

Operating Systems

Process Interaction

Prof. Dr. Oliver Hahm

Frankfurt University of Applied Sciences
Faculty 2: Computer Science and Engineering
oliver.hahm@fb2.fra-uas.de
<https://teaching.dahahm.de>

December 06, 2022

Agenda

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

Agenda

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

Interprocess Communication (IPC)

- In many cases processes do **not** operate isolated on separated data
- Processes will typically...
 - 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 others?
 - How can multiple processes access shared resources?

Question: 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

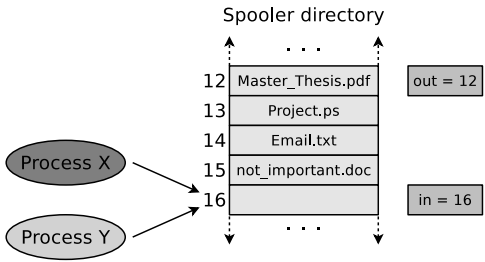
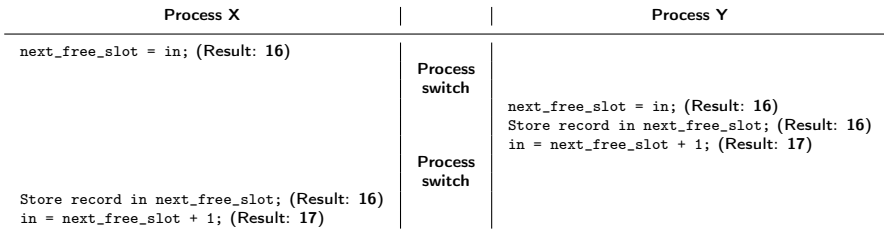
Critical Sections

- If multiple processes run in parallel, the processes consist of...
 - **Uncritical sections:** The processes do not access shared data or carry out only read operations on shared data
 - **Critical sections:** The processes carry out read and write operations on shared data
 - Critical sections must not be processed by multiple processes at the same time
- For processes to be able to access a shared memory (\implies common data), the operating system must provide **mutual exclusion**

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
 - The symptoms may be different or disappear with each test run
- Race conditions can be avoided with the **semaphore** concept
(\implies slide 41)

Critical Sections – Example: Print Spooler

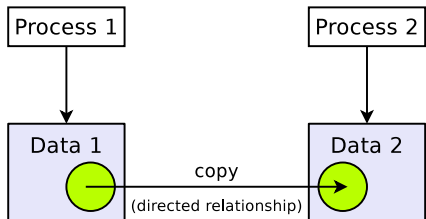


- The spooling directory is consistent
 - But the entry of process Y was overwritten by process X and got lost
- Such a situation is called race condition

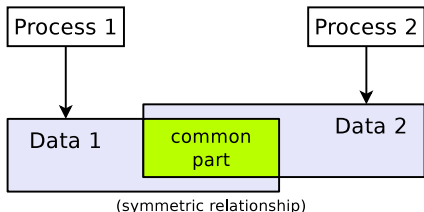
Communication vs. Cooperation

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

Communication
(= explicit data transport)

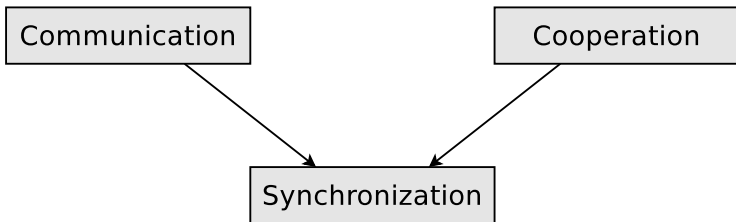


Cooperation
(= access to common data)



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 interacting partners to obtain correct results
 - Therefore, we first discuss the **synchronization**



Agenda

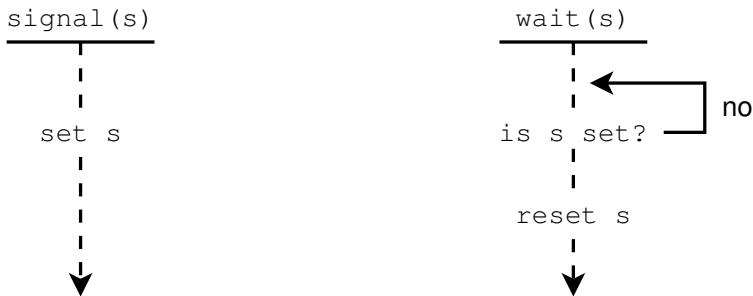
- Process Interaction
- **Process Synchronization**
- Inter-Processes Communication (IPC)
- Process Cooperation

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



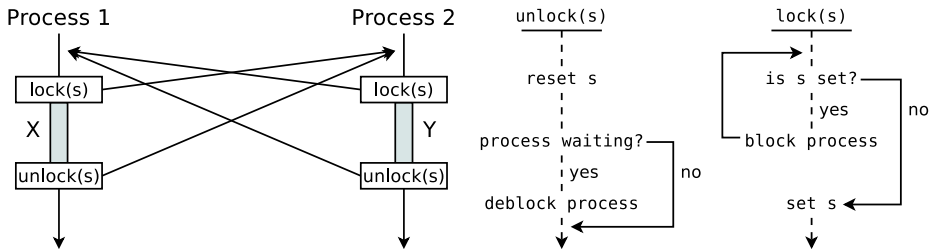
Most Simple Form of Signaling (Busy Waiting)



- 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**

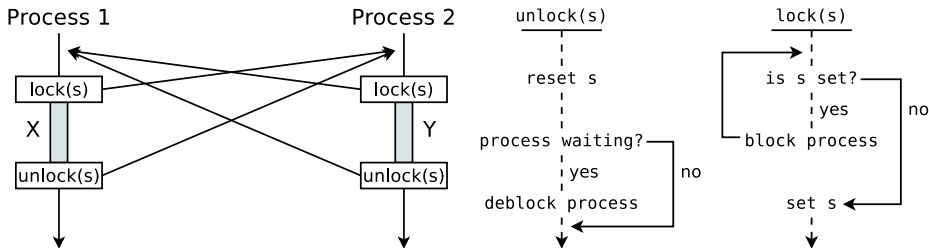
Securing critical Sections by Locking

- **Signaling** always specifies the **execution order**
 - But if it is just necessary to ensure that there is **no overlap** in the execution of the critical sections, it is possible to use the two operations **lock** and **unlock**



- **Locking** prevents the overlapping execution of two critical sections
 - **Example:** Critical Sections **X** of process P_A and **Y** of process P_B

Locking and Unlocking Processes in Linux (1/2)



Useful system calls and standard library function to call the operations lock and unlock in Linux

sigsuspend, kill, pause and sleep

- **Alternative 1:** Implementation of locking with the signals SIGSTOP (No. 19) and SIGCONT (No. 18)
 - With SIGSTOP another process can be stopped
 - With SIGCONT another process can be reactivated

Locking and Unlocking Processes in Linux (2/2)

- **Alternative 2:** A local file serves as a locking mechanism for mutual exclusion
 - Each process verifies before entering its critical section whether it can open the file exclusively
 - e.g., with the system call `open` or the standard library function `fopen`
 - If this is not the case, it must pause for a certain time (e.g., with the system call `sleep`) and then try again (**busy waiting**).
 - Alternatively, it can pause itself with `sleep` or `pause` and hope that the process that has already opened the file unblocks it with a signal at the end of its critical section (**passive waiting**)

Summary: 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

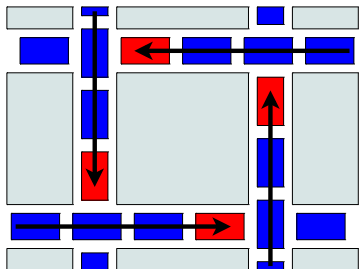
Problems caused by Locking

■ 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

System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. *Computing Surveys*, Vol. 3, No. 2, June 1971, P.67-78

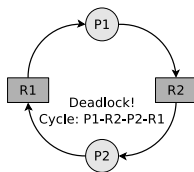
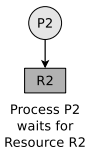
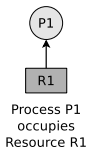
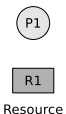
http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

- A deadlock situation can arise if these conditions are all fulfilled
 - **Mutual exclusion**
 - At least one resource is either occupied by exactly one process or is available \implies 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.
- If one of these conditions is not fulfilled, no deadlock can occur

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

Process



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

Deadlock Detection with Matrices

- One drawback of deadlock detection with resource graphs is that only individual resources can be represented with it
 - If multiple copies (instances) of a resource exist, then graphs are not suited for the visualisation and detection of deadlocks
 - If multiple copies of a resource exist, a matrices-based algorithm can be used, which requires two vectors and two matrices
- 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 with Matrices – Example (1/2)

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

Existing resource vector = (4 2 3 1)

- Four resources of class 1 exist
- Two resources of class 2 exist
- Three resources of class 3 exist
- One resource of class 4 exist

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

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

Available resource vector = (2 1 0 0)

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

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 execution, it deallocates its resources

Available resource vector = (2 2 2 0)

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

- 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
- If process 2 finished execution, it deallocates its resources
- Process 1 is blocked, because no free resources of class 4 exist
- Process 2 is not blocked

Available resource vector = (4 2 2 1)

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

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

Conclusion about Deadlocks

- Deadlock detection is complicated and causes overhead
- In all operating systems, deadlocks can occur:
 - Full process table
 - No more 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 0
 - Such potential deadlocks are 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)

Sometimes it is tolerated that deadlocks can occur

A deadlock which statistically occurs every five years is not a problem in a system which crashes because of hardware failures or other software problems one time per week

Agenda

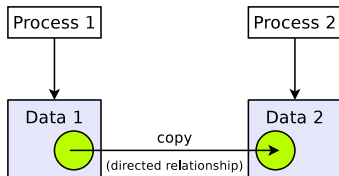
- Process Interaction
- Process Synchronization
- Inter-Processes Communication (IPC)
- Process Cooperation

Communication of Processes

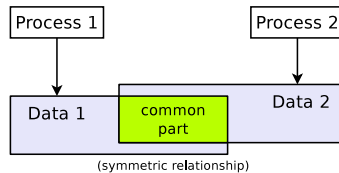
■ Types of IPC

- Files
- Signals/Flags
- Shared Memory
- Message Queues
- Pipes
- Sockets

Communication
(= explicit data transport)



Cooperation
(= access to common data)



Files

- An 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 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

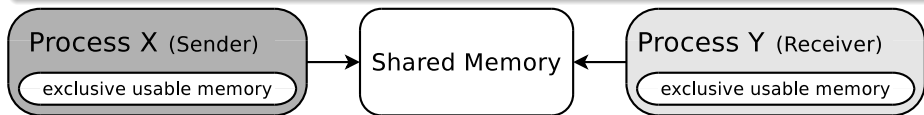
- Notify another process about the occurrence of an event
- **Linux**
 - **POSIX** signals
 - Standardized messages to trigger a certain behaviour
 - The receiver process gets interrupted
 - If a signal is unhandled by the receiver, it will terminate
- **RIOT**
 - Thread flags
 - Optional kernel feature
 - 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
- The processes need to coordinate the access operations by themselves and ensure that their memory requests are mutually exclusive
 - A receiver process, cannot read data from the shared memory, before the sender process has finished its current write operation
 - If access operations are not coordinated carefully \implies inconsistencies

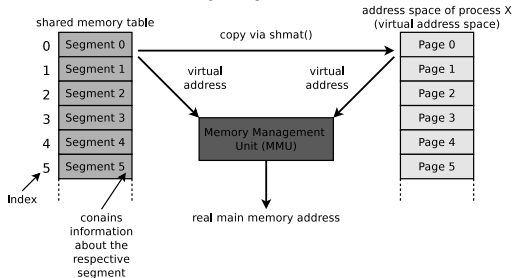
RIOT

Since most microcontrollers do not provide a MMU (memory management unit) 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 in a similar manner as files



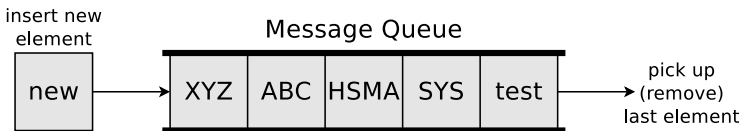
- A shared memory segment is always addressed via its index number in the shared memory table

- **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 pick 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

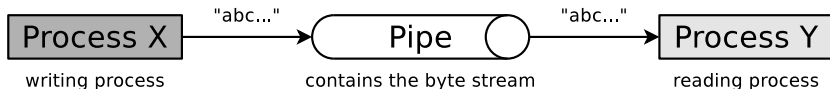
- **POSIX** and System V message queues
- Queues are named and can be shared via this name between processes
- Message have priorities

■ RIOT

- Kernel messages and mailboxes
- Optional feature
- Block and non-block API available
- A thread may create a message buffer
- Mailboxes can be accessed by multiple processes

Anonymous Pipes (1/2)

- Pipes can be **anonymous pipes** or **named pipes** (see slide 33)
- An **anonymous pipe**. . .
 - A buffered unidirectional communication channel between two processes (⇒ simplex FIFO)
 - One process accesses the write end, the other the read end of the pipe
 - ⇒ If communication in both directions shall be possible at the same time two pipes are necessary – one for each communication direction
 - has a limited capacity and can block on both ends:
 - If the pipe is filled ⇒ the writing process gets blocked
 - If the Pipe is empty ⇒ the reading process gets blocked



Anonymous Pipes (2/2)

- In Linux pipes are created with the system call `pipe()`
 - The kernel creates an \rightarrow *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: `lsuf` | `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

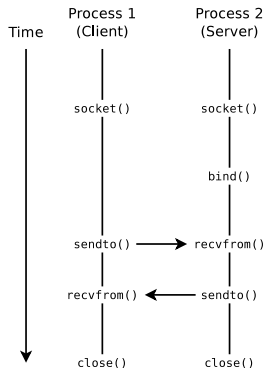
Using Sockets

- 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 eines 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 Communication via Sockets – UDP



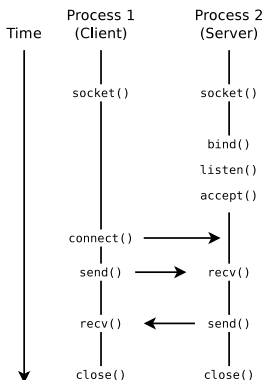
■ Client

- Create socket (`socket`)
- Send (`sendto`) and receive data (`rcvfrom`)
- Close socket (`close`)

■ Server

- Create socket (`socket`)
- Bind socket to a port (`bind`)
- Send (`sendto`) and receive data (`rcvfrom`)
- Close socket (`close`)

Connection-oriented Communication via 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

	Shared Memory	Message Queues	(anon./named) Pipes	Sockets
Scheme	Memory-based	Message-based	Message-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
⇒ 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

Agenda

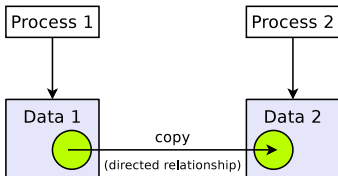
- Process Interaction
- Process Synchronization
- Inter-Processes Communication (IPC)
- **Process Cooperation**

Cooperation

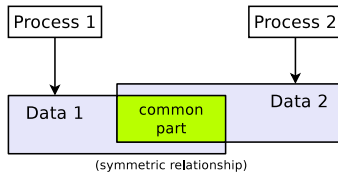
■ Cooperation

- Semaphore
- Mutex

Communication
(= explicit data transport)



Cooperation
(= access to common data)



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)
- The **access operations are atomic** \implies can not be interrupted (indivisible)
- May allow multiple processes accessing the critical section
 - In contrast, locks (\implies slide 13) can only be used to allow a single process entering the critical section at the same time

Cooperating sequential processes. Edsger W. Dijkstra (1965)

<https://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF>

Semaphore Access Operations (1/3)

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

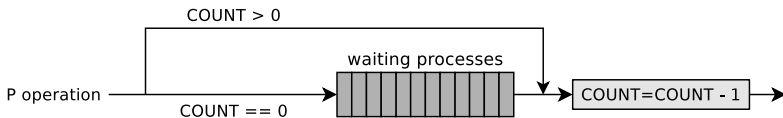
```
1 // apply the INIT operation on semaphore SEM
2 SEM.INIT(unsigned int init_value) {
3     // initialize the variable COUNT of Semaphor SEM
4     // with a non-negative initial value
5     SEM.COUNT = init_value;
6 }
```

Semaphore Access Operations (2/3)

Image Source: Carsten Vogt

- **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

```
1 SEM.P() {  
2   // if the counter variable = 0, the process becomes blocked  
3   if (SEM.COUNT == 0)  
4     < block >  
5   // if the counter variable is > 0, the counter variable  
6   // is decremented immediately by 1  
7   SEM.COUNT = SEM.COUNT - 1;  
8 }
```

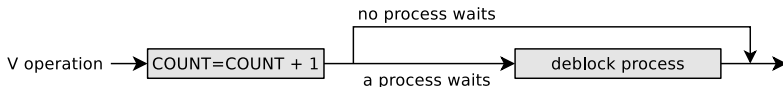


Semaphore Access Operations (3/3)

Image Source: Carsten Vogt

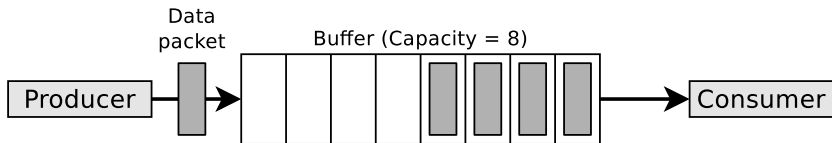
- **V operation** (*raise*): It first increases the counter variable by value 1
 - If processes are in the waiting room, one process gets deblocked
 - The process, which just got deblocked, continues its P operation and first reduces the counter variable

```
1 SEM.V() {  
2   // counter variable = counter variable + 1  
3   SEM.COUNT = SEM.COUNT + 1;  
4   // if processes are in the waiting room, one gets deblocked  
5   if ( < SEM waiting room is not empty > )  
6     < deblock a waiting process >  
7 }
```



Producer/Consumer Example (1/3)

- A **producer** sends data to a **consumer**
- A **buffer** with limited **capacity** is used to minimize the **waiting times** of the consumer
- Data is placed into the buffer by the producer and the consumer removes data from the buffer
- Mutual exclusion is mandatory in order to avoid inconsistencies

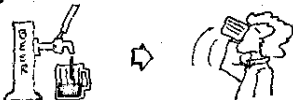


- If the buffer is full \implies producer must be blocked
- If the buffer is empty \implies consumer must be blocked

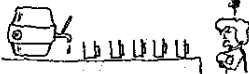
A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vignau

① PRODUCER CONSUMER

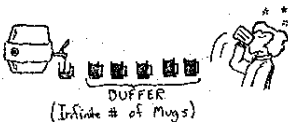


④ One-way hallway

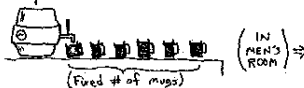


The consumer must wait for producer to produce before it can consume...

②



⑤ BOUNDED BUFFER



③ PROBLEM-



Consumer takes from buffer before producer is done adding to it - trouble!
This is solved by _____

If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now _____ (fill in)

(fill in the blank)

Source: Kenneth Baclawski (Northeastern University in Boston), Image source: Michael Vignau (license: unknown)

<http://www.ccs.neu.edu/home/kenb/tutorial/example.gif>

Producer/Consumer Example (2/3)

- Three semaphores are used to synchronize access to the buffer
 - empty
 - filled
 - mutex
- The semaphores filled and empty are used in opposite to each other
 - empty counts the number of empty locations in the buffer and its value is reduced by the producer (P operation) and raised by the consumer (V operation)
 - $\text{empty} = 0 \implies$ buffer is completely filled \implies producer is blocked
 - filled counts the number of data packets (occupied locations) in the buffer and its value is raised by the producer (V operation) and reduced by the consumer (P operation)
 - $\text{filled} = 0 \implies$ buffer is empty \implies consumer is blocked
- The semaphore mutex is used to ensure for the mutual exclusion

Binary Semaphores

- **Binary semaphores** are initialized with value 1 and ensure that 2 or more processes cannot simultaneously enter their critical sections
- Example: The semaphore mutex from the producer/consumer example

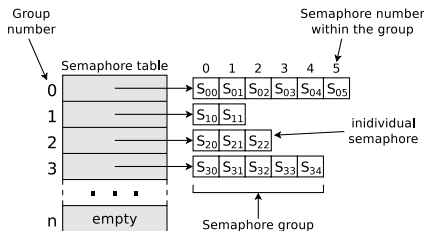
Producer/Consumer Example (3/3)

```
1 typedef int semaphore;           // semaphores are of type integer
2 semaphore filled = 0;           // counts the number of occupied locations in the buffer
3 semaphore empty = 8;           // counts the number of empty locations in the buffer
4 semaphore mutex = 1;           // controls access to the critical sections
5 void producer (void) {
6     int data;
7     while (TRUE) {              // infinite loop
8         createDatapacket(data); // create data packet
9         P(empty);               // decrement the empty locations counter
10        P(mutex);               // enter the critical section
11        insertDatapacket(data); // write data packet into the buffer
12        V(mutex);               // leave the critical section
13        V(filled);              // increment the occupied locations counter
14    }
15 }
16 void consumer (void) {
17     int data;
18     while (TRUE) {              // infinite loop
19         P(filled);              // decrement the occupied locations counter
20         P(mutex);               // enter the critical section
21         removeDatapacket(data); // pick data packet from the buffer
22         V(mutex);               // leave the critical section
23         V(empty);               // increment the empty locations counter
24         consumeDatapacket(data); // consume data packet
25    }
26 }
```


Semaphores in Linux (System V)

Image Source: Carsten Vogt

- 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, which means that they are indivisible
- 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



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 **Mutual Exclusion**) 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`
- **Focus: Cooperation of threads of a process (intra-process synchronization)**
 - Cooperation of processes (inter-process synchronization) is not always possible and if, then via a shared memory segment (System V or POSIX)

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`

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**)?

