Software Solutions: Peterson's Algorithm (3) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
  - Assumption (reminder):
    - P0 is blocked infinitely in its inner **`while`** loop.
    - This means:
      - `flag[1] ==` **`true`** and `turn ==` **`1`**
  - Case 1:
    - P1 does not want to enter the critical section (=is outside).
      - This means, `flag[1] ==` **`false`**, hence the condition for the blocking of P0 in its while loop is not fulfilled and thus, blocking of P0 is not possible.
        $\Rightarrow$Contradiction to assumption.

Software Solutions: Peterson's Algorithm (4) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
  - Assumption (reminder):
    - P0 is blocked infinitely in its inner **`while`** loop.
    - This means:
      - `flag[1] ==` **`true`** and `turn ==` **`1`**
  - Case 2:
    - P1 is waiting as well in its **`while`** loop to enter the critical section.
      - Not possible, as P1 would be able to enter the critical section because `turn ==` **`1`**.
        (`turn` is either **`0`** or **`1`**, hence it is impossible that both processes are blocked at the same time.)
        ⇒Contradiction to assumption.

Software Solutions: Peterson's Algorithm (5) – Proof

- 1. No process waits infinitely to enter its critical section (bounded waiting):
  - Assumption (reminder):
    - P0 is blocked infinitely in its inner **`while`** loop.
    - This means:
      - `flag[1] == true` and `turn == 1`
  - Case 3:
    - P1 enters the critical section over and over again, therefore infinitely blocking P0.
      - Not possible as P1 sets **`turn`** to 0 each time it tries to enter the critical section, thus giving P0 the possibility to enter the critical section. $\Rightarrow$Contradiction to assumption.

Software Solutions: Peterson's Algorithm (6) – Proof

- 2. No two processes may be in their critical section at the same time (mutual exclusion):
  - Assumption:
    Both, P0 and P1 want to enter the critical section.
    - P0 sets `flag[0]` to **`true`**, P1 sets `flag[1]` to **`true`**.
    - `turn` is set by P0 as well as by P1. The one that sets it later, "wins". `turn` can only have one value at a time, depending on the scheduling, two cases are possible:
      - `turn==0`:
        - P1 cannot enter critical section.
      - `turn==1`:
        - P0 cannot enter critical section.

$\Rightarrow$ Only one process in critical section.

Software Solutions: Peterson’s Algorithm (7) – Proof

- 3. No process that is outside its critical section may block other processes (progress):
  - As soon as P1 leaves its critical section, it sets `flag[1]` to **`false`**, hence P0 may enter the critical section.
  - The same considerations are valid in the opposite direction:
    As soon as P0 leaves its critical section, it sets `flag[0]` to **`false`**, hence P1 may enter the critical section.

Reminder: Requirements on Solutions for the Critical Section Problem

- No two processes may be in their critical section at the same time (mutual exclusion).
  - See proof 2. ✓
- No process waits infinitely to enter its critical section (bounded waiting).
  - See proof 1. ✓

Well… if a process crashes inside it's critical section, then the other process waits infinitely. (But this is a general problem that cannot be prevented – except using timeouts.)

- No process that is outside its critical section may block other processes (progress).
  - See proof 3. ✓
- Furthermore: No assumptions concerning the relative speed of processes and the number of CPUs are made.
  - No such assumptions were necessary. ✓

Well… there may always be unlucky schedules: priority inversion problem →6-52

Remark: Running Software Solutions on Modern Systems: **`volatile`** keyword

- Modern compilers apply various optimisations to speed up program execution:
  - E.g. in the sequence "`turn = 1;` **`while`** `(flag[1] && turn == 1)`" a compiler could conclude that the check `turn == 1` is unnecessary (because turn has just been set to 1 in the statement before) and thus generate no code for it.
    $\Rightarrow$ A software solution (that is based on that variable **`turn`** is shared and may be set in parallel by another process) would not work anymore!
  - To avoid this, most programming languages allow to declare a variable as "**`volatile`**" to prevent such optimisations.
    - **`volatile`** will tell the compiler to always read the value from main memory.
    - Useful for shared variables modified by other processes/threads.
    - Useful if memory location may change for other reasons, e.g. an I/O controller directly writing data to that location.
  - However, even using the "**`volatile`**" keyword does not guarantee that software solutions work in modern systems ⇒next slide.

Remark: Running Software Solutions on Modern Systems: Reordering by CPUs

- Modern CPUs reorder internally instructions and their memory accesses to improve execution speed, e.g. to avoid memory stalls:
  - When running software solutions, the actual accesses to the synchronisation variables may occur in a different order than expected (only a problem with shared memory).

  $\Rightarrow$ The software solution will not work anymore on any modern CPU!
- The memory model of a CPU describes what ordering is guaranteed and what ordering you cannot rely on, e.g.:
  - Strongly ordered: a memory modification on one processor/core is immediately visible to all other processors/cores.
  - Weakly ordered: modifications to memory on one processor /core may not be immediately visible to other processors/cores.

Remark: Running Software Solutions on Modern Systems: Memory barriers

- To enforce an ordering, CPUs have special machine code instructions that do not get reordered: "memory barriers"/"memory fences":
  - All memory accesses of machine code instructions will be completed before the barrier machine code instruction continues execution and after continuing, all cores will see the result of these accesses performed before the barrier.
    - Also special atomic instruction (→Hardware solutions, next slides) have that property.
  - Compilers do not automatically generate these instructions, because these slow down execution and a compiler cannot detect those rare case when they are actually needed.
- Note: since version (1.)5 of Java, accesses to **`volatile`** variables use memory barrier instructions to prevent memory access reordering!
  - I.e. the memory model of the Java Virtual Machine changed!
- Other programming languages may not use barriers for **`volatile`**…

Want to play more computer games? The Deadlock Empire strikes back!

- Browser game: **http://deadlockempire.github.io/**
  - Reminder: in these games always the yellow background line will be executed next.
    - Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.
- Now, play the three "Unsynchronized Code" games:
  - Boolean Flags Are Enough For Everyone
  - Simple Counter
  - Confused Counter
    - "the failure instruction" refers to `Debug.Assert(false);`
    - C# is used: the `Interlocked.Increment` mentioned in one of the pop-ups refers to an atomic C# operation (comparable atomic Java operations exist as well, e.g. in class `java.util.concurrent.atomic.AtomicInteger`).
  - The further games refer to topics that you'll learn later (or not at all).

If you are stuck, watch Panopto videos with solutions

6.4 Hardware Solutions

- Solving the critical section problem may use special hardware features of a CPU:
  - Either: Disabling interrupts
  - Or: Atomic instructions.

  - Atomic machine code instructions may be used even in user mode (however, typically a high-level programming language does not support them);
  - However, disabling interrupts is typically a privileged machine code instruction that requires kernel mode and thus this is typically only used by the kernel itself.
    - Furthermore, not reasonable on today's multicore-processors.

Disabling Interrupts

- Simple approach: before entering the critical section, all hardware interrupts are disabled (and enabled again after leaving the critical section):

```
while (true) {
  Disable interrupts
  <critical section>
  Enable interrupts
  Remainder
};
```

  - Prevents that a timer interrupt is processed by the OS scheduler.
  - $\Rightarrow$ No context switch (preemption) occurs and thus no other processes get executed that might enter the critical section. (well – see problems discussed on next slide…)

Disabling Interrupts: Problems

- Not reasonable in multi-core/multi-processor systems:
  - On multi-core/multi-processor systems, multiple processes can execute at the same time (one process on each core/processor) even without a scheduler and context switch, thus each of them might enter a critical section even if all interrupts are disabled.
- Process must not reside too long in critical section, otherwise
  - it may be too late to serve interrupt,
    - (Imagine the interrupt of a mass storage device that indicates that it wants to be served: data may get lost if it is not served fast enough.)
  - interrupts may get lost.
    - (Even worse than just handling interrupt too late.)
- If a process crashes (or is in a deadlock) while being in the critical section, the whole system gets blocked (because, interrupts never get enabled again)!
  - (The timer interrupt of the scheduler is disabled and thus the scheduler cannot interrupt crashed processes!)
- Realises only mutual exclusion, however not condition synchronisation.
  - A process cannot wait that a certain condition gets fulfilled by another process.

How to achieve atomicity?

- On single core/single processor systems, typically interrupt disabling is used to achieve mutual exclusion when accessing kernel data structures.
  - If care is taken that interrupts are disabled for a short section of code, this is an appropriate solution to achieve mutual exclusion within kernel code.
- However, for SMP (symmetric multi core/multi processor) systems, a different solution is required:
  - Typically, this is based on atomic machine code instructions:
    - The difficult part of the software solutions of the critical section problem was that between testing and setting a lock variable (variable “flag” in the previous examples) another process could set the lock variable as well.
  - Most CPUs provide special machine code instructions that allow to test and set the value of a memory location without being interrupted in-between.
    - No internal memory access reordering and even with multiple cores/CPUs, only one will be able to access memory at a time (“atomic instruction”).
    - Note: Instead of an atomic test-and-set, a CPU might offer an atomic swap or atomic exchange instruction: Sets value of memory location and returns previous value (that can later-on be tested).

Atomic Instructions: Test-And-Set Instruction

- If a Test-And-Set machine code instruction of a CPU would be implemented using Java syntax (in fact, a CISC CPU uses microcode to implement it's machine code instructions), its implementation would look as follows:

```
boolean lock=false;
boolean testAndSet(boolean newValue) {
   boolean oldValue=lock;
   lock=newValue;
   return oldValue;
}
```

Global variable used as "lock".

Atomic read/write access to memory location `lock` that cannot be interrupted and gets not reordered.

Return value can be used to check whether another process is already in critical section.

- Variant without `newValue` parameter:

```
boolean lock=false;
boolean testAndSet() {
   boolean oldValue=lock;
   lock=true;
   return oldValue;
}
```

Global variable used as "lock".

Atomic read/write access to memory location `lock` that cannot be interrupted and gets not reordered.

Always set to true: If `lock` was already true, it remains true, if it was false, it becomes true. Previous value will be returned and can be checked.

Atomic Instructions: Test-And-Set Instruction: Example

- Mutual exclusion of *n* processes using test-and-set instruction:

```
boolean lock=false;
void P() {
  while(true) {
    while( testAndSet() ){ /* busy waiting */ };

    /* critical section */

    lock = false;
    /* further program code */
  }
}
void main() {
  parallel { P() }{ P() } ... { P() };
}
```

Wait until `lock` becomes false and set it to true. Due to atomicity, only one process will read a **`false`**.

Setting `lock` (without testing it) requires no atomic instruction.

- Easy to use.
- Applicable for mutual exclusion of *n* processes (not just 2).

Atomic Instructions: Discussion

- Advantages:
  - Easy to use.
  - Applicable for an arbitrary number of processes.
  - Test-and-Set (or Exchange/Swap) allows (in contrast to disabling interrupts) control of different critical sections for different shared resources at the same time: just use a different lock variable for each shared resource.
- Disadvantages:
  - Busy waiting is used (wastes CPU time).
  - Starvation is possible:
    - A low priority process will never be able to enter critical section as long as there are higher priority processes competing for critical section using busy waiting.
  - Deadlocks are possible ("priority inversion problem" $\rightarrow$next slide)

6.5 Operating System-supported Solutions

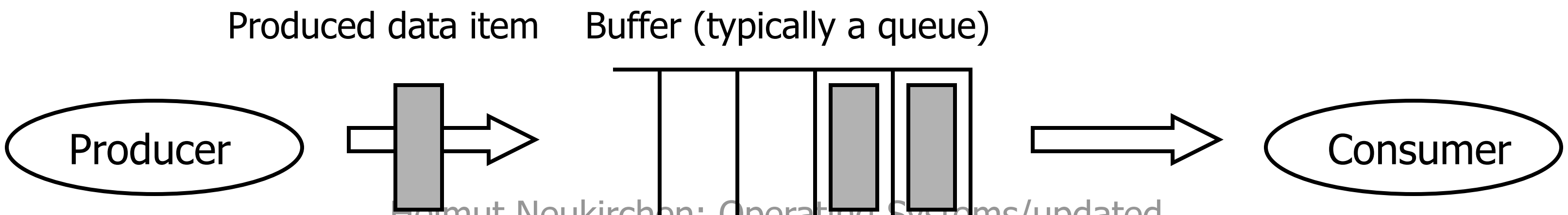
- Main problem of software and hardware solutions for the critical section problem is busy waiting.
- Wanted: synchronisation mechanisms that avoid busy waiting.
  - Can only be achieved on operating system-level, because only OS can block processes in a non-busy waiting style by putting processes into the scheduler's blocked queue/waiting state.

  ⇒ Operating system-supported process synchronisation mechanisms.
    - Influence directly the scheduling by blocking one or more processes: scheduler assigns CPU only to processes that are in ready state.

    ⇒ Set of possible schedules gets restricted.

Example: Producer-Consumer Problem

- In the following, we will use the producer-consumer problem as case study for applying operating system-supported process synchronisation mechanisms.
- Reminder producer-consumer problem: Concurrent producer and consumer processes share a common buffer.
  - Producer:
    - Puts (inserts) data items into buffer until it is full.
    - Waits if buffer is full.
    - Resumes when buffer is not full any more.
  - Consumer:
    - Consumes (removes) data items from buffer.
    - Waits if buffer is empty (no items available).
    - Resumes when new items are available.



Example: Producer-Consumer Problem First try: Sleep and Wakeup (1)

- Assume that we have system calls `sleep` and `wakeup`:
  - `sleep()` system call:
    - The calling process volunteers to be put to sleep by the operating system until it is woken up by another process.
  - `wakeup(processID)` system call:
    - The calling process wakes up the sleeping process with PId `processID`.
      - If process with PId `processID` is not sleeping, wakeup signal is discarded.
- How would a solution of the producer-consumer problem look like when using `sleep` and `wakeup`?

Example: Producer-Consumer Problem First try: Sleep and Wakeup (2)

- Producer

Producer and consumer share buffer and this variable.

```
#define N 100              /* Size of buffer */
int count = 0;             /* Number of items currently in buffer */

void producer() {
  int item;

  while (TRUE) {                      /* Endless loop */
    item = produce_item();            /* produce new item */
    if (count == N){sleep()};/* sleep if buffer exceeded */
    insert_item(item);                /* put item into buffer */
    count = count + 1;                /* incr. number of items in buffer */
    if (count == 1){wakeup(consumer)}; /* if buffer was empty
before, consumer was sleeping to wait for new items */
  }
}
```

Example: Producer-Consumer Problem First try: Sleep and Wakeup (3)

- Consumer

```
#define N 100              /* Size of buffer */
int count = 0;             /* Number of items currently in buffer */

void consumer(){
  int item;
  while (TRUE) {                /* Endless loop */
    if (count == 0){sleep()};/* sleep if buffer empty */
    item = remove_item();       /* fetch new item from buffer */
    count = count - 1;          /* decr. number of items in buffer */
    if (count == N - 1){wakeup(producer)}; /* if buffer was full,
now one slot is available again: wakeup producer */
    consume_item(item);         /* consume fetched item*/
  }
}
```

Example: Producer-Consumer Problem First try: Sleep and Wakeup (4)

- Solution looks good, however deadlock is possible. Consider the following scenario where both processes have just been started and the buffer is empty (i.e. `count`=0):

Consumer:	checks `count` in if statement with condition `count == 0`
Scheduler:	interrupts consumer (just before `sleep()`), and selects producer

Producer:	produces item and inserts it into empty buffer:
	`insert_item(item);`
	`count = count + 1; /* count is now 1 */`
	`if (count == 1) wakeup(consumer);`
	wakeup signal is sent to consumer. However, consumer process does not sleep yet; hence, wakeup signal is discarded!

Scheduler:	interrupts producer and selects consumer
Consumer:	resumes interrupted execution: - Executes second part of if statement (sleep()): puts itself to sleep
Scheduler:	consumer is blocked, hence scheduler selects producer
Producer:	resumes execution: - produces new items and thus fills buffer until it is exceeded. - Hence, puts it self to sleep to wait until consumer consumes item.
***** Deadlock, both processes are sleeping! *****	

Example: Producer-Consumer Problem First try: Sleep and Wakeup (5)

- The problem of `sleep` and `wakeup` in the previous example is that the wakeup signal gets discarded if the target process is not sleeping yet.
  - Problem would be solved, if wakeup signal gets not discarded but would be buffered in some queue for later consumption.
- Furthermore, an atomic "check and sleep" would have prevented the problem as well.
  - Besides this, another problem that might occur is that **`count=count+1`** in producer and **`count=count-1`** in consumer might be updated in parallel → race condition!
    ⇒ An atomic update of the counter is needed & the above queue.
  - This is how semaphores work!
    - So forget all about **`sleep`** and **`wakeup`**, instead find more on semaphores on the next slides.

Semaphores (1)

- Suggested 1965 by the Dutch computer scientist Edsger Wybe Dijkstra (1930-2002).
  - (Historically, the term "semaphore" refers to indicators whether a railway section is occupied or free – comparable to a traffic light.)

- A semaphore can be considered as a special type of variable (or as a special class in case of object-orientation). It consists of:
  - Integer value ("value" of the semaphore, used as counter),
  - Operations that can be applied to it:
    - Initialise (either pass a parameter or use 1 as default value): init
    - P ("Proberen": Dutch for "try"): down/acquire/wait
    - V ("Verhogen": Dutch for "increase"): up/release/signal
    - wait and signal are implemented as atomic operations.
    - In pseudocode, typically an object-oriented notation is used:
      - `mysemaphore.init(1), mysemaphore.wait()` etc.

In literature, all these different names can be found for these operations. From now on, only the names wait and signal will be used

Semaphores (2): Implementation of init()

- Internal data structure used for each instance of a semaphore:

```
struct semaphore {
   int        count;
   queueType  queue;
}
```

Queue for keeping track of sleeping processes.

"Value" of semaphore: current value essentially indicates how much "space" (in terms of number of processes that are allowed to enter) is left in the critical section.
(There may be applications where it is reasonable to use a value higher than 1, in particular if the semaphore is not used to achieve mutual exclusion, but rather for condition synchronisation.)

- Init operation (Pseudocode implementation):

```
init(int init_value) {
   count = init_value;
   queue = empty;
}
```

Semaphores (3): Implementation of wait()

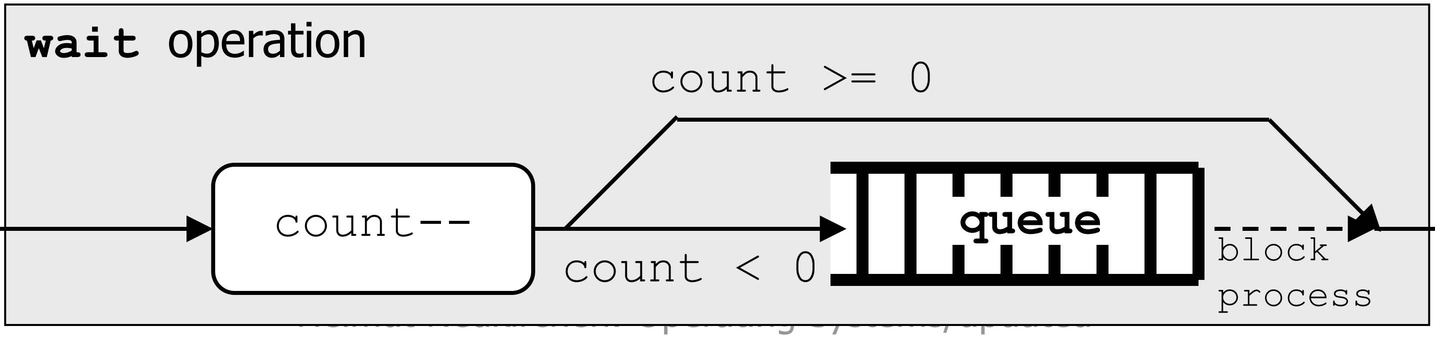
- Atomic wait operation (pseudocode):

```
wait() {
   count = count - 1;
   if (count < 0) {
     put this process in semaphore's queue;
     block this process on OS level;
    (=move process into waiting queue on OS level ⇒ process returns
     only from this wait() once it gets woken up by signal())
   }
}
```

Atomic
(e.g. using hardware or software
solution of critical section problem)

The same process that called the wait() operation will be put to sleep in this case!

- Schematic view:



Semaphores (4): Implementation of signal()

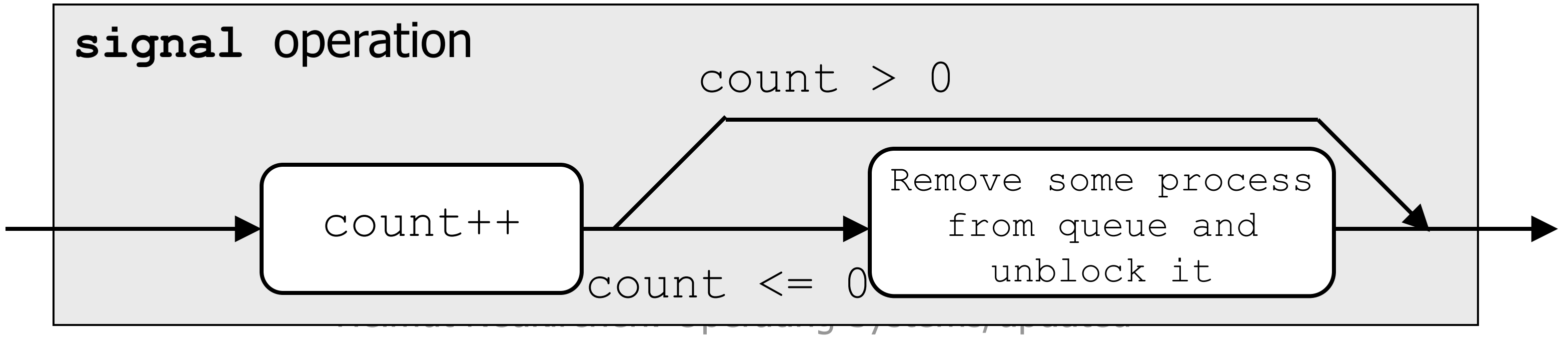
Another process (one that called the wait() operation and was put to sleep) will get woken up in this case!

- Atomic signal operation (pseudocode):

Atomic
(e.g. using hardware or software solution of critical section problem)

```
signal() {
  count = count + 1;
  if (count <= 0) {
    Remove some process P from semaphore's queue;
    Move process P to ready queue on OS level;
   (=wakeup process P)
  }
}
```

- Schematic view:



Semaphores (5): Typically Patterns of Using Semaphores

- Mutual exclusion:

```
sem_me.init(1)

Process A:
sem_me.wait()
<critical section>
sem_me.signal()

Process B:
sem_me.wait()
<critical section>
sem_me.signal()
```

- Condition synchronisation:

```
sem_condition.init(0)

Process A:
<condition occurred>
sem_condition.signal()

Process B:
sem_condition.wait()
<do something after
condition occurred>
```

  - For more complex problems, multiple condition synchronisations may have to be used in parallel or mutual exclusion and condition synchronisation have to be combined. (In either case, this requires use of multiple independent semaphores.) Semaphores may have to be initialised with other values than 0 or 1 to use them as counters. →Examples on next slides…
  - Take care that each `wait()` has somewhere a matching `signal()` for that semaphore, otherwise a waiting process gets never unblocked!
  - A blocked process cannot unblock itself (it sleeps: only another process can signal).

Semaphores (6): Example Mutual Exclusion

- Mutual exclusion of *n* processes using a semaphore:

```
semaphore S_ME = S_ME.init(1);   // S_ME=Semaphore for mutual excl.

void P() {
  while (true) {
     S_ME.wait();       /* Enter critical section or wait */
     /* critical section */
     S_ME.signal();     /* Leave critical section */

     /* Remainder */
  }
}

void main() {
  parallel { P() }{ P() } ... { P() };
}
```

Initial value of 1 achieves that at most 1 process may enter critical section. (If some problem would allow *n* processes in the critical section at the same time, initialise semaphore with *n*.)

Create n concurrent processes executing code of P()

Semaphores (7): Example Producer-Consumer Problem

- Solution for bounded producer-consumer problem (condition synchronisation) with a buffer size of 1:
  - Semaphores are not used to achieve mutual exclusion, but to put processes to sleep and wake them up according to conditions.

```
semaphore S_Data = S_Data.init(0); // >0: Producer produced item
semaphore S_NoData = S_NoData.init(1); // >0: Consumer requests item
```

```
void Producer() {
  while (true) {
    S_NoData.wait();
    /* Produce item */
    S_Data.signal();
  }
}
```

```
void Consumer() {
  while (true) {
    S_Data.wait();
    /* Consume item */
    S_NoData.signal();
  }
}
```

```
void main() {
  parallel { Producer() }{ Consumer() };
}
```

Semaphores (8): Example Producer-Consumer Problem

- Sample trace for the previous example:

Progress of time

```
<S_Data=0, S_NoData=1>
Producer: S_NoData.wait
<S_Data=0, S_NoData=0>
Producer: S_Data.signal
<S_Data=1, S_NoData=0>
Producer: S_NoData.wait
<S_Data=1,S_NoData=-1,
S_NoData.queue=Producer>
Consumer: S_Data.wait
<S_Data=0, S_NoData=-1,
S_NoData.queue=Producer>
```

```
Consumer: S_NoData.signal
<S_Data=0, S_NoData=0>
```

(Above `signal` unblocks Producer)

```
Producer: S_Data.signal
<S_Data=1, S_NoData=0>
```

Assume scheduler switches to Consumer (e.g. timeslice of Producer expired)

```
Consumer: S_Data.wait
<S_Data=0, S_NoData=0>
Consumer: S_NoData.signal
<S_Data=0, S_NoData=1>
```

Now, execution might repeat from the beginning of this trace (or many other traces possible as well).

Semaphores (9): Example Producer-Consumer Problem

- The following slide will provide a solution of the bounded producer-consumer problem with an arbitrary size of the buffer. For this, we need three different semaphores:
  - S_ME: Achieves mutual exclusion when accessing shared buffer that contains the produced items.
  - S_Free: Counts number of free slots in the buffer.
    - Used for condition synchronisation: wait if no free slots in the buffer.
  - S_Prod: Counts number of produced items in the buffer.
    - Used for condition synchronisation: wait if no items in the buffer.
  - (Values of S_Free and S_Prod will be used for condition synchronisation and will grow and shrink inverse to each other.
    - Two semaphores necessary because of two conditions: 1. buffer full, 2. buffer empty. For any buffer fill state in-between, producer and consumer can run concurrently and thus a third semaphore is needed to achieve mutual exclusive access to shared buffer.)

Semaphores (10): Example Producer-Consumer Problem

- Bounded producer-consumer problem with arbitrary buffer size:

```
semaphore S_Prod = S_Prod.init(0);          // no. of produced items
semaphore S_Free = S_Free.init(capacity);   // capacity: buffer size
semaphore S_ME = S_ME.init(1);              // mutual exclusion
```

```
void Producer() {                      void Consumer() {
  while (true) {                         while (true) {
    S_Free.wait();                         S_Prod.wait();
    S_ME.wait();                           S_ME.wait();
    /* Produce item */                     /* Consume item*/
    S_ME.signal();                         S_ME.signal();
    S_Prod.signal();                       S_Free.signal();
  }                                      }
}                                      }
```

```
void main() {
  parallel { Producer() }{ Consumer() };
}
```

- Sample trace for a buffer size of 2 will be shown on next slide…

Semaphores (11)

```
<S_Prod=0, S_Free=2, S_ME=1>
Producer: S_Free.wait
<S_Prod=0, S_Free=1, S_ME=1>
Producer: S_ME.wait
<S_Prod=0, S_Free=1, S_ME=0>
Producer: S_ME.signal
<S_Prod=0, S_Free=1, S_ME=1>
Producer: S_Prod.signal
<S_Prod=1, S_Free=1, S_ME=1>
Producer: S_Free.wait
<S_Prod=1, S_Free=0, S_ME=1>
Producer: S_ME.wait
<S_Prod=1, S_Free=0, S_ME=0>
Producer: S_ME.signal
<S_Prod=1, S_Free=0, S_ME=1>
Producer: S_Prod.signal
<S_Prod=2, S_Free=0, S_ME=1>
Producer: S_Free.wait
<S_Prod=2,S_Free=-1, S_ME=1,
S_Free.queue=Producer>
```

Producer gets blocked

```
Consumer: S_Prod.wait
<S_Prod=1, S_Free=-1, S_ME=1,
S_Free.queue=Producer>
Consumer: S_ME.wait
<S_Prod=1, S_Free=-1, S_ME=0,
S_Free.queue=Producer>
Consumer: S_ME.signal
<S_Prod=1, S_Free=-1, S_ME=1,
S_Free.queue=Producer>
Consumer: S_Free.signal
<S_Prod=1, S_Free=0, S_ME=0>
```

Producer gets unblocked, but assume: producer does not get CPU, yet (e.g. timeslice of consumer not yet expired).

```
Consumer: S_Prod.wait
<S_Prod=0, S_Free=0, S_ME=1>
Consumer: S_ME.wait
<S_Prod=0, S_Free=0, S_ME=0>
Consumer: S_ME.signal
<S_Prod=0, S_Free=0, S_ME=1>
Consumer: S_Free.signal
<S_Prod=0, S_Free=1, S_ME=1>
```

Assume: producer gets CPU now (e.g. timeslice of consumer expired now).

```
Producer: S_ME.wait
<S_Prod=0, S_Free=1, S_ME=0>
Producer: S_ME.signal
<S_Prod=0, S_Free=1, S_ME=1>
Producer: S_Prod.signal
<S_Prod=1, S_Free=1, S_ME=1>
```

Assume: consumer gets CPU.

```
Consumer: S_Prod.wait
<S_Prod=0, S_Free=1, S_ME=1>
Consumer: S_ME.wait
<S_Prod=0, S_Free=1, S_ME=0>
Consumer: S_ME.signal
<S_Prod=0, S_Free=1, S_ME=1>
Consumer: S_Free.signal
<S_Prod=0, S_Free=2, S_ME=1>
```

Now, execution might repeat from the beginning of this trace (or many other traces are possible as well).

Semaphores (12): Implementation of Atomicity

- wait and signal operations of a semaphore must be atomic.
  - Semaphores are implemented by the entity that manages the waiting and ready queues of the scheduler: typically, the OS kernel (or a user-level thread library).
  - However, even functions provided by the OS kernel are not atomic:
    - Kernel may be interrupted by hardware interrupts.
    - In multi-processor/multi-core systems, another processor/core may concurrently execute instructions and modify memory.

$\Rightarrow$In single processor/core systems, the OS kernel typically disables interrupts (no busy waiting) to implement atomicity of wait and signal system call.

$\Rightarrow$On multi-processor/multi-core systems, the OS kernel typically uses atomic test-and-set operations (together with busy waiting) to make wait and signal system calls mutual exclusive.

Semaphores (13): Further Types of Semaphores

- Mutex/binary semaphore:
  - A semaphore that has no generic counter variable, but that only controls mutual exclusion.
    - "Mutex" refers to "mutual exclusion".
    - "Binary semaphore" refers to the fact that the semaphore has just two states and does not really need an integer counter:
      - count=1, if critical section is free,
      - count=0, if critical section is occupied (no negative values required, because it is only relevant whether critical section is free or not. However, there is still a queue of waiting process in case multiple processes do call wait operation while the critical section is occupied).
- Counting semaphore:
  - Term is used for generic semaphores that are used with values >1.

Semaphores (14): Further Types of Semaphores

- Weak semaphore:
  - A semaphore where the order of processes in the semaphore queue has no influence on the order in which processes are woken up again (=no FIFO queue discipline).
  - Most OSes/semaphore APIs offer only weak semaphores!

  - Note: sometimes, the term “weak semaphore” refers to the fact, that after being woken up, you need to re-check the condition you were waiting for (and if necessary: do a wait again).

- Strong semaphore:
  - A semaphore using FIFO queue discipline concerning the order in which processes are woken up again.

Semaphores vs. Spinlocks (1)

- Semaphores are nice, because they avoid busy waiting.
  - However, for performing the semaphore operations, system calls are required: System calls are "expensive" (slow, because they involve, e.g., context switch which is time consuming).
    - On single processor/single core system, there are not much alternatives: if wait operation would block, we would anyway need a context switch, because after blocking our process, the scheduler has to give the CPU to other (ready) processes that eventually unblock our waiting process.
    - However, on SMP systems, there may be situations where it is reasonable to avoid "expensive" semaphore operations and to use a busy waiting instead:
      if the busy waiting is shorter than a system call and the other CPUs/cores may execute processes that eventually unblock our waiting process. $\Rightarrow$ Spinlocks ($\rightarrow$next slide)

Semaphores vs. Spinlocks (2)

- Spinlock: Mutual exclusion based on busy waiting (typically, using atomic test-and-set instruction →6-49…) that avoids an "expensive" system call.
  - Reasonable only in SMP systems, where one CPU/core executes a process that is in its critical section and the other CPU/core performs busy waiting (which is faster than a semaphore if the critical section(=busy waiting for it) is short).
    - Spinlocks are typically used by an SMP kernel itself when accessing its kernel data structures shared by the CPU cores.

The Java API for Semaphores

- Since Java 1.5 (or Java 5 respectively), Java supports semaphores that can be used by Java threads.
- Class **`Semaphore`** with corresponding methods **`acquire()`** and **`release()`** provided by Java API package **`java.util.concurrent.Semaphore`**
- Example:

```
import java.util.concurrent.Semaphore;
…
Semaphore sem = new Semaphore(1); // Initial semaphore value: 1
…
try {
  sem.acquire();
  // Start of critical section
  …
  // End of critical section
  sem.release();
} catch (InterruptedException ie) { // Handle deferred cancellation,
  // i.e. someone called interrupt() while we were waiting in
  // sem.acquire() (see "Thread cancellation" in chapter 4)
}
```

Semaphore to be shared by multiple threads

“wait” operation

“signal” operation

POSIX Pthreads API for Semaphores (for info)

Semaphore shared between threads of same process only or between different processes (provided `*sem` memory location needs then to be in shared memory).

Initial value

Predefined data structure for semaphore to be initialised

- Initialise semaphore:
  `int sem_init(sem_t *sem, int pshared, unsigned int value)`
- Signal on semaphore:
  `int sem_post(sem_t *sem)`

Semaphore to signal on

- Wait on semaphore:
  `int sem_wait(sem_t *sem)`

Semaphore to wait on

- Perform semaphore wait only if it will not block, otherwise return immediately with `EAGAIN` errno:
  `int sem_trywait(sem_t *sem)`
- Perform semaphore wait only until absolute time, after that return with `ETIMEDOUT` errno:
  `int sem_timedwait(sem_t *sem, timespec *abstime)`

Pretty useless operation: value is likely already outdated when you do something based on it. Better use atomic operations sem_signal/sem_wait/ sem_trywait/ sem_timedwait

- Get just value of semaphore counter:
  `int sem_getvalue(sem_t *sem,int *sval)`

Pointer to memory location for storing the semaphores value

- Release all ressources having created by OS at **`sem_init`**:
  `int sem_destroy(sem_t *sem)`

Semaphore to deallocate

Self-check questions: Semaphores

- After having read this chapter, you should be able to explain and apply semaphores for process synchronisation to create synchronised concurrent programs.
- In the following, some simple questions are given (and solutions on the following slides): they refer to using semaphores in pseudo code.
  - Try to answer these questions on your own (before looking at the solution).
    - If you are not able to answer: re-read again, check videos, make a comment on Piazza and ask at our weekly meetings.

Semaphores: Question

- How do you have to change (by adding semaphores) the following pseudo code to make the execution of P1 and P2 mutual exclusive?

```
int a = 1;          ← Shared variables

parallel {
    P1() {

        a = a + 1;

    }

    P2() {

        a = 2*a;

    }
}
```

Semaphores: Solution

- Introduce semaphore initialised with 1 (=one process may enter immediately),
- wait() in front of mutual exclusive section,
- signal() at end of mutual exclusive section.

```
int a = 1;
semaphore mutex = mutex.init(1);
parallel {
   P1() {
      mutex.wait();
      a = a + 1;
      mutex.signal();
   }

   P2() {
      mutex.wait();
      a = 2*a;
      mutex.signal();
   }
}
```

Semaphores: Question

- While mutual exclusion of P1 and P2 is nice, the result is still not deterministic (=there is still a race between P1 and P2 possible and depending who wins the race, different results are possible).
- How to you have to change the code (using semaphores) so that **`a=2*a`** in P2 is always executed after **`a=a+1`** in P1?

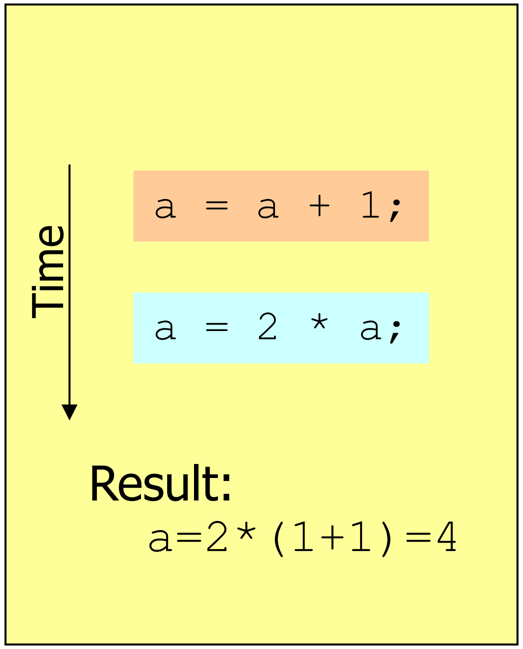
```
int a = 1;

parallel {
  P1() {
    a = a + 1;

  }

  P2() {

    a = 2*a;
  }
}
```



```
a = a + 1;

a = 2 * a;
```

Result:
`a=2*(1+1)=4`

Semaphores: Solution

- Introduce semaphore initialised with 0 (=P2 process has to wait in case it wins the race), i.e. condition synchronisation for condition “P1 finished”.
- wait() in front of a=2*a,
- signal() at end of a=a+1.

```
int a = 1;
semaphore cond = cond.init(0);
parallel {
  P1() {
    a = a + 1;
    cond.signal();
  }

  P2() {
    cond.wait();
    a = 2*a;
  }
}
```

Semaphores: Question

- What is wrong with the following Semaphore-based code that is supposed to achieve mutual exclusion of P1 and P2?

```
int a = 1;
semaphore mutex = mutex.init(0);
parallel {
    P1() {
      mutex.wait();
      a = a + 1;
      mutex.signal();
    }

    P2() {
      mutex.wait();
      a = 2*a;
      mutex.signal();
    }
}
```

Semaphores: Solution

- The semaphore is initialised with 0 and thus both P1 and P2 will go to sleep and as there are no other processing signalling on the semaphore, they will sleep forever.

```
int a = 1;
semaphore mutex = mutex.init(0);
parallel {
  P1() {
    mutex.wait();
    a = a + 1;
    mutex.signal();
  }

  P2() {
    mutex.wait();
    a = 2*a;
    mutex.signal();
  }
}
```

Semaphores: Question

- What is wrong with the following semaphore-based code that is supposed to achieve condition synchronisation so that P2 waits for P1?

```
int a = 1;
semaphore cond = cond.init(1);
parallel {
  P1() {
    a = a + 1;
    cond.signal();
  }

  P2() {
    cond.wait();
    a = 2*a;
  }
}
```

Semaphores: Solution

- The semaphore is initialised with 1 and thus the wait() by P2 will not make P2 go to sleep and hence P2 will not wait for P1.

```
int a = 1;
semaphore cond = cond.init(1);
parallel {
  P1() {
    a = a + 1;
    cond.signal();
  }

  P2() {
    cond.wait();
    a = 2*a;
  }
}
```

Want to play even more computer games? The Return of the Deadlock Empire!

- Browser game: **http://deadlockempire.github.io/**
  - Reminder: in these games always the yellow background line will be executed next.
    - Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.
- Now, play the three "Semaphores" games:
  - Semaphores
  - Producer-Consumer
  - Producer-Consumer (variant)
  - Notes:
    - Initialisation of semaphores not shown there: default value is 0.
    - C# is used: **`.Release()`** operation is our semaphore "signal" operation.
    - **`.Wait(relativeTime)`** is a "wait" operation that blocks only for the given amount of milliseconds.
    - Dequeuing from an empty queue will raise an exception.
    - Do not forget to expand a statement where possible.

If you are stuck, watch Panopto videos with solutions

Problems of Semaphores (1)

- Disadvantage of semaphores:
  - Even though (or maybe just because) they are quite simple, it is easy to create wrong solutions using them.
  - Incorrect solutions may lead to a deadlock of all involved processes (see example on next slides).

Problems of Semaphores (2): Example

- Example for an error made when trying to solve the producer-consumer problem (in comparison to the correct solution on slide 6-68, just two statements are swapped):

```
semaphore S_Prod = S_Prod.init(0);          // no. of produced items
semaphore S_Free = S_Free.init(capacity);  // capacity: buffer size
semaphore S_ME = S_ME.init(1);              // mutual exclusion
```

```
void Producer() {
  while (true) {
    S_ME.wait();
    S_Free.wait();
    /* Produce item */
    S_ME.signal();
    S_Prod.signal();
  }
}
```

Statements swapped

```
void Consumer() {
  while (true) {
    S_Prod.wait();
    S_ME.wait();
    /* Consume item*/
    S_ME.signal();
    S_Free.signal();
  }
}
```

```
void main() {
  parallel { Producer() }{ Consumer() };
}
```

- Problematic example trace with buffer size of 2 shown on next slide…

Problems of Semaphores (3)

```
<S_Prod=0, S_Free=2, S_ME=1>
Producer: S_ME.wait
<S_Prod=0, S_Free=2, S_ME=0>
Producer: S_Free.wait
<S_Prod=0, S_Free=1, S_ME=0>
Producer: S_ME.signal
<S_Prod=0, S_Free=1, S_ME=1>
Producer: S_Prod.signal
<S_Prod=1, S_Free=1, S_ME=1>
Producer: S_ME.wait
<S_Prod=1, S_Free=1, S_ME=0>
Producer: S_Free.wait
<S_Prod=1, S_Free=0, S_ME=0>
Producer: S_ME.signal
<S_Prod=1, S_Free=0, S_ME=1>
Producer: S_Prod.signal
<S_Prod=2, S_Free=0, S_ME=1>
```

```
Producer: S_ME.wait
<S_Prod=2, S_Free=0, S_ME=0>
Producer: S_Free.wait
<S_Prod=2,S_Free=-1, S_ME=0,
S_Free.queue=Producer>
```

Producer sleeps

```
Consumer: S_Prod.wait
<S_Prod=1, S_Free=-1, S_ME=0,
S_Free.queue=Producer>
Consumer: S_ME.wait
<S_Prod=1, S_Free=-1, S_ME=-1,
S_Free.queue=Producer
S_ME.queue=Consumer>
```

Consumer sleeps

Deadlock, both processes are sleeping

Monitors (1)

- Reason for possible problems when using semaphores:
  - Processes (or their developers) themselves are responsible for achieving the synchronisation.
- Hoare (1974) and Brinch Hansen (1975) suggested a higher level synchronisation mechanism:
  - Monitors (has nothing to do with a computer display):
    - A programmer only specifies that synchronisation is required, but not how to achieve it.
      - The implementation is provided by the higher level mechanism.

Monitors (2)

- Monitor construct is based on the encapsulation of resources (comparable to an object-oriented class).
  - A monitor is a collection of variables (=shared resources) and operations to access these variables.
    - Information hiding: resource cannot be directly accessed from outside.
    - Processes can access resource only via operations of the monitor.
    - Only one process at a time can be active in the monitor (i.e. mutual exclusion).
  - Just like object-oriented classes are a programming language concept, monitors are a programming language concept.
    - I.e. the compiler (and not the programmer who is likely to introduce errors) is responsible for achieving the mutual exclusion.
      - E.g. by generating code that internally uses semaphores provided by OS.

Monitors (3): Mutual Exclusion Example

- Mutual exclusive access to a critical region that accesses a shared resource (e.g. a variable) contained in a monitor (pseudocode):

```
monitor myVarMonitor
  private int myResource;
  ...
  public void accessResource() {
    /* critical region
    where variable myResource
    is accessed
    */
  }
}
```

Process A:

```
while(true) {
  myVarMonitor.accessResource();
}
```

Process B:

```
while(true) {
  myVarMonitor.accessResource();
}
```

Call to this operation (and any other operation) of the monitor is mutual exclusive (only one process-call at a time can be active within the monitor).

Monitors (4): Condition Synchronisation

- Achieving mutual exclusion using monitors is really easy and convenient.
  - Compiler and run-time system of a programming language that supports monitors take care of the nasty and error-prone implementation details.
- However, does not allow to synchronise with respect to conditions.
  - I.e. one process wakes up another if a certain condition gets true.
    - Well, you could do this using a Boolean variable `condition` together with operations `setConditionToTrue()` and `isConditionTrue()`: Process B probes `isConditionTrue()` periodically in a busy waiting style until process A calls `setConditionToTrue()`. However, this would be busy waiting – and furthermore this would in fact not work inside the monitor, because process A would not be able to change the condition while process B is probing it (due to the mutual exclusion that a monitor provides!).
- Condition synchronisation provided by monitors:
  - Special Condition variables on which two special operations can be applied:
    - **`.wait`** to put the calling process to sleep until another process calls **`.signal`**.
    - **`.signal`** to wake up one process that has called **`.wait`** on the same condition variable.

Monitors (5): Condition Synchronisation Example

- Pseudocode example that defines a condition variable `myCondition` and uses **`.wait`** and **`.signal`**:

```
monitor example
   condition myCondition;
   ...
   public void op1() {
      ...
      myCondition.wait;  // Blocks the calling process
      ...                // until condition is signalled
   }     // (During this, others can enter the monitor)
   public void op2() {
      ...
      myCondition.signal;// Unblocks the process that
      ...                // did a myCondition.wait
   }
}
```

Even though method execution in a monitor is mutual exclusive, when this **`wait`** blocks, the method is put to sleep and others may then enter the monitor while this **`wait`** is blocked.

What to do after the **`signal`**? We definitely have to avoid that mutual exclusion is violated, i.e. the process that called `op1()` and the process that called `op2()` must not execute at the same time instructions within the monitor!

Monitors (6): Condition Synchronisation

- How to proceed after the signal operation has been issued?

- Three possible solutions:
  - "Signal-and-return" (suggested by Brinch Hansen):
    - Signal is the last statement of a monitor operation, i.e. signalling leaves the monitor.
  - "Signal-and-wait" (suggested by Hoare):
    - The signalling process is suspended until the unblocked formerly waiting process leaves the monitor.
  - "Signal-and-continue" (used by, e.g., Java):
    - The waiting process is only unblocked after the signalling process left the monitor.

Monitors (7): Condition Synchronisation

- **`wait`** and **`signal`** are to some extent like the **`sleep`** and **`wakeup`** operations on slide 6-56 and not that much like semaphores.
- Comparison to **`sleep`** and **`wakeup`**:
  - A monitor's **`signal`** for which no is **`wait`** performed, gets lost just like with **`sleep`** and **`wakeup`**.
    - Note: the problem in the **`sleep`** and **`wakeup`** example was not only that a **`sleep`** got lost, but that a "test-and-sleep" sequence was not atomic.
      - (Note: To motivate semaphores, slide 6-60 claimed for teaching purposes that the problem was a lost signal – but in fact it was also because of the lacking atomicity!)
    - Monitors provide mutual exclusion, hence a "test-and-wait" or "test-and-signal" sequence cannot get interrupted, but is rather atomic!
- Comparison to semaphores:
  - A monitor's **`signal`** for which no **`wait`** is performed, gets lost – in contrast to a semaphore's **`signal`** and **`wait`** operations.
  - Condition variables are not counters in contrast to semaphores – they can only be used for conveying a signal.

Monitors (8): Example Producer-Consumer Problem

- Solution for producer-consumer problem by implementing the shared buffer of size N using a monitor with condition variables:
  - Signal-and-continue semantics is assumed.
  - (Implementation of actual buffer data structure for items and corresponding insert/remove operations not shown.)
  - Note: solution quite similar to buggy sleep & wakeup solution (6-57 & 6-58). However, this time we cannot get interrupted during the **`if`** and the **`wait`** (because of the atomicity provided by a monitor)!

Pseudocode:

```
monitor ProducerConsumerBuffer {
  condition full, empty;
  private int count = 0;
  public void insert(int item) {
    if (count == N) full.wait;
    insert_item(item);
    count = count + 1;
    if (count == 1) empty.signal;
  }
  public int remove() {
   if (count == 0) empty.wait;
   int item = remove_item();
   count = count - 1;
   if (count == N-1) full.signal;
   return item;
  }
}
```

Monitors (9): Example Producer-Consumer Problem

- Producer and consumer processes that call monitor operations and thus produce and consume in a synchronised manner:

```
void producer() {
  int item;
  while (true) {
    item = produce_item();
    ProducerConsumerBuffer.insert(item);
  }
}
```

```
void consumer() {
  int item;
  while (true) {
    item=ProducerConsumerBuffer.remove();
    consume_item(item);
  }
}
```

Monitors in Java (1)

- Monitors are supported by the Java language in order to synchronise Java threads:
  - Every Java object has internally a lock flag.
  - A method that is declared as **`synchronized`** can only be executed if that lock is not set.
    - Otherwise, the thread that wants to execute that method is blocked.
  - If a method that is declared as **`synchronized`** executes, the lock flag gets set and is reset after the method is left.
    - If there are blocked threads that are waiting for the object's lock to be reset, one of them becomes the new "owner" of the lock and may enter the **`synchronised`** method it was waiting for.
  - Mutual exclusion is achieved for all **`synchronized`** methods of the same object.
    - For all other (i.e. non **`synchronised`**) methods of an object, this monitor property does not apply: they may be entered concurrently.

Monitors in Java (2): Example

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

- Typically, you would pass the same instance of such a class to different threads.
  - Due to the **`synchronized`** keyword, access to the counter by different threads will be mutual exclusive.
  - Note: internal lock flag is object specific, not class specific.
    Hence, mutual exclusion is only guaranteed within the same instance of a class.
    - Well, for **`static`** methods, there is also a class specific (=**`static`**) internal lock flag.

Monitors in Java (3): Condition Variables

- There is no special data type for condition variables. Instead, each object has automatically exactly one unnamed (anonymous) condition variable:
  - Predefined methods `this.wait()`, `this.notify()` and `this.notifyAll()` use this unnamed condition variable.
  - These methods can only be called from within **`synchronized`** methods.
  - The notify methods correspond to the previously described signal operations of a monitor. (More on them on the next slides.)
  - The signal-and-continue approach is used for the notify methods, i.e. a waiting method is only unblocked after a signalling method has finished.
  - To make thread cancellation (→chapter 4) possible while a thread is blocked in a `wait()` method, `wait()` will throw an exception when `interrupt()` is called on a thread while waiting.
    - `wait()` must be surrounded by **`try`** … **`catch`**`(InterruptedException)`.

Monitors in Java (4): Condition Variables

- When a thread calls `wait()`:
  - The thread is set to the blocked state.
  - The lock of the object is released.
    - So that other threads can use `notify()`/`notifyAll()` in their `synchronized` methods in order to wake up this waiting thread.
  - The thread is placed in a wait set of threads.
    - Like the lock flag, each instance/each object has its own wait set of threads that are waiting to be notified.
      - One condition variable per object $\Rightarrow$ one wait set per object.

Monitors in Java (5): Condition Variables

- When a thread calls **`notify():`**
  - One arbitrary thread *t* is picked and removed from the wait set of the object.
  - This thread *t* is set from the blocked state to the runnable state.
  - Once, the notifying thread finishes the current method, the lock is released.
  - Now, either the picked waiting thread *t* or any other further runnable thread that competes for the lock flag will become the owner of the lock and resume execution.

  ⇒ Exactly one waiting thread will be woken up.

- When a thread calls **`notifyAll():`**
  - All threads are removed from the wait set of the object and set from the blocked state to the runnable state.
  - Once, the notifying thread finishes the current method, the lock is released.
  - Now, either one of notified waiting threads or any other further runnable thread that competes for the lock flag will become the owner of the lock and resume execution.

  ⇒ All waiting threads will be woken up. (However, as the wait occurs within a synchronized method, these threads will not get immediately executed, but only after another thread that just got the lock releases the lock again, one of these can run.)

Monitors in Java (6): Example Condition Variables

- A example of a monitor that can be used to inform a consumer that an item has been produced:

```
class MyMonitor {
  int noOfItems = 0;
  public synchronized void produce() {
    noOfItems++;
    notify();
  }
  public synchronized void consume() {
    if (noOfItems == 0) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    noOfItems--;
  }
}
```

Note: only one condition variable for each object supported in Java ⇒ Using multiple conditions requires multiple objects and you can then call some of the synchronized methods of these object!

- Typically, you pass the same instance of such a class to different threads.

Monitors in Java (7): Example Condition Variables

- Java monitors uses the signal-and-continue approach ⇒ it may be the case that after the **`notify()`** has been made, the condition on which the **`wait()`**ing thread is waiting has been invalidated in the meantime (i.e. when being woken up, condition is invalid again).

⇒ Enclose the **`wait()`** in a conditional loop, i.e. if we are woken up from the **`wait()`**, we check first whether the condition to continue is still given, otherwise we **`wait()`** again.

```
class MyMonitor {
  int noOfItems = 0;
  public synchronized void produce() {
    noOfItems++;
    notify();
  }
  public synchronized void consume() {
    while (noOfItems == 0) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    noOfItems--;
  }
}
```

- See also: http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--

Self-check questions: Monitors

- After having read this chapter, you should be able to explain and apply monitors for process synchronisation to create synchronised concurrent programs.
- In the following, some simple questions are given (and solutions on the following slides): they refer to usage of the Java monitor concept.
  - Try to answer these questions on you own (before looking at the solution).
    - If you are not able to answer: re-read again, check videos, make a comment on Piazza and ask at our weekly meetings.

Monitors with Java: Question

- How do you have to change the following Java class to make the method `advance()` mutual exclusive (so that no two threads that use the same instance of that class can execute that method at the same time)?

```
class Clock {
    private int time = 0;

    public void advance() {
        time = time + 1;
    }

    public int getTime() {
        return time;
    }
}
```

Solution on next slide

Monitors with Java: Solution

- Add keyword **`synchronized`** to the respective method:

```
class Clock {
    private int time = 0;

    public synchronized void advance() {
        time = time + 1;
    }

    public int getTime() {
        return time;
    }
}
```

Monitors with Java: Question

- How do you have to change the following Java class to make the caller of the method `goodNight()` go to sleep if field `time` has the >= 22?

```
class Clock {
    private int time = 0;

    public synchronized void advance() {
        time = time + 1;
    }

    public void goodNight() {





    }

    public int getTime() {
        return time;
    }
}
```

Solution on next slide

Monitors with Java: Solution

- Use `wait()` (which must be called from the context of synchronized method).

```
class Clock {
    private int time = 0;

    public synchronized void advance() {
        time = time + 1;
    }

    public synchronized void goodNight() {
        if (time >= 22) { // Even better would be while instead of if
            try {
                wait();
            } catch (InterruptedException e) { }
        }
    }

    public int getTime() {
        return time;
    }
}
```

Monitors with Java: Question

- How do you have to change the following Java method `wakeUpOneSleeper()` to wake up just one thread that is sleeping due a `goodNight()` method call that has been made on the same instance of this class?

```
class Clock {
    …
    public synchronized void goodNight() {
        if (time >= 22) { // Even better would be while instead of if
            try {
                wait();
            } catch (InterruptedException e) { }
        }
    }

    public void wakeUpOneSleeper() {

    }
}
```

Solution on next slide

Monitors with Java: Solution

- Use `notify()` (which must be called from the context of synchronized method).

```
class Clock {
    …
    public synchronized void goodNight() {
        if (time >= 22) { // Even better would be while instead of if
            try {
                wait();
            } catch (InterruptedException e) { }
        }
    }

    public synchronized void wakeUpOneSleeper() {
        notify();
    }
}
```

Monitors with Java: Question

- How do you have to change the following Java method `wakeUpAllSleepers()` to wake up all threads that are sleeping due a `goodNight()` method call that has been made on the same instance of this class?

```
class Clock {
    …
    public synchronized void goodNight() {
        if (time >= 22) { // Even better would be while instead of if
            try {
                wait();
            } catch (InterruptedException e) { }
        }
    }

    public synchronized void wakeUpOneSleeper() {
        notify();
    }
    public void wakeUpAllSleepers() {

    }
}
```

Solution on next slide

Monitors with Java: Solution

- Use `notifyAll()` (which must be called from the context of synchronized method).

```
class Clock {
    …
    public synchronized void goodNight() {
        if (time >= 22) { // Even better would be while instead of if
            try {
                wait();
            } catch (InterruptedException e) { }
        }
    }

    public synchronized void wakeUpOneSleeper() {
        notify();
    }
    public synchronized void wakeUpAllSleepers() {
        notifyAll();
    }
}
```

Want to play more computer games? The Deadlock Empire awakens! (1)

- Browser game: **http://deadlockempire.github.io/**
  - Reminder: in these games always the yellow background line will be executed next.
    - Sometimes, there are white background lines, but these are in fact supposed to be transparent background lines.

- Note that some games use the C# API for synchronisation (which we did not cover):
  - Hence, you probably want to skip “High-Level Synchronization Primitives” (and “The Final Stretch”),
  - but you should still be able to play the games described on the two next slides!

If you are stuck in the games on the next two slides, watch Panopto videos with solutions

Want to play more computer games? The Deadlock Empire awakens! (2)

- Now play the "Locks" games which are somewhat similar to monitors without condition variables, but you have to explicitly write **`Monitor.Enter(mutex)`** and **`Monitor.Exit(mutex)`** using an object such as **`mutex`** that you pass in and use as lock.

- Notes:
  - Even though all games have a critical section, in none of the games it will be possible to be in the critical sections at the same time (they are well protected by the locks). Instead the goal in the respective game is:
    - In "Insufficient Lock" you win by reaching line **`Debug.Assert(false);`**
    - In "Deadlock" two lock objects are used (which you do not have in Monitors: there each monitor has exactly one implicit lock that is automatically used) and you win by provoking a deadlock, i.e. each process waiting on the other.
    - In "A More Complex Thread", you win as well if manage to provoke a deadlock, i.e. achieving that none of the processes can progress.

Want to play more computer games? The Deadlock Empire awakens! (3)

- Now play the "Condition Variables" game which is in fact about monitors with condition variables:
  - **`Monitor.Enter(mutex)`** and **`Monitor.Exit(mutex)`** use the lock of the **`mutex`** object are just like entering and leaving a **`synchronized`** method in Java.
  - **`Monitor.Wait(mutex)`** waits on the condition variable of the **`mutex`** object and is thus like the **`wait()`** in Java.
    - In particular (just like in Java), a **`wait()`** that blocks does unlock the lock while waiting.
  - **`Monitor.pulseAll(mutex)`** is like the **`notifyAll()`** in Java.

- Note:
  - In the only existing game "Condition Variables" you win, if you manage to **`queue.Dequeue();`** on an empty queue.

6.6 Alternative Approaches

- Implicit synchronisation (≈mutual exclusion with monitors) in OpenMP (→4.5):

```
void update(int value) {
   …
   #pragma omp critical
   {
     count += value
   }
   …
}
```

Following {…} block contains critical section for which mutual exclusion is achieved.

- Functional programming languages (e.g. Scala, Erlang) offer a different paradigm than imperative (or procedural) languages in that they do not allow shared global variables with changing state (leading to race conditions).
  - There is increasing interest in functional languages such as Erlang and Scala: support threads/concurrency extremely well!
  - Might be the solution to multicore programming, but functional language paradigm hard to learn once you have been spoiled with imperative programming.
- Further alternative approach: Message passing →next slides.
  - While semaphores and monitors work only locally, works also across network.

6.7 Classical Problems of Synchronisation

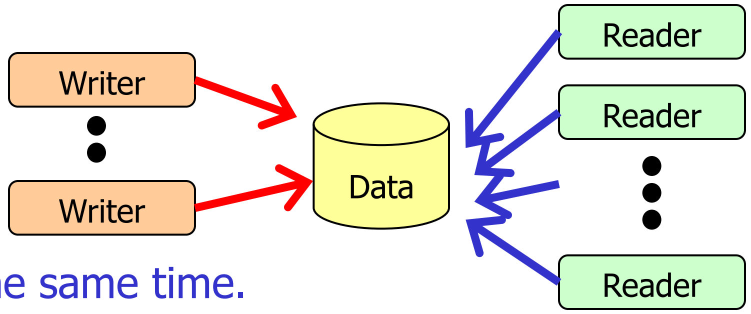
- Like the Producer-Consumer problem, there are further classical synchronisation problems:
  - Bounded-Buffer problem,
  - Readers-Writers problem,
  - Dining-Philosophers problem,
  - Sleeping-Barber problem.
- At first glance, not all of these problems are relevant for real-life SW development. Nevertheless, they are –like the producer-consumer problem– used to study synchronisation problems and solutions.
  - Every computer scientist/software engineer should have heard about them, hence they are presented in the following. (However, without a solution.)

Bounded-Buffer Problem

- In fact, more or less the Producer-Consumer problem.
  - As the name implies, refers to that variant of the Producer-Consumer problem that uses a bounded buffer, i.e. buffer size is neither ∞ nor zero.
  - However, the Bounded-Buffer problem itself does not involve the actual production and consumption of items, but only that part of the solution of the Producer-Consumer problem that implements the actual buffer data structure and the synchronised access to it.
    - (=condition synchronisation and mutual exclusion).
    - Solution: See previous slides on semaphores, monitors, message passing.

Readers-Writers Problem

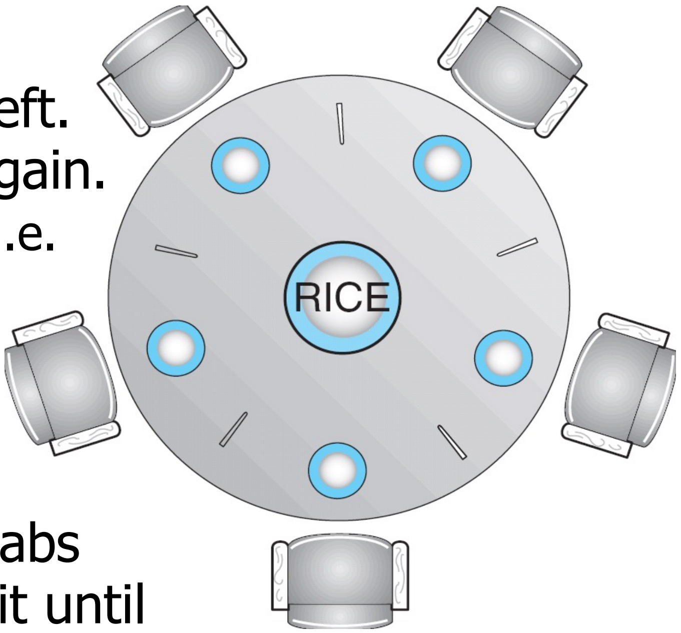
- A data set (e.g. a file in a file system or a table in a data base) is shared among a number of concurrent processes:
  - Readers processes: only read the data set; they do not perform any updates.
  - Writers processes: can write (update) the data set; during write, the data is in an inconsistent state.



- Readers-Writers problem:
  - Avoid that multiple writers are writing at the same time.
  - Allow multiple readers to read at the same time.
  - Avoid that readers are reading and a writer is writing at the same time.
- Variants of the Readers-Writers problem (starvation possible):
  - "First Readers-Writers" problem: readers have priority, i.e. even if a writer is already waiting for readers to finish reading, new readers may join reading.
  - "Second Readers-Writers" problem: writers have priority, i.e. if a writer is already waiting for readers to finish reading, new readers may not join reading.

Dining-Philosophers Problem

- Five independent philosophers share a table with five shared chopsticks: each philosopher thinks for a while, then decides to eat for a while, think, eat, and so on.
- For eating, a philosopher has to grab one chopstick from the right and one from the left. After eating, the chopsticks are put down again.
  - Grabbing a pair of chopsticks is not atomic, i.e. while grabbing the chopstick from the right, the philosopher to the left may grab the second chopstick.



- A solution must deal with:
  - Deadlocks (imagine each philosopher grabs its right chopstick and does not release it until (s)he gets the chopstick to the left as well),
  - Starvation / fairness: (a philosopher that wants to eat, finally gets access to the left and right chopstick).

6.8 Summary

- A critical section of code accesses shared resources or data.
- Interacting processes need to be synchronised to avoid race conditions: mutual exclusion or condition synchronisation.
- Software-based solutions (Peterson algorithm), Hardware-based solutions (disabling interrupts, atomic instructions), or Operating System-based (Semaphores) / Programming Language-based solutions (Monitors) possible.
  - Higher-level operating system/programming language-based solutions are implemented using hardware solutions.
    - Operating system/programming language-based solutions restrict the possible schedules (by blocking some processes which are then not considered anymore by the scheduler unless they get unblocked).
  - Unless you are using Java, semaphores are used in practise instead of monitors (because not many languages support monitors):
    - E.g. using the semaphore API provided by the POSIX Pthreads library (POSIX semaphore calls not presented in this course).
    - Since Java (1.)5, the Java API supports also semaphores.

Not discussed in this course (but in the Silberschatz book)

- Atomic transactions:
  - A set of operations (e.g. updating data) shall be either committed successfully or when it was (for whatever reason) unsuccessful, it is not simply aborted, but all the changes that have already applied as part of this set of operations are rolled back. (I.e. either it is executed completely or it is not executed at all.)

- Atomic transactions are typically discussed in a course on database systems.
  - However, we will in a later chapter on filesystems cover an example of some sort of atomic transactions when talking about log-based/journaling file systems.