9th Slide Set
Operating Systems

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Faculty of Computer Science and Engineering
christianbaun@fb2.fra-uas.de

Learning Objectives of this Slide Set

- At the end of this slide set You know/understand...
  - what **critical sections** are
  - how **race conditions** occur
    - the **consequences** of race conditions
  - the difference between **communication** and **cooperation**
  - what **synchronization** is
    - different options to specify the execution order of processes via **signaling**
    - how critical sections can be secured via **blocking**
  - what **problems** may arise from blocking
    - the difference between **starvation** and **deadlocks**
  - the **conditions**, which must be fulfilled for deadlock occurrence
  - how **resource graphs** illustrate the relationships between processes and resources
  - how **deadlock detection with matrices** works

Exercise sheet 9 repeats the contents of this slide set which are relevant for these learning objectives

Inter-Process Communication (IPC)

- Processes do not only carry out operations on data, but also:
  - call each other
  - wait for each other
  - coordinate each other
  - In short: They must **interact** with each other
- Important questions regarding **inter-process communication** (IPC):
  - How can a process to transmit information to others?
  - How can multiple processes access shared resources?

Question: What is the situation here with threads?

- For threads, the same challenges and solutions exist as for inter-process communication with processes
- Only the communication between the threads of a process is no problem because they operate in the same address space

Critical Sections

- If multiple processes are executed in parallel, the processes consist of...
  - **Uncritical sections**: The processes do not access common data or carry out only read operations on common data
  - **Critical sections**: The processes carry out read and write operations on common data
    - Critical sections may not be processed by multiple processes at the same time
- In order to allow processes to access shared memory ($\Longrightarrow$ common data), the operating system must provide **mutual exclusion**

Critical Sections – Example: Print Spooler

Process X | | **Process Y**

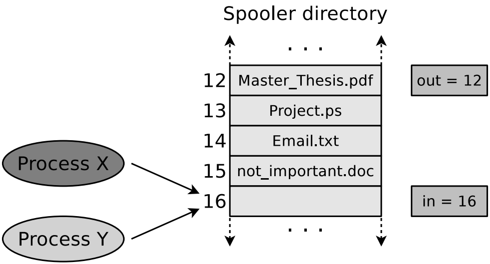
```
next_free_slot = in; (16)
```

Process switch

```
next_free_slot = in; (16)
Store entry in next_free_slot; (16)
in = next_free_slot + 1; (17)
```

Process switch

```
Store entry in next_free_slot; (16)
in = next_free_slot + 1; (17)
```



- The spooling directory is consistent
  - But the entry of **process Y** was overwritten by **process X** and got lost
- Such a situation is called **race condition**

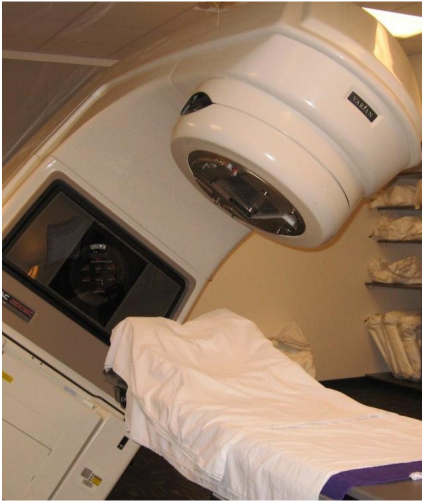
Race Condition

- **Unintended race condition** of 2 processes, which want to modify the value of the same record
  - The result of a process depends on the order or timing of other events
  - Frequent reason for bugs, which are hard to locate and fix
- Problem: The occurrence of the symptoms depends on different events
  - In each test run, the symptoms may vary or disappear
- Race conditions can be avoided with the **semaphore** concept ($\Longrightarrow$ slide set 10)

Therac-25: Race Condition with tragic Result (1/2)

- Therac-25 is a linear particle accelerator for the radiation therapy of cancer tumors
- Mid-1980s: In the United States some accidents happened because of poor programming and quality assurance
  - Some patients got an up to 100 times increased radiation dose

Image source: Google image search



Therac-25: Race Condition with tragic Result (2/2)

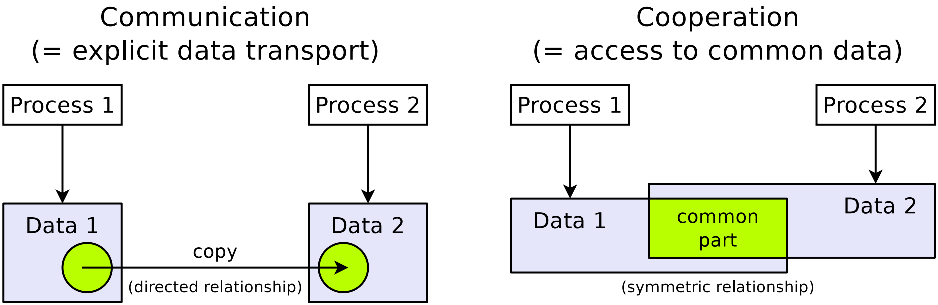
An Investigation of the Therac-25 Accidents. Nancy Leveson, Clark S. Turner
IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41
`http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html`

- 3 patients died because of *bugs*
- 2 patients died because of a race condition, which resulted in inconsistent settings of the device, causing an increased radiation dose
  - The control process did not synchronize correctly with the user interface process
  - The bug only occurred in case the operator was too fast
  - During testing, the error did not occur, because experience (routine) was required to operate the device this fast

Image search: `http://www.ircrisk.com/blognet/`

Process interaction has 2 aspects...

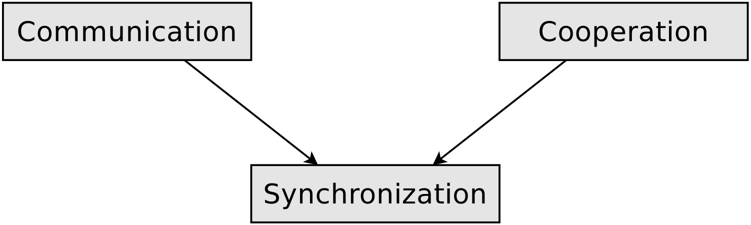
1. Functional aspect: **communication** and **cooperation**



2. Temporal aspect: **synchronization**

Forms of Interaction

- Communication and cooperation base on synchronization
  - Synchronization is the most elementary form of interaction
    - Reason: communication and cooperation need a synchronization between the interaction partners to obtain correct results
  - Therefore, we first discuss the **synchronization**

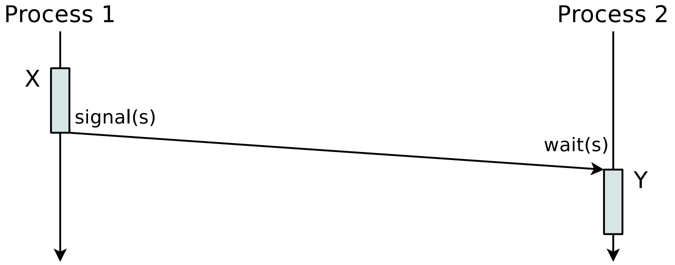


Synchronization

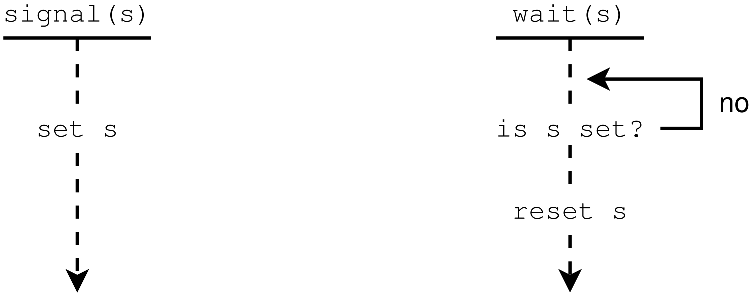
- Synchronization
  - Signaling
    - Busy waiting
    - Signal and wait
    - Rendezvous
    - Many-process signaling
    - Many-process rendezvous with barriers
  - Blocking
    - Lock and unlock

Signaling

- Special form of synchronization
- Used to specify an **execution order**
- Example: Section **X** of process 1 must be executed **before** section **Y** of process 2
  - The `signal` operation signals that process 1 has finished section **X**
  - Perhaps, process 2 must wait for the signal of process 1



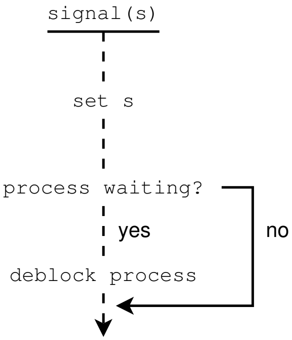
Most Simple Form of Signaling (Busy Waiting)



- Procedure: Busy waiting at the signaling variable `s`
  - Computing time of the CPU is wasted because it is again and again occupied by the process

Signal and Wait

- Better concept: Block process 2 until process 1 has finished section **X**
  - Advantage: Lesser CPU workload
  - Disadvantage: Only a single process can wait



Signal and Wait with JAVA

- In JAVA the `signal()` operation is called `notify()`
  - `notify()` unblocks a waiting process
- `wait()` blocks the execution of a process
- The keyword `synchronized` is used in JAVA to implement mutual exclusion of all methods of an object, which are labeled by this keyword

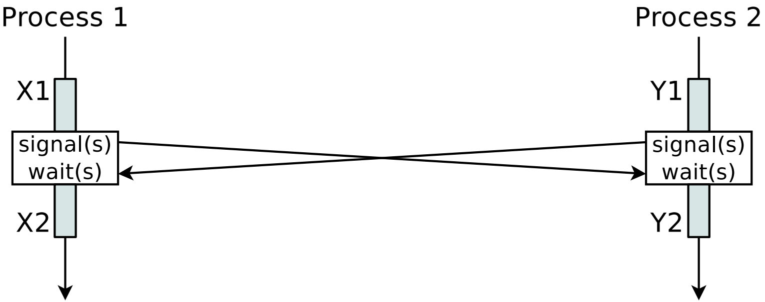
```java
class Signal {
    private boolean set = false;      // signal variable

    public synchronized void signal() {
        set = true;
        notify();                     // unblocks the waiting process
    }

    public synchronized void wait() {
        if (!set)
        wait();                       // waits for the signal
        set = false;
    }
}
```

In this JAVA example and in the following JAVA examples some parts are missing...

main method, Object instantiation with threads...
Further information: `http://docs.oracle.com/javase/7/docs/api/?java/lang/Thread.html`

Rendezvous

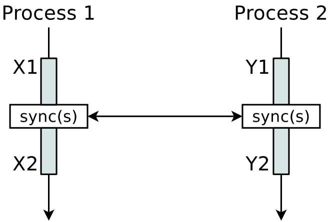
- If `wait()` and `signal()` are executed symmetrically, the sections `X1` and `Y1` are executed before the sections `X2` and `Y2`



- In this case, the processes 1 and 2 are **synchronized**

Rendezvous – sync()

- Implementation of a synchronization (rendezvous) with JAVA
- sync() combines the operations wait() and signal()



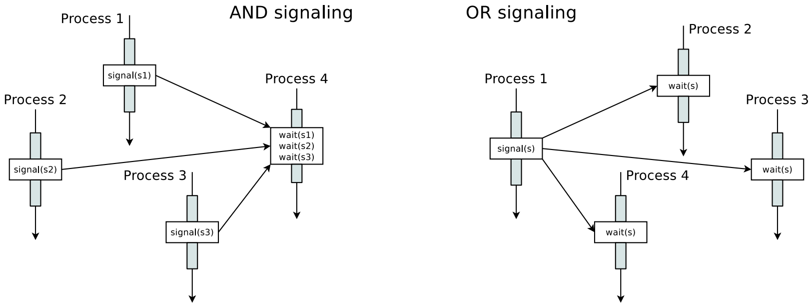

```java
class Synchronisierung {
    private boolean set = false;      // signal variable

    public synchronized void sync() {
        if (set == false) {           // this is the 1st process
            set = true;               // signal that the process is in ready state
            wait();                   // wait for the other process
        } else {                      // this is the 2nd process
            set = false;              // set the signal variable back to value false
            notify();                 // unblock the waiting process
        }
    }
}
```

Helpful Resources about theis topic. . .

- Carsten Vogt, **Nebenläufige Programmierung – Ein Arbeitsbuch mit UNIX/LINUX und JAVA**, Hanser (2012) P.137-141
- David Flanagan, **JAVA in a Nutshell**, german translation of the 3rd edition, O'Reilly (2000), P.166

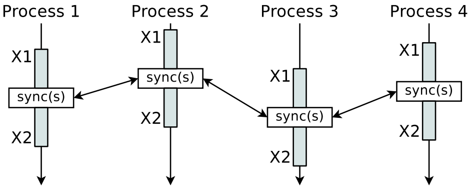
Many-Process Signaling

- Signaling with > 2 processes
- Examples:
  - **AND signaling**:
    - A process cannot executed further until multiple processes call a signal
  - **OR signaling**:
    - Several processes wait for a signal
    - If the signal is called, one of the waiting processes is unblocked



Many-Process Rendezvous with Barrier

- A barrier synchronizes the involved processes at one point

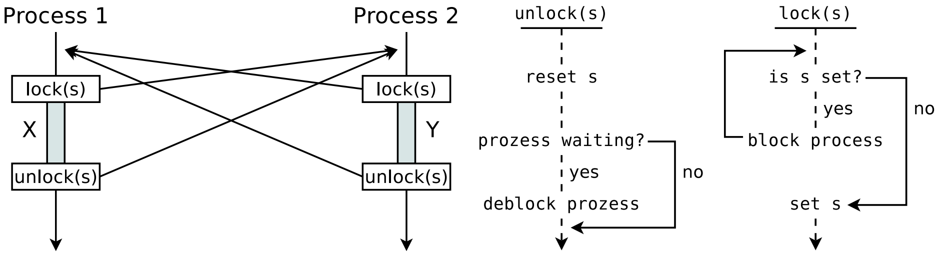


- Only when all processes have reached the synchronization point, their execution may continue

```
class BarrierenSynchronisation {
    private int sum = p;                // number of processes
    private int counter = 0;            // number of processes in waiting state

    public synchronized void sync() {
        counter = counter + 1;
        if (counter < sum) {            // some processes are still missing
            wait();                     // wait for missing processes
        } else {                        // all processes arrived
            notifyAll();                // unblock all processes in waiting state
            counter = 0;
        }
    }
}
```

Blocking (Lock and Unlock)

- **Critical sections** can be secured with the operations **lock** and **unlock**



- Blocking ensures that the processing of 2 critical sections does not overlap
  - Example: Critical Sections **X** of process 1 and **Y** of process 2

Difference between Signaling and Blocking

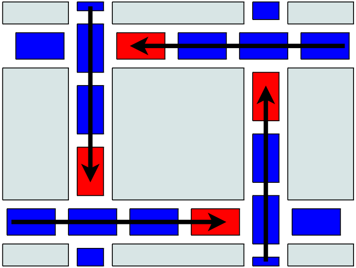
- **Signaling** specifies the execution order
  - Example: Execute section A of process 1 before section B of 2
- **Blocking** secures critical sections
  - The execution order of the critical sections of the processes is not specified
  - It is just ensured that the execution of critical sections does not overlap

```
class Sperre {
    private boolean locked = false;       // signal variable

    public synchronized void sperre() {
        while(locked)
            wait();                       // wait missing process
            locked = true;                // lock process
    }

    public synchronized void entsperren() {
        locked = false;                   // unlock process
        notify();                         // notify all process in waiting state
    }
}
```

Problems caused by Blocking

Image Source: Google Image Search

- **Starvation**
  - If a process never calls `unlock()` after `lock()`, blocked processes must wait indefinitely
- **Deadlock**
  - If several processes wait for resources, locked by each other, they lock each other
  - Because all processes, which are involved in the deadlock, must wait forever, no one can raise an event to solve the issue



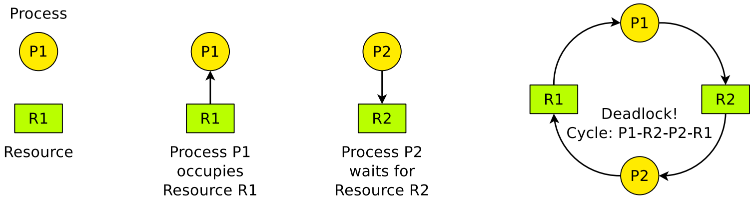
Conditions for Deadlock Occurrence

System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78
http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

- A deadlock situation can arise if these conditions are all fulfilled
  - **Mutual exclusion**
    - At least 1 resource is occupied by exactly 1 process or is available $\Longrightarrow$ non-sharable
  - **Hold and wait**
    - A process, which currently occupies at least 1 resource, requests additional resources which are being held by other processes
  - **No preemption**
    - Resources, which are occupied by a process can not be deallocated by the operating system, but on released by the holding process voluntarily
  - **Circular wait**
    - A cyclic chain of processes exists
    - Each process requests a resource, which is occupied by the next process in the chain
- If one of these conditions is not fulfilled, no deadlock can occur

Resource Graph

- The relationship of processes and resources can be illustrated with directed graphs
- Resource graphs can be used to model deadlock model
  - The nodes of a resource graph are:
    - **Processes**: Are shown as circles
    - **Resources**: Are shown as rectangles
  - An edge from a process to a resource means:
    - The process is blocked because it waits for the resource
  - An edge from a resource to a process means:
    - The process occupies the resource



Resource Graph – Example

- 3 processes exist
  - P1, P2 and P3
- Each process requests 2 resources and releases them afterwards

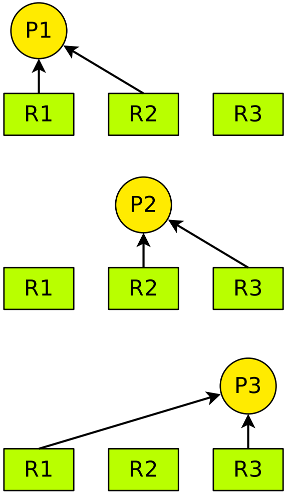
Process **P1**	Process **P2**	Process **P3**
Request R1	Request R2	Request R3
Request R2	Request R3	Request R1
Release R1	Release R2	Release R3
Release R2	Release R3	Release R1

Example from Tanenbaum. Moderne Betriebssysteme. Pearson Studium. 2003

Sources

A good description of resource graphs provides the book **Betriebssysteme – Eine Einführung**, *Uwe Baumgarten, Hans-Jürgen Siegert*, 6th edition, Oldenbourg Verlag (2007), chapter 6

No concurrency: P1 $\Rightarrow$ P2 $\Rightarrow$ P3

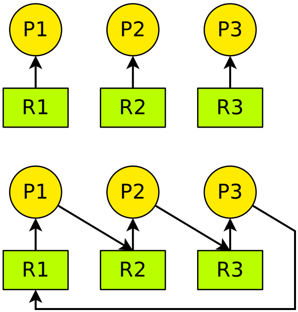


- P1 requests R1
- P1 requests R2
- P1 releases R1
- P1 releases R2

- P2 requests R2
- P2 requests R3
- P2 releases R2
- P2 releases R3

- P3 requests R3
- P3 requests R1
- P3 releases R3
- P3 releases R1
- **No Deadlock**

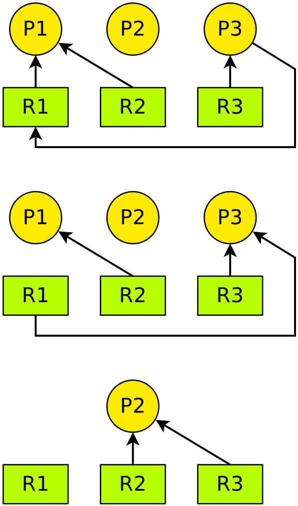
Concurrency with a bad Sequence



- P1 requests R1
- P2 requests R2
- P3 requests R3

- P1 requests R2
- P2 requests R3
- P3 requests R1
- **Deadlock** because of cyclic chain

Concurrency with a better Sequence



- P1 requests R1
- P3 requests R3
- P1 requests R2
- P3 requests R1

- P1 releases R1
- P1 releases R2
- P3 releases R1
- P3 releases R3

- P2 requests R2
- P2 requests R3
- P2 releases R2
- P2 releases R3

Deadlock Detection with Matrices

- If only a single resource per resource class (scanners, CD burners, printers, etc.), exists, deadlocks can be identified via graphs
- If multiple copies of a resource exist, a matrices-based algorithm can be used
- We specify 2 vectors
  - **Existing resource vektor**
    - Indicates the number of existing resources of each class
  - **Available resource vektor**
    - Indicates the number of free resources of each class
- Additionally 2 matrices are required
  - **Current allocation matrix**
    - Indicates, which resources are currently occupied by the processes
  - **Request matrix**
    - Indicates, which resource the processes would like to occupy

Deadlock Detection with Matrices – Example (1/2)

Source of the example: Tanenbaum. Moderne Betriebssysteme. Pearson. 2009

Existing resource vektor $= \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$

- 4 resources of class 1 exist
- 2 resources of class 2 exist
- 3 resources of class 3 exist
- 1 resource of class 4 exist

Available resource vektor $= \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$

- 2 resources of class 1 are available
- 1 resource of class 2 is available
- No resources of class 3 are available
- No resources of class 4 are available

Current allocation matrix $= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$

- Process 1 occupies 1 resource of class 3
- Process 2 occupies 2 resources of class 1 and 1 resource of class 4
- Process 3 occupies 1 resource of class 2 and 2 resources of class 3

Request matrix $= \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$

- Process 1 is blocked, because no free resources of class 4 exist
- Process 2 is blocked, because no free resources of class 3 exist
- **Process 3 is not blocked**

Deadlock Detection with Matrices – Example (2/2)

- If process 3 finished executing, it deallocates its resources

$$\text{Available resource vektor} = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix} \qquad \text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

- 2 resources of class 1 are available
- 2 resources of class 2 are available
- 2 resources of class 3 are available
- No resources of class 4 are available

- Process 1 is blocked, because no free resources of class 4 exist
- **Process 2 is not blocked**

- If process 2 finished executing, it deallocates its resources

$$\text{Available resource vektor} = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix} \qquad \text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

- **Process 1 is not blocked** $\Longrightarrow$ no deadlock in this example

Conclusion about Deadlocks

- Sometimes the possibility of deadlocks occurrence is accepted
  - What matters is how important a system is
    - A deadlock, which statistically occurs every 5 years, is not a problem in a system, which crashes because of hardware failures or other software problems one time per week
- Deadlock detection is complicated and causes overhead
- In all operating systems, deadlocks can occur:
  - Full process table
    - No new processes can be created
  - Maximum number of inodes allocated
    - No new files or directories can be created
- The probability that this happens is low, but $\neq 0$
  - Such potential deadlocks be accepted because an occasional deadlock is not as troublesome as the otherwise necessary restrictions (e.g. only 1 running process, only 1 open file, more overhead)