OPERATING SYSTEMS

Process Interaction

Prof. Dr. Oliver Hahm

2024-12-19

AGENDA

- Process Interaction
- Inter-Processes Communication (IPC)
- Process Synchronization
- Process Cooperation

COMPANY CONTACT EXCHANGE

- **For whom?** Students who are looking for a position for the practical phase
- **When:** 15.01.2025 from 17:00 to 19:00
- **Where?** HoST, Hungener Str. 6, Building B, Room 05
- **Procedure:** 4 companies present their projects, followed by time for contact and one-on-one discussions We were able to win the following companies for the company contact exchange:
  - Cap Gemini
  - Coherent Mainz, DILAS Diodenlaser GmbH
  - Deutsche Bundesbank
  - FES Frankfurt

Questions? Answered by the *Praxisreferat*

Updates can be found on campUAS in the courses
Practical phase Computer Science and Computer Science - Mobile Applications

PROCESS INTERACTION

Why do processes need to interact?

INTERPROCESS COMMUNICATION (IPC)

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

- In many cases processes do **not** operate isolated on separated data
- Processes will often...
  - **call** each other,
  - **wait** for each other, or
  - **coordinate** with each other
- They must interact with each other
- Important questions regarding interprocess communication (IPC):
  - How can a process transmit information to other processes?
  - How can multiple processes access shared resources?

COMMUNICATING THREADS

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

What about threads?

- Essentially threads are facing the same problems and challenges
- However, the solutions can often be simpler because threads operate in the same address space

CRITICAL SECTIONS

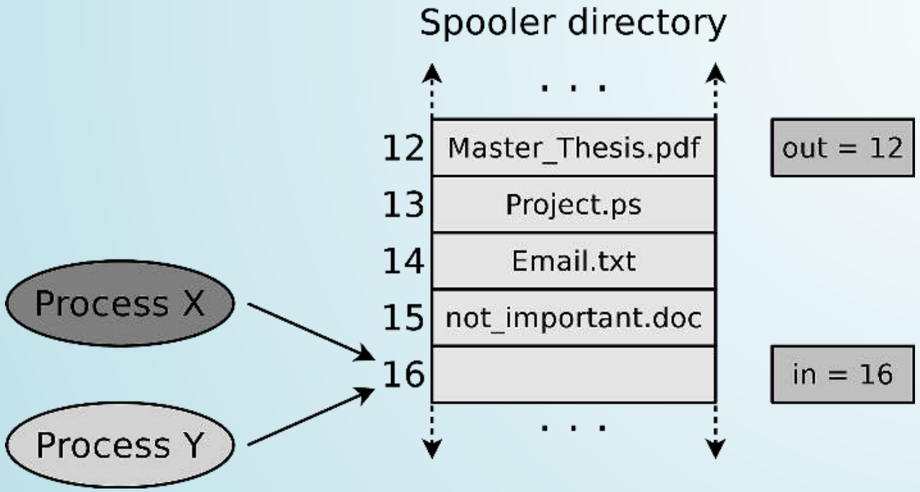
- If multiple processes access **shared resources**, i.e., common data, they contain critical sections
  - Only one process may enter this section at a time ($\Rightarrow$ it must be protected against concurrent access)
  - It appears as an atomic operation to the outside
  - **Uncritical sections**: The processes do not access shared data or carry out only read operations on shared data
- The OS must provide mechanisms for mutual exclusion

RACE CONDITION

- If the process' behaviour depends on the order of multiple code paths, it is called a race condition
  - 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
  - The symptoms may be different or disappear with each test run
- Race conditions can be avoided with the **semaphore** concept

CRITICAL SECTIONS – EXAMPLE: PRINT SPOOLER

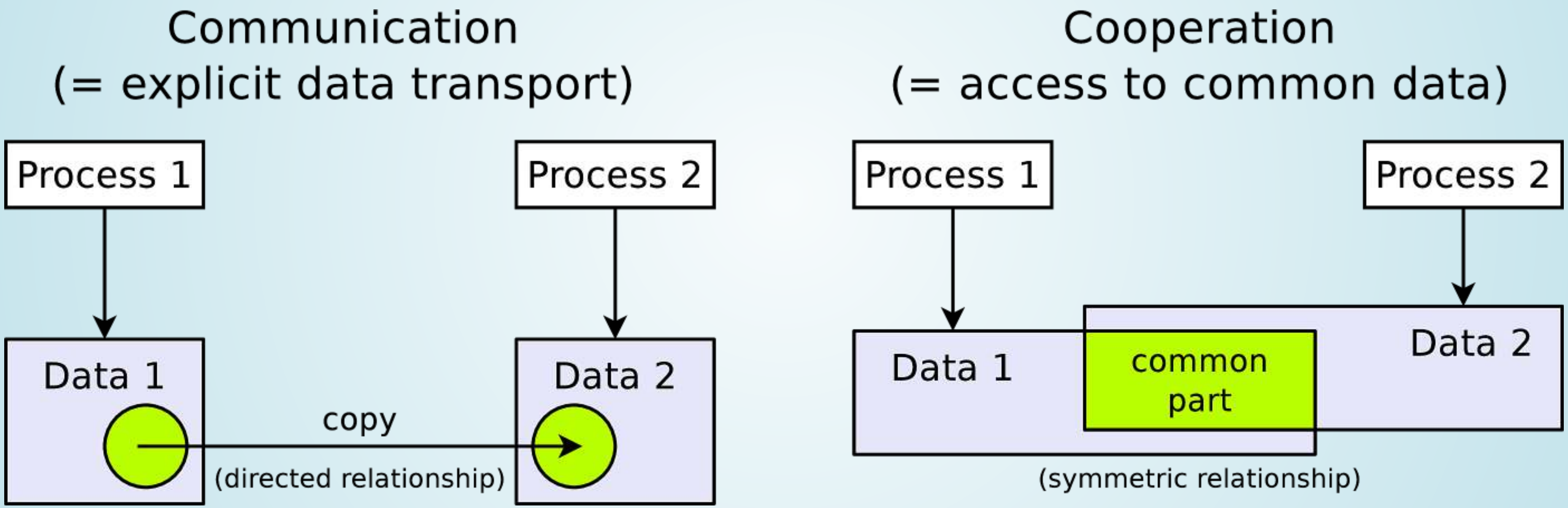
Process X	Process Y
`next_free_slot = in;` (Result: **16**)	
Process switch	
	`next_free_slot = in;` (Result: **16**)
	`Store record in next_free_slot;` (Result: **16**)
	`in = next_free_slot + 1;` (Result: **17**)
Process switch	
`Store record in next_free_slot;` (Result: **16**)	
`in = next_free_slot + 1;` (Result: **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**

COMMUNICATION VS. COOPERATION

- Interprocess communication has 2 aspects:
  - Functional aspect: **communication** and **cooperation**
  - Temporal aspect: **synchronization**



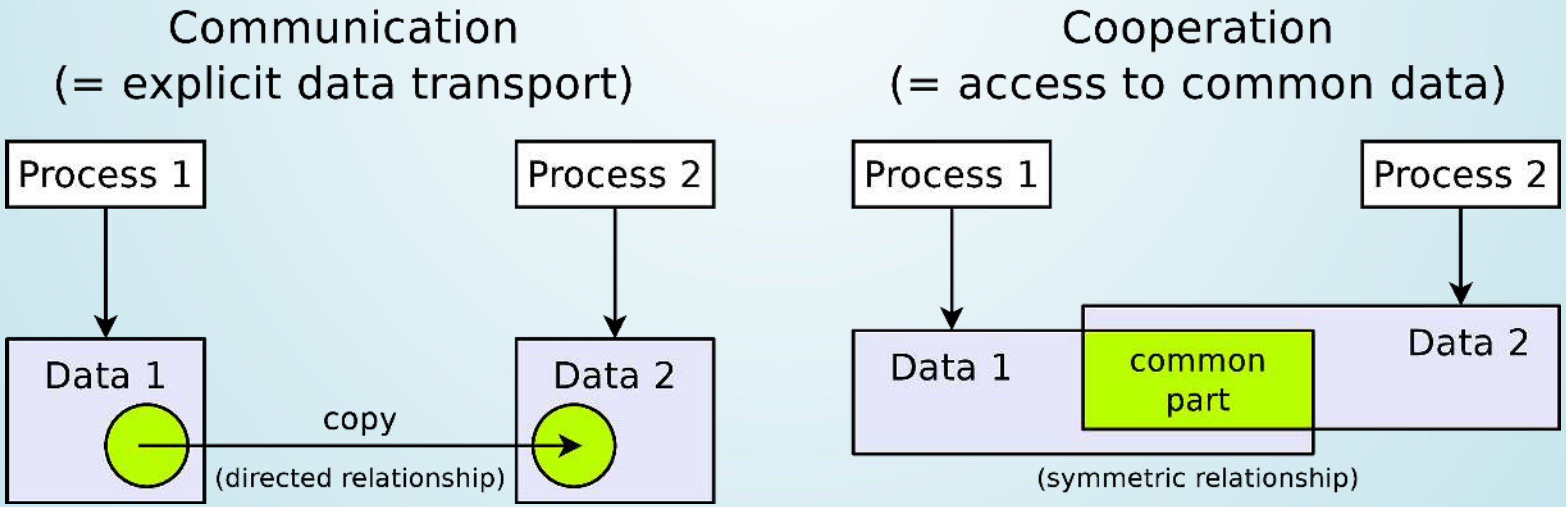
- **Communication** and **cooperation** base on **synchronization**

INTER-PROCESSES COMMUNICATION (IPC)

How can processes communicate?

COMMUNICATION OF PROCESSES

- Types of IPC
  - Files
  - Signals/Flags
  - Shared Memory
  - Message Queues
  - Pipes
  - Sockets



FILES

- A resource stored in the → **file system** which can be accessed by multiple processes
- **Linux**
  - File descriptors represent file handles
  - Part of the **POSIX** API
  - Per default every process owns three file descriptors (`stdin`, `stdout`, and `stderr`)
  - File descriptors can be used for, e.g., reading, writing, seeking, or truncating a file
- **RIOT**
  - Virtual File System (VFS) may be implemented by various backends
  - Not all IoT devices provide persistent memory
  - If available, persistent memory is often realized on flash memory → wear leveling is required

SIGNALS AND FLAGS

- Notify another process about the occurrence of an event
- **Linux**
  - **POSIX** signals
  - Standardized messages to trigger a certain behaviour
  - The receiver process gets interrupted
  - If a signal is unhandled by the receiver, it will terminate
- **RIOT**
  - Thread flags
  - The receiver needs to wait for a flag
  - Optional kernel feature
  - Notify threads of conditions in a race-free and allocation-less way

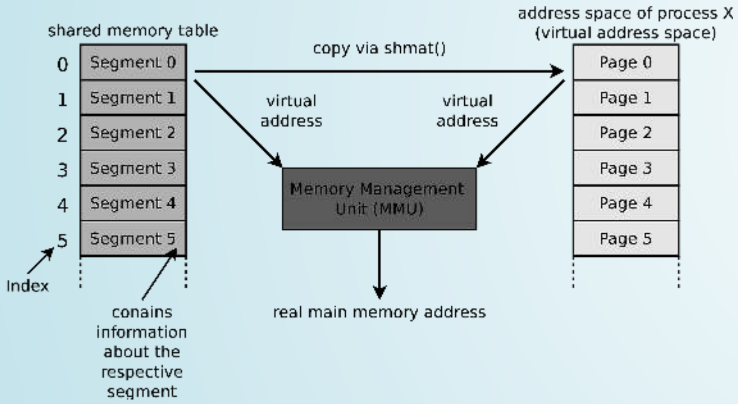
SHARED MEMORY

- **IPC** via **shared memory** is also called memory-based communication
- **Shared memory segments** are memory areas which can be accessed by multiple processes
  - These memory areas are *mapped* in the address space of multiple processes
- Coordination (→ **synchronization**) between the processes accessing the shared memory is required

RIOT Since most microcontrollers do not provide a → **MMU** all processes can typically access all memory regions …

SHARED MEMORY IN LINUX/UNIX

- Linux/UNIX operating systems contain a **shared memory table**, which contains information about the existing shared memory segments
  - This information includes: Start address in memory, size, owner (username and group) and privileges
- Shared memory objects are accessed similar to files

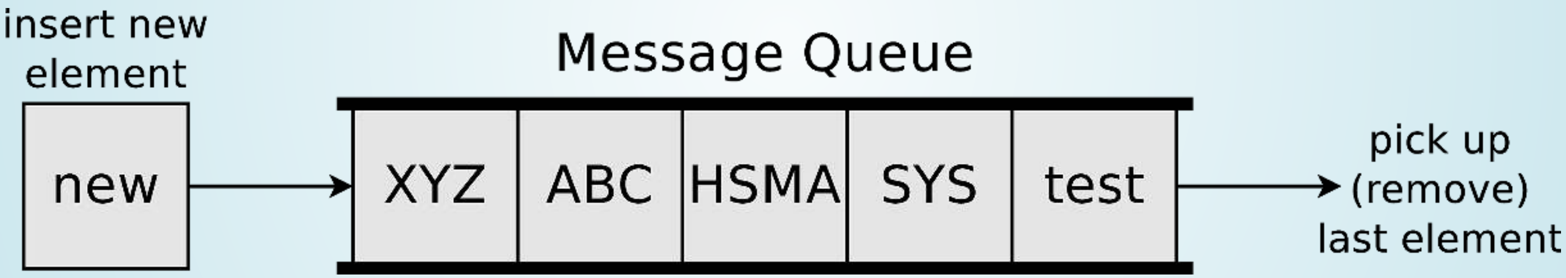


- A shared memory segment is always addressed via its index number in the shared memory table
- **Advantage**: A shared memory segment which is not attached to a process is not erased by the operating system automatically

When the operating system is rebooted, the shared memory segments and their contents are lost

MESSAGE QUEUES

- Are linked lists with messages
- Operate according to the **FIFO** principle
- Processes can store data inside and picked them up from there
- **Benefit**: Even after the termination of the process which created the message queue the data inside the message queue stays available



MESSAGE QUEUES

- **Linux**
  - **POSIX** and System V message queues
  - Queues are named and can be shared via this name between processes
  - Message have priorities
- **RIOT**
  - Kernel messages and mailboxes
  - Optional feature
  - Blocking and non-blocking API available
  - A thread may create a message buffer for queuing
  - Mailboxes can be accessed by multiple processes

ANONYMOUS PIPES

- In Linux pipes are created with the system call `pipe()`
  - The kernel creates an ⟶ *inode* and two **file descriptors** (*handles*)
  - Processes access the access identifiers with `read()` and `write()` system calls (or standard library functions) similar to files
- When child processes are created with `fork()`, the child processes also inherit access to the file descriptors
- **Anonymous pipes** allow process communication only between closely related processes
  - Only processes, which are closely related via `fork()` can communicate with each other via anonymous pipes
  - If the last process, which has access to an anonymous pipe, terminates, the pipe gets erased by the operating system

Overview of the pipes in Linux/UNIX: `lsof | grep pipe`

NAMED PIPES

- Processes, which are not closely related with each other, can communicate via **named pipes**
  - These pipes can be accessed by using their names
    - They are created in C by: `mkfifo("<pathname>",<permissions>)`
  - Any process, which knows the name of a pipe, can use the name to access the pipe and communicate with other processes
- The operating system ensures **mutual exclusion**
  - At any time, only a single process can access a pipe
- Named pipes are not erased automatically by the operating system (unlike anonymous pipes)

DIFFERENT TYPES OF SOCKETS

- **Connectionless sockets** (= **datagram sockets**)
  - Use the Transport Layer protocol UDP
  - Advantage: Better data rate as with TCP
    - Reason: Lesser overhead for the protocol
  - Drawback: Segments may arrive in wrong sequence or may get lost
- **Connection-oriented sockets** (= **stream sockets**)
  - Use the Transport Layer protocol TCP
  - Advantage: Better reliability
    - Segments cannot get lost
    - Segments always arrive in the correct sequence
  - Drawback: Lower data rate as with UDP
    - Reason: More overhead for the protocol

USING SOCKETS

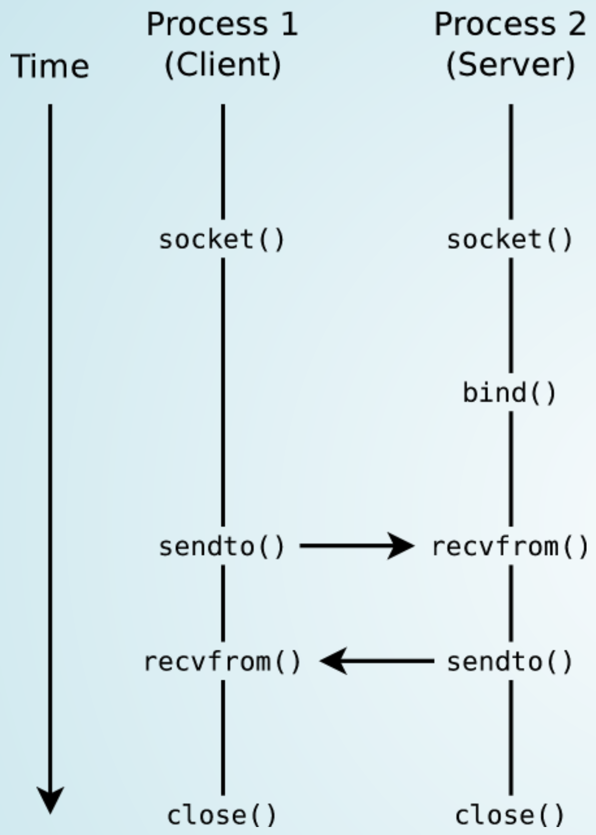
FRANKFURT UNIVERSITY OF APPLIED SCIENCES

- Almost all major operating systems support sockets
  - Advantage: Better portability of applications
- Functions for communication via sockets:
  - Creating a Socket:
    `socket()`
  - Binding a socket to a port number and making it ready to receive data:
    `bind()`, `listen()`, `accept()` and `connect()`
  - Sending/receiving messages via the socket:
    `send()`, `sendto()`, `recv()` and `recvfrom()`
  - Closing of a socket:
    `shutdown()` or `close()`

Overview of the sockets in Linux/UNIX: `netstat -n` or `lsof | grep socket`

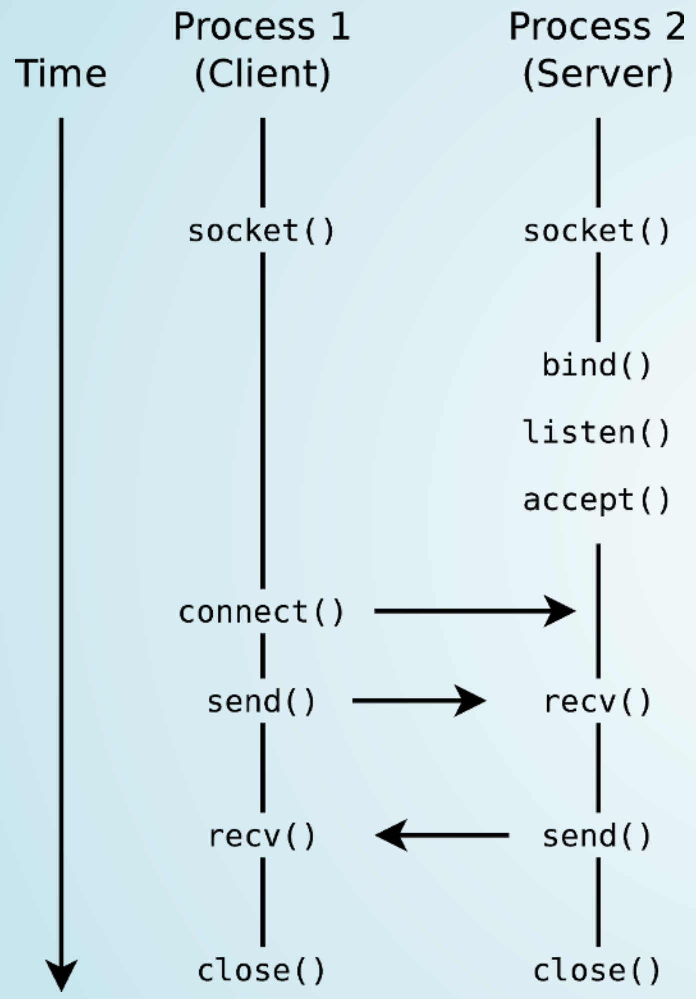
Examples of Interprocess communication via sockets (TCP and UDP) in Linux can be found on the website of this course

CONNECTION-LESS SOCKETS (UDP)



- **Client**
  - Create socket (`socket`)
  - Send (`sendto`) and receive data (`recvfrom`)
  - Close socket (`close`)
- **Server**
  - Create socket (`socket`)
  - Bind socket to a port (`bind`)
  - Send (`sendto`) and receive data (`recvfrom`)
  - Close socket (`close`)

CONNECTION-ORIENTED SOCKETS (TCP)



- **Client**
  - Create socket (`socket`)
  - Connect client with server socket (`connect`)
  - Send (`send`) and receive data (`recv`)
  - Close socket (`close`)
- **Server**
  - Create socket (`socket`)
  - Bind socket to a port (`bind`)
  - Make socket ready to receive (`listen`)
    - Set up a queue for connection requests. Specifies the number of connection requests, which can be stored in the queue
  - Server accepts connections (`accept`)
    - Fetch the first connection request from the queue
  - Send (`send`) and receive data (`recv`)
  - Close socket (`close`)

COMPARISON OF COMMUNICATION SYSTEMS

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

	Shared Memory	Message Queues	(anon./named) Pipes	Sockets
Scheme	Memory-based	Message-based	Stream-based	Message-based
Bidirectional	yes	no	no	yes
Platform independent	no	no	no	yes
Processes relation required	no	no	for anon. pipes	no
Common address space required	yes	yes	yes	no
Bound to a process	no	on	yes	yes
Automatic synchronization	no	yes	yes	yes

- Advantages of message-based communication versus memory-based communication:
  - The operating system takes care about the synchronization of accesses $\Longrightarrow$ comfortable
  - Can be used in distributed systems without a shared memory
  - Better portability of applications

Storage can be integrated via network connections

- This allows memory-based communication between processes on different independent systems
- The problem of synchronizing the accesses also exists here

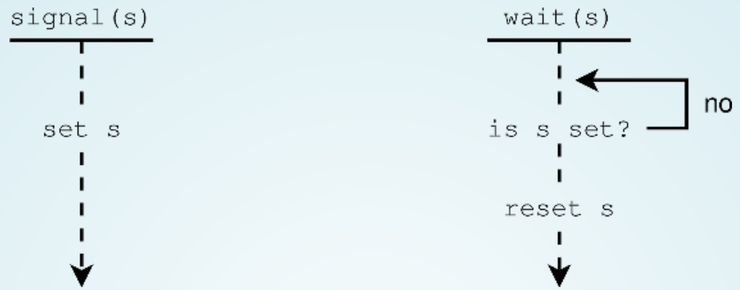
PROCESS SYNCHRONIZATION

What is required if process P_A needs to process X before process P_B can do Y?

SIGNALING

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

MOST SIMPLE FORM OF SIGNALING (BUSY WAITING)

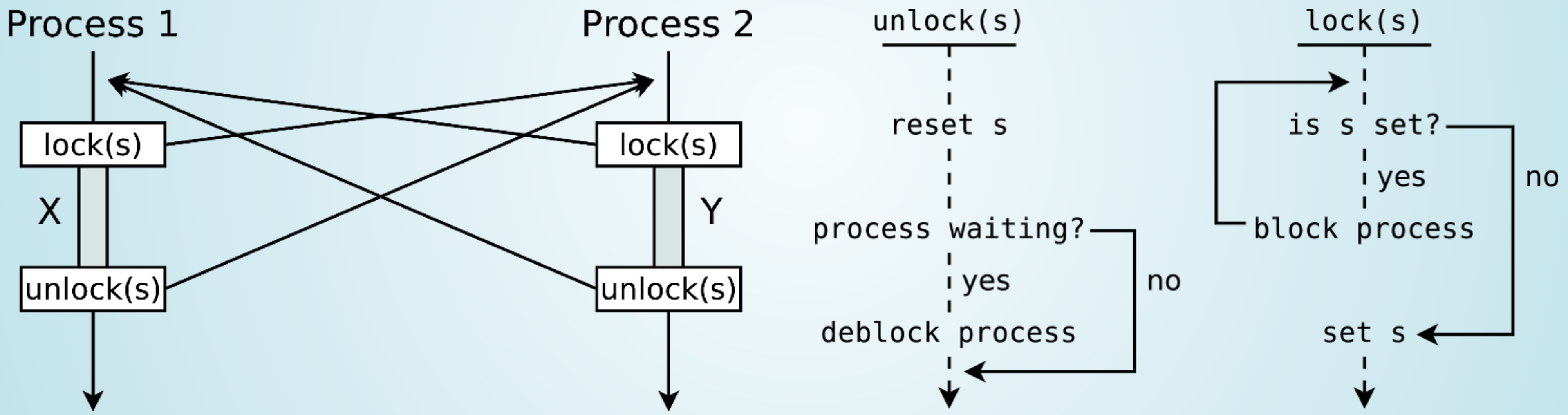


- The figure shows busy waiting at the signal variable `s`
  - The signal variable can be located in a local file, for example
  - **Drawback:** CPU resources are wasted, because the `wait` operation occupies the processor at regular intervals
- This technique is also called **spinlock** or **polling**

What can be done if the order of execution is not important?

LOCKING

- In order to protect **critical sections**, i.e., no overlap in their execution, locking can be used
- In contrast to **signaling** the execution order is not specified
- The necessary operations are `lock` and `unlock`



- **Example**: Critical Sections **X** of process P_A and **Y** of process P_B

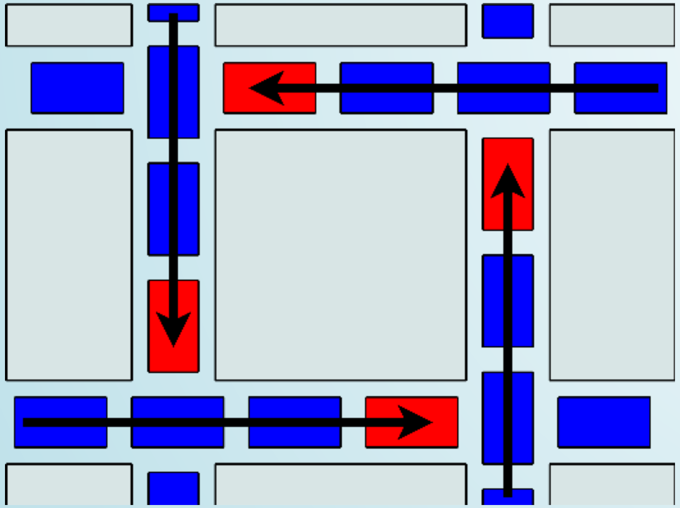
DIFFERENCE BETWEEN SIGNALING AND LOCKING

- **Signaling** specifies the execution order
  Example: Execute section X of process P_A before section Y of P_B
- **Locking** 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

What may go wrong?

PROBLEMS CAUSED BY LOCKING

- **Starvation**
  - If a process does never remove a lock, the other processes need to wait infinitely long for the release
- **Deadlock**
  - If several processes wait for resources, locked by each other, they lock each other mutually
  - Because all processes, which are involved in the deadlock, must wait forever, no one can initiate an event that resolves the situation



Source: https://i.redd.it/vvu6v8pxvue11.jpg
(author and license: unknown)

CONDITIONS FOR DEADLOCK OCCURRENCE

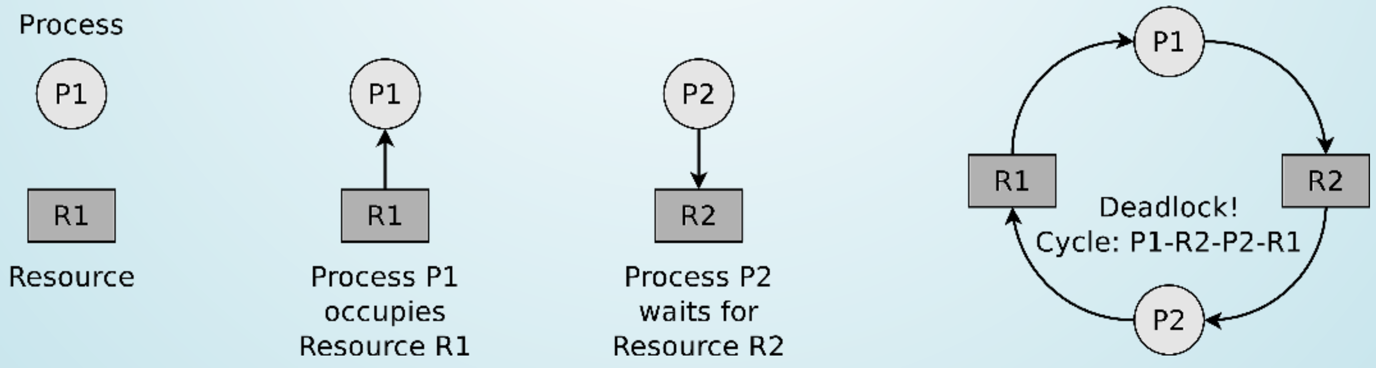
- A deadlock situation can arise if these conditions are all fulfilled
  - **Mutual exclusion**
    - At least one resource is either occupied by exactly one process or is available $\Longrightarrow$ non-sharable resource
  - **Hold and wait**
    - A process, which currently occupies at least one resource, requests additional resources which are being held by another process
  - **No preemption**
    - Resources occupied by a process cannot be deallocated by the OS but only be released by the holding process voluntarily
  - **Circular wait**
    - A cyclic chain of processes exists
    - Each process requests a resource that the next process in the chain occupies.
- Only if **all** of these conditions are fulfilled a deadlock occurs

DEADLOCK HANDLING

- Ignore it ($\rightarrow$ Ostrich algorithm)
- Detect and correct it:
  - Terminate one or more processes
  - Rollback a process
  - Preempt resource usage
- Avoid it

RESOURCE GRAPHS

- The relations of processes and resources can be visualized using directed graphs
- In this way, deadlocks can also be modeled
  - 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



DEADLOCK DETECTION WITH MATRICES

Limitations of deadlock detection with resource graphs

Only individual resources (i.e., no copies) can be represented

- If multiple copies of a resource exist, an algorithm based on matrices can be used
- We specify two vectors
  - **Existing resource vector**
    - Indicates the number of existing resources of each class
  - **Available resource vector**
    - Indicates the number of free resources of each class
- Additionally two 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 – EXAMPLE

- If process 3 finished execution, it deallocates its resources

$$\text{Available resource vector} = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

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

- Two resources of class 1 are available
- Two resources of class 2 are available
- Two 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 execution, it deallocates its resources

$$\text{Available resource vector} = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$

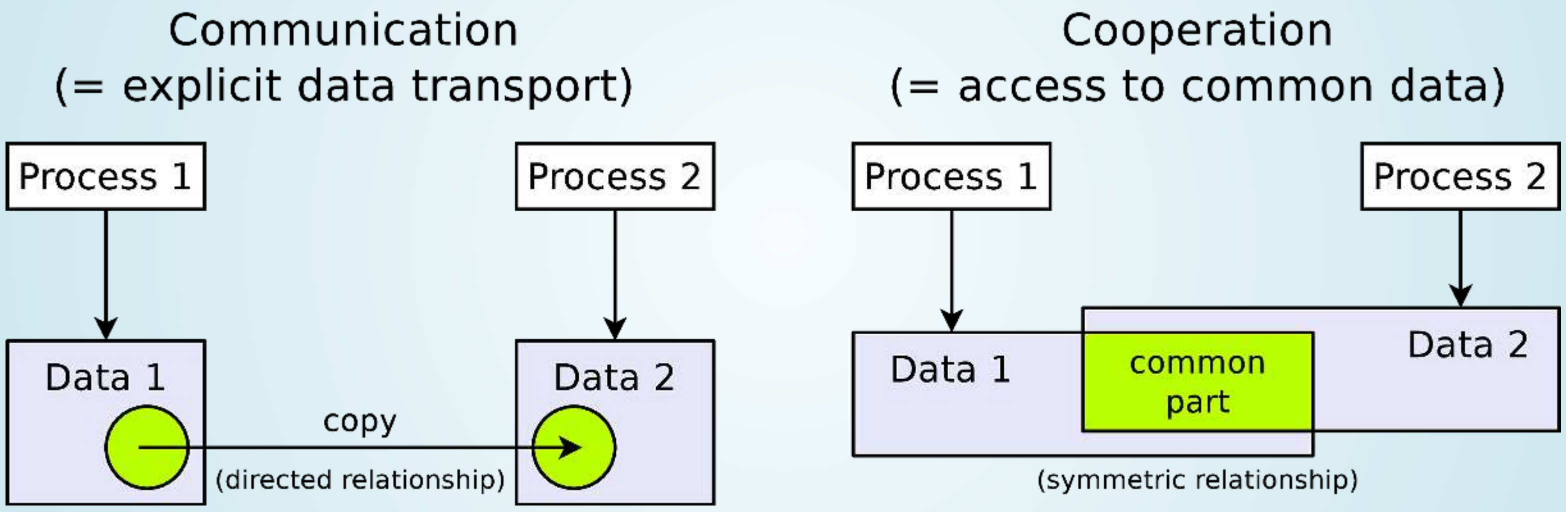
$$\text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

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

PROCESS COOPERATION

COOPERATION

- Cooperation
  - Semaphor
  - Mutex



SEMAPHORE

- In order to protect (lock) critical sections not only the already discussed locks can be used but also semaphores
- First published in 1965 by **Edsger W. Dijkstra**
- A semaphore is a counter lock **S** with operations **P(S)** and **V(S)**
  - **V** comes from the dutch *verhogen* = raise
  - **P** comes from the dutch *proberen* = try (to reduce)
- These **access operations are atomic** $\Longrightarrow$ can not be interrupted
- May allow multiple processes accessing the critical section

Cooperating sequential processes. Edsger W. Dijkstra (1965)
https://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF

SEMAPHORE ACCESS OPERATIONS (1/3)

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

A Semaphore consists of 2 Data Structures

- `COUNT`: An **integer, non-negative counter variable**.
  Specifies how many processes can pass the semaphore now without getting blocked
- A `waiting room` for the processes, which **wait** until they are allowed to pass the semaphore
  The processes are in `blocked` state until they are transferred into `ready` state by the operating system when the semaphore allows to access the critical section

- Initialization: First, a new semaphore is created or an existing one is opened
  - For a new semaphore, the counter variable is initialized at the beginning with a non-negative initial value

```
// apply the INIT operation on semaphore SEM
SEM.INIT(unsigned int init_value) {
    // initialize the variable COUNT of Semaphor SEM
    // with a non-negative initial value
    SEM.COUNT = init_value;
}
```

SEMAPHORE ACCESS OPERATIONS (2/3)

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

- **P operation** (*reduce*): It checks the value of the counter variable
  - If the value is 0, the process becomes blocked
  - If the value > 0, it is reduced by 1

```
SEM.P() {
    // if the counter variable = 0, the process becomes blocked
    if (SEM.COUNT == 0)
    < block >
    // if the counter variable is > 0, the counter variable
    // is decremented immediately by 1
    SEM.COUNT = SEM.COUNT - 1;
}
```

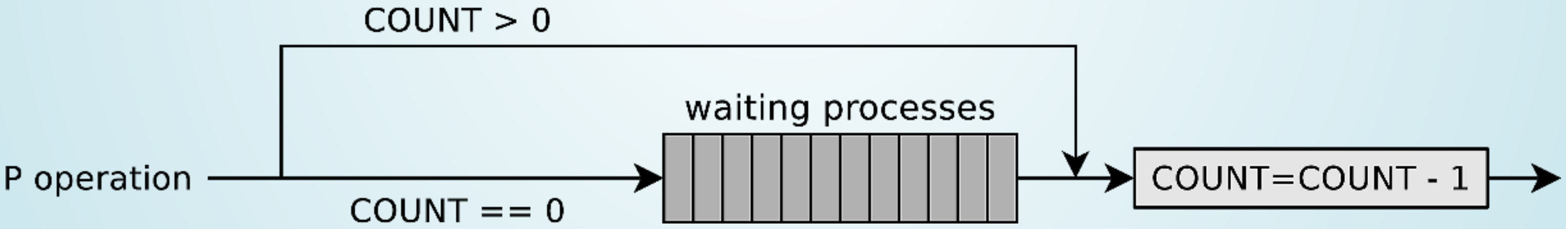


Image Source: Carsten Vogt

SEMAPHORE ACCESS OPERATIONS (3/3)

- **V operation** (*raise*): It first increases the counter variable by value 1
  - If processes are in the waiting room, one process gets unblocked
  - The process, which just got unblocked, continues its P operation and first reduces the counter variable

```
SEM.V() {
    // counter variable = counter variable + 1
    SEM.COUNT = SEM.COUNT + 1;
    // if processes are in the waiting room, one gets unblocked
    if ( < SEM waiting room is not empty > )
    < unblock a waiting process >
}
```

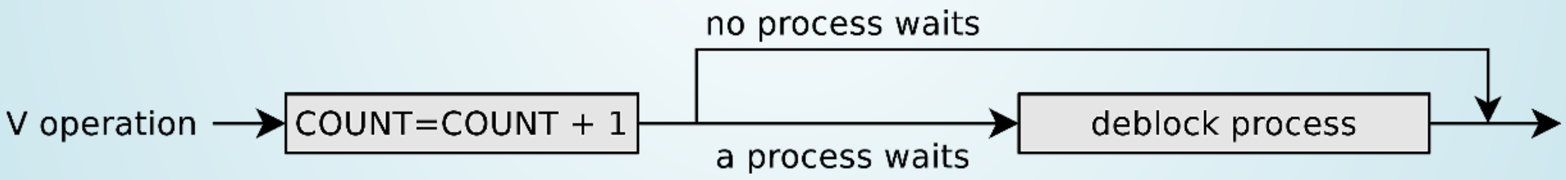


Image Source: Carsten Vogt

PRODUCER/CONSUMER EXAMPLE

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

```
typedef int semaphore;         // semaphores are of type integer
semaphore filled = 0;          // counts the number of occupied locations in the buffer
semaphore empty  = 8;          // counts the number of empty locations in the buffer
semaphore mutex  = 1;          // controls access to the critial sections

void producer (void) {
    int data;
    while (TRUE) {                // infinite loop
        createDatapacket(data);     // create data packet
        P(empty);                   // decrement the empty locations counter
        P(mutex);                   // enter the critical section
        insertDatapacket(data);     // write data packet into the buffer
        V(mutex);                   // leave the critical section
        V(filled);                  // increment the occupied locations counter
    }
}

void consumer (void) {
    int data;
    while (TRUE) {                // infinite loop
        P(filled);                  // decrement the occupied locations counter
        P(mutex);                   // enter the critical section
        removeDatapacket(data);     // pick data packet from the buffer
        V(mutex);                   // leave the critical section
        V(empty);                   // increment the empty locations counter
        consumeDatapacket(data);    // consume data packet
    }
}
```

SEMAPHORES IN LINUX (SYSTEM V)

- The semaphore concept of Linux differs from the Dijkstra concept
  - The counter variable can be incremented or decremented with a P or V operation by more than value 1
  - Multiple access operations on different semaphores can be carried out in an atomic way
- Linux systems maintain a semaphore table, which contains references to arrays of semaphores
  - Individual semaphores are addressed using the table index and the position in the group

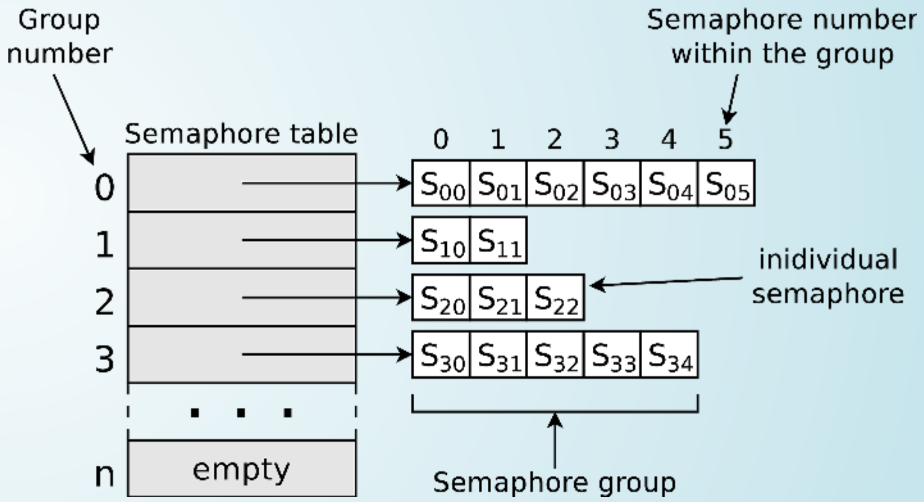


Image Source: Carsten Vogt

SYSTEMS CALLS FOR SYSTEM V SEMAPHORES

Linux/UNIX operating systems provide three system calls for working with *System V* semaphores

- `semget()`: Create new semaphore or a group of semaphores or open an existing semaphore
- `semctl()`: Request or modify the value of an existing semaphore or of a semaphore group or erase a semaphore
- `semop()`: Carry out P and V operations on semaphores
- Information about existing semaphores (**System V**) provides the command `ipcs`

MUTEXES

- If the Semaphore feature of counting is not required a simplified alternative, the **mutex** can be used instead
  - **Mutexes** (derived from **Mut**ual **Ex**clusion) are used to protect critical sections, which are allowed to be accessed by only **a single process** at any given moment
    - Mutexes can only have two states: **occupied** and **not occupied**
    - Mutexes have the same functionality as **binary semaphores**

Several implementations of the mutex concept exist

- **C standard library**: `mtx_init`, `mtx_unlock` (**V operation**), `mtx_lock` (**P operation**), `mtx_trylock`, `mtx_timedlock`, `mtx_destroy`
- **POSIX threads**: `pthread_mutex_init`, `pthread_mutex_unlock`, `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_timedlock`, `pthread_mutex_destroy`
- **C standard library** (Sun/Oracle Solaris): `mutex_init`, `mutex_unlock`, `mutex_lock`, `mutex_trylock`, `mutex_destroy`

MONITOR AND ERASE IPC OBJECTS

- Information about existing **System V** shared memory segments, **System V** message queues, and **System V** semaphores provides the command `ipcs`
- The easiest way to erase such shared memory segments, message queues and semaphores from the command line is the command `ipcrm`

```
ipcrm [-m shmid] [-q msqid] [-s semid]
[-M shmkey] [-Q msgkey] [-S semkey]
```

- **POSIX** memory segments and **POSIX** semaphores can be inspected and manually erased in the directory `/dev/shm`
- **POSIX** message queues can be inspected and manually erased in the directory `/dev/mqueue`

SUMMARY

You should now be able to answer the following questions:

- What are **critical sections** and **race conditions**?
- What is **synchronization**?
- How can critical sections be secured via **blocking**?
- Which problems are described by (**starvation** and **deadlocks**)?
- How does **deadlock detection with matrices** work?
- What are different options to implement **communication** between processes?
- How can critical sections be protected via **semaphores** (and **mutex**)?