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** and **race conditions** are
  - what **synchronization** is
    - how **signaling** influences the execution order of the processes
    - how critical sections can be secured via **blocking**
    - what problems (**starvation** and **deadlocks**) may arise from blocking
    - how **deadlock detection with matrices** works
  - different options to implement **communication** between processes:
    - **Shared memory**
    - **Message queues**
    - **Pipes**
    - **Sockets**
  - different options to implement **cooperation** between processes
    - how critical sections can be protected via **semaphores**
    - the difference between **semaphore** and **mutex**

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

Interprocess Communication (IPC)

- Processes do not only carry out read and write operations on data, but also:
  - call each other
  - wait for each other
  - coordinate with each other
  - In short: 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 is the situation here with threads?

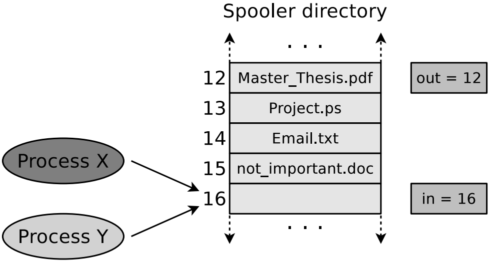
- For threads, the same challenges and solutions exist as for interprocess communication with processes
- Only the communication between the threads of a process is no problem because they operate in the same address space

Critical Sections

- If multiple processes 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
- In order for processes to be able to access a 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;` (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**

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 ($\Longrightarrow$ slide 65)

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

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

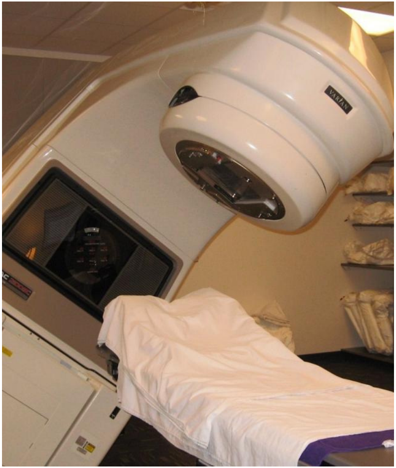


Image source: Google image search. Frequently shown picture in this context. (author and license: unknown)

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

- A race condition („Texas-Bug“) led to incorrect settings of the device and consequently to increased radiation doses.
  - The control process did not synchronize correctly with the user interface process
  - The error occurred only during a quick input correction (time window: 8 seconds) by the user
  - During testing the error did not occur because experience (routine) was required to operate the device this fast

The Worst Computer Bugs in History: Race conditions in Therac-25:
`https://www.bugsnag.com/blog/bug-day-race-condition-therac-25`

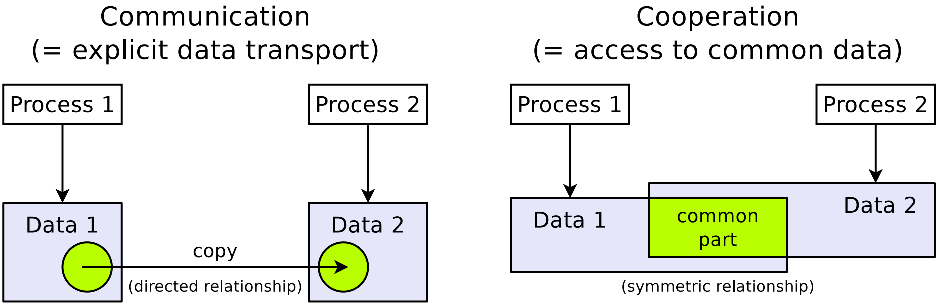
„Once the data entry phase was marked complete, the magnet setting phase began. However, if a specific sequence of edits was applied in the Data Entry phase during the 8 second magnet setting phase, the setting was not applied to the machine hardware, due to the value of the completion variable. The UI would then display the wrong mode to the user, who would confirm the potentially lethal treatment.“

Other interesting sources

https://www-dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/mehner_RisikoComputer_zs09.pdf
Killer Bug. Therac-25: Quick-and-Dirty: https://www.viva64.com/en/b/0438/
Killed by a machine: The Therac-25: https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

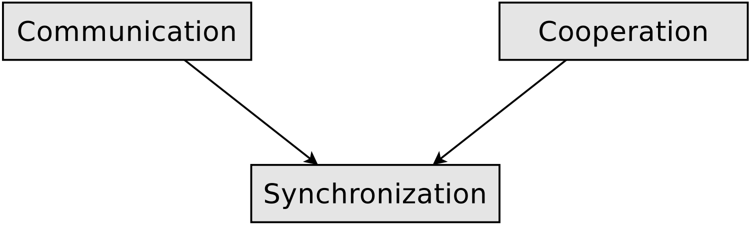
Communication vs. Cooperation

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



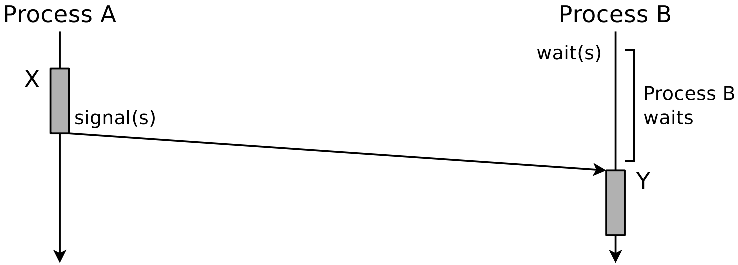
Forms of Interaction

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

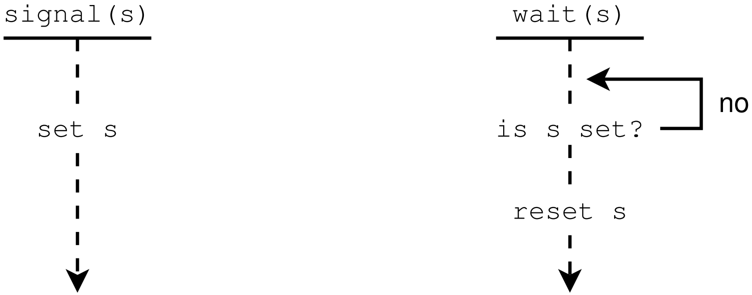


Signaling

- One way to synchronize processes
- 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



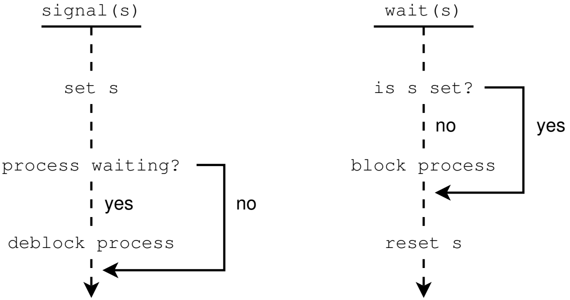
Most Simple Form of Signaling (Busy Waiting)



- 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**

Signal and Wait

- Better concept: Blocking of process P_B until process P_A has finished section **X**
  - Advantage: No CPU resources are wasted
  - Drawback: Only a single process can wait
  - In literature, this technique is also called **passive waiting**

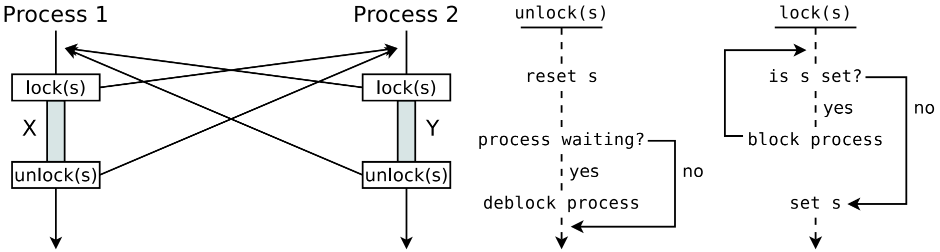


One way to specify in Linux an execution order with passive waiting, is by using the function `sigsuspend`. Thereby a process blocks itself until another process sends it an appropriate signal (usually `SIGUSR1` or `SIGUSR2`) with the command `kill` (or the system call of the same name) and in this way signals that it should continue working.

Alternative system calls and function calls by which a process can block itself until it is woken up again by a system call are `pause` and `sleep`

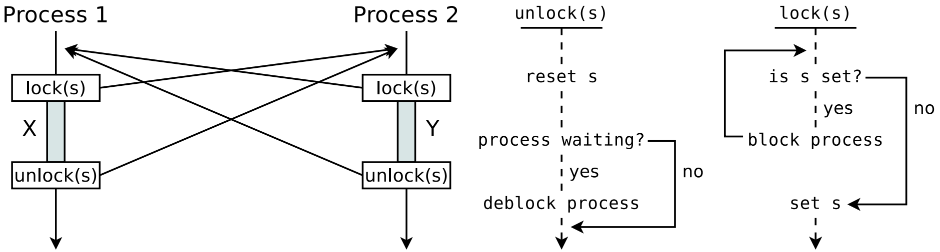
Securing critical Sections by Locking / Blocking

- 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`



- Blocking (locking) prevents the overlapping execution of 2 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)

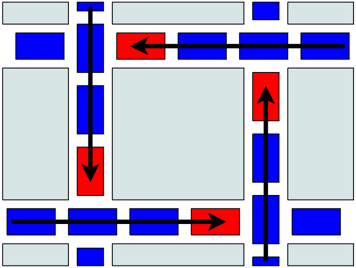
- Alternative 2: A local file serves as a locking mechanism for mutual exclusion
  - Each process verifies before entering its critical section whether it can open the file exclusively
    - e.g. with the system call `open` or the standard library function `fopen`
  - If this is not the case, it must 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 Blocking

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

- **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)

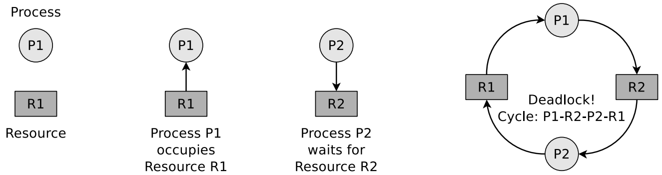
Conditions for Deadlock Occurrence

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 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 another process
  - **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 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



A good description of resource graphs provides the book **Betriebssysteme – Eine Einführung**, *Uwe Baumgarten, Hans-Jürgen Siegert*, 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 2 vectors and 2 matrices
- We specify 2 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 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 vector $= \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$

- 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 vector $= \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$

- 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 execution, it deallocates its resources

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

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 execution, it deallocates its resources

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

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

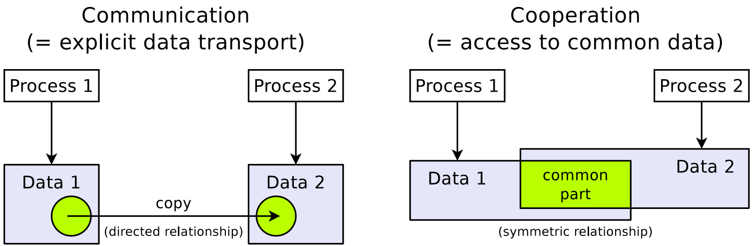
- **Process 1 is not blocked** $\Longrightarrow$ no deadlock in this example

Conclusion about Deadlocks

- Sometimes it is tolerated that deadlocks can occur
  - 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 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 $\neq 0$
  - 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)

Communication of Processes

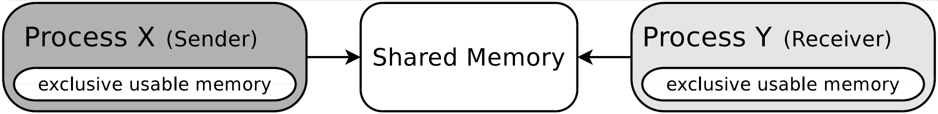
- Communication
  - Shared Memory
  - Message Queues
  - Pipes
  - Sockets



Shared Memory

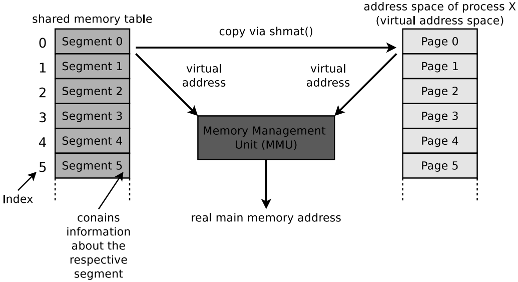
- Interprocess communication via a shared memory is also called **memory-based communication**
- **Shared memory segments** are memory areas, which can be accessed by multiple processes
  - These memory areas are located in the address space of multiple processes
- The processes need to coordinate the 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 $\Longrightarrow$ inconsistencies

In all other forms of interprocess communication, the operating system takes care about the synchronization of the access operations



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



- 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

Working with Shared Memory

Linux/UNIX operating systems provide 4 system calls for working with shared memory

- `shmget()`: Create a shared memory segment or access an existing one
- `shmat()`: Attach a shared memory segment to a process
- `shmdt()`: Detach a shared memory segment from a process
- `shmctl()`: Request status information (e.g. privileges) of a shared memory segment, modify or erase it

One example of working with shared memory segments in Linux can be found on the website of this course

`ipcs`

The command `ipcs` provides information about existing shared memory segments

Create a Shared Memory Segment (in C)

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;

    // Create shared memory segment or access an existing one
    // IPC_CREAT = create a shared memory segment, if it does not still exist
    // 0600 = Access privileges for the new message queue
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);

    if (returncode_shmget < 0) {
        printf("Unable to create the shared memory segment.\n");
        perror("shmget");
    } else {
        printf("The shared memory segment has been created.\n");
    }
}
```

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00003039 56393780   bnc        600        20         0

$ printf "%d\n" 0x00003039        # Convert from hexadecimal to decimal
12345
```

Attach a Shared Memory Segment (in C)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;
    char *sharedmempointer;

    // Create shared memory segment or access an existing one
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...

        // Attach shared memory segment
        sharedmempointer = shmat(returncode_shmget, 0, 0);
        if (sharedmempointer==(char *)-1) {
            printf("Unable to attach the shared memory segment.\n");
            perror("shmat");
        } else {
            printf("The shared memory segment has been attached %p\n", sharedmempointer);
        }
    }
}
```

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00003039 56393780   bnc        600        20         1
```

Write into a Shared Mem. Segment and read from it (in C)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget, returncode_shmdt, returncode_sprintf;
    char *sharedmempointer;

    // Create shared memory segment or access an existing one
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...
        // Attach shared memory segment
        sharedmempointer = shmat(returncode_shmget, 0, 0);
        ...

        // Write a string into the shared memory segment
        returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
        if (returncode_sprintf < 0) {
            printf("The write operation did fail.\n");
        } else {
            printf("%i chareacters written into the segment.\n", returncode_sprintf);
        }

        // Read the string from the shared memory segment
        if (printf ("%s\n", sharedmempointer) < 0) {
            printf("The read operation did fail.\n");
        }
    ...
```

Detach a Shared Memory Segment (in C)

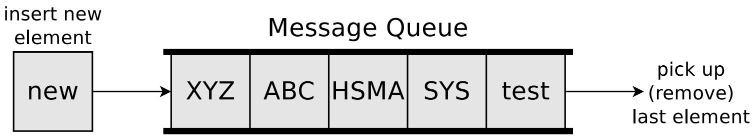
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;
    int returncode_shmdt;
    char *sharedmempointer;

    // Create shared memory segment or access an existing one
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...

        // Attach the shared memory segment
        sharedmempointer = shmat(returncode_shmget, 0, 0);
        ...

        // Detach the shared memory segment
        returncode_shmdt = shmdt(sharedmempointer);
        if (returncode_shmdt < 0) {
            printf("Unable to detach the shared memory segment.\n");
            perror("shmdt");
        } else {
            printf("The shared memory segment has been detached.\n");
        }
    }
}
```

Erase a Shared Memory Segment (in C)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;
    int returncode_shmctl;
    char *sharedmempointer;

    // Create shared memory segment or access an existing one
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...

        // Ease shared memory segment
        returncode_shmctl = shmctl(returncode_shmget, IPC_RMID, 0);
        if (returncode_shmctl == -1) {
            printf("Unable to erase the shared memory segment.\n");
            perror("semctl");
        } else {
            printf("The shared memory segment has been erased.\n");
        }
    }
}
```

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



Linux/UNIX operating systems provide 4 system calls for working with message queues

- `msgget()`: Create a message queue or access an existing one
- `msgsnd()`: Write messages into message queues ($\Longrightarrow$ send operation)
- `msgrcv()`: Read messages from message queues ($\Longrightarrow$ receive operation)
- `msgctl()`: Request status information (e.g. privileges) of a message queue, modify or erase it

The command `ipcs` provides information about existing message queues

Create Message Queues (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>

int main(int argc, char **argv) {
    int returncode_msgget;

    // Create message queue or access an existing one
    // IPC_CREAT => create a message queue, if it does not still exist
    // 0600 = Access privileges for the new message queue
    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
    if(returncode_msgget < 0) {
        printf("Unable to create the message queue.\n");
        exit(1);
    } else {
        printf("The message queue 12345 with the ID %i has been created.\n",
             returncode_msgget);
    }
}
```

```
$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 98304      bnc        600        0            0

$ printf "%d\n" 0x00003039        # Convert from hexadecimal to decimal
12345
```

Write Messages into Message Queues (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>                         // This header file is required for strcpy()

struct msgbuf {                             // Template of a buffer for msgsnd and msgrcv
    long mtype;                             // Message type
    char mtext[80];                         // Send buffer
} msg;

int main(int argc, char **argv) {
    int returncode_msgget;

    // Create message queue or access an existing one
    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
    ...

    msg.mtype = 1;                          // Specifiy the message type
    strcpy(msg.mtext, "Testnachricht");     // Write the message into the send buffer

    // Write a message into the message queue
    if (msgsnd(returncode_msgget, &msg, strlen(msg.mtext), 0) == -1) {
        printf("Unable to write the message into the message queue.\n");
        exit(1);
    }
}
```

- The message type (a positive integer value) specifies the user

Result of writing a Message into a Message Queue

- Before. . .

```
$ ipcs -q
------ Message Queues --------
key        msqid     owner     perms     used-bytes   messages
0x00003039 98304     bnc       600       0            0
```

- Afterwards. . .

```
$ ipcs -q
------ Message Queues --------
key        msqid     owner     perms     used-bytes   messages
0x00003039 98304     bnc       600       80           1
```

Pick a Message from a Message Queue (in C)

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>                   // This header file is required for strcpy()
struct msgbuf {                       // Template of a buffer for msgsnd and msgrcv
    long mtype;                       // Message type
    char mtext[80];                   // Send buffer
} msg;

int main(int argc, char **argv) {
    int returncode_msgget, returncode_msgrcv;
    msg receivebuffer;                // Create a receive buffer

    // Create message queue or access an existing one
    returncode_msgget = msgget(12345, IPC_CREAT | 0600)

    msg.mtype = 1;                    // Pick the first message of type 1
    // MSG_NOERROR => The message will be truncated when it is too long
    // IPC_NOWAIT  => Do not bock the process if no message exists
    returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
         MSG_NOERROR | IPC_NOWAIT);
    if (returncode_msgrcv < 0) {
        printf("Unable to pick a message from the message queue.\n");
        perror("msgrcv");
    } else {
        printf("This message was picked from the message queue: %s\n", msg.mtext);
        printf("The received message is %i characters long.\n", returncode_msgrcv);
    }
}
```

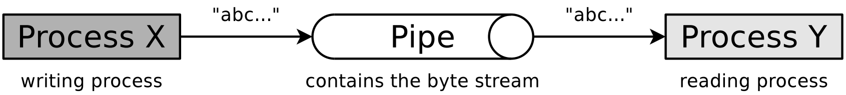
Erase a Message Queue (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>

int main(int argc, char **argv) {
    int returncode_msgget;
    int returncode_msgctl;

    // Create message queue or access an existing one
    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
    ...

    // Erase message queue
    returncode_msgctl = msgctl(returncode_msgget, IPC_RMID, 0);
    if (returncode_msgctl < 0) {
        printf("Unable to erase the message queue with the ID %i.\n", returncode_msgget);
        perror("msgctl");
        exit(1);
    } else {
        printf("The message queue with the ID %i has been erased.\n", returncode_msgget);
    }
    exit(0);
}
```

One example of working with message queues in Linux can be found on the website of this course

Pipes (1/2)

- An **anonymous Pipe**. . .
  - is a buffered unidirectional communication channel between 2 processes
    - If communication in both directions shall be possible at the same time, 2 pipes are necessary – one for each communication direction
  - operates according to the FIFO principle
  - has a limited capacity
    - Pipe = filled $\Longrightarrow$ the writing process gets blocked
    - Pipe = empty $\Longrightarrow$ the reading process gets blocked
  - is created with the system call `pipe()`
    - During this process, the kernel of the operating system creates an Inode ($\Longrightarrow$ slide set 6) and 2 file descriptors (*handles*)
    - Processes access the access identifiers with `read()` and `write()` system calls (or standard library functions) for reading data from or writing data into the pipe



Pipes (2/2)

- 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
- 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

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

An Anonymous Pipe Example (in C) – Part 1/2

One example of working with named pipes in Linux can be found on the website of this course

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void main() {
  int pid_des_Kindes;
  // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
  int testpipe[2];

  // Die Pipe testpipe anlegen
  if (pipe(testpipe) < 0) {
    printf("Das Anlegen der Pipe ist fehlgeschlagen.\n");
    // Programmabbruch
    exit(1);
  } else {
    printf("Die Pipe testpipe wurde angelegt.\n");
  }

  // Einen Kindprozess erzeugen
  pid_des_Kindes = fork();

  // Es kam beim fork zu einem Fehler
  if (pid_des_Kindes < 0) {
    perror("Es kam bei fork zu einem Fehler!\n");
    // Programmabbruch
    exit(1);
  }
```

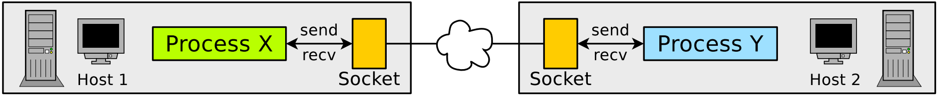
An Anonymous Pipe Example (in C) – Part 2/2

```c
  // Elternprozess
  if (pid_des_Kindes > 0) {
    printf("Elternprozess: PID: %i\n", getpid());
    // Lesekanal der Pipe testpipe blockieren
    close(testpipe[0]);
    char nachricht[] = "Testnachricht";
    // Daten in den Schreibkanal der Pipe schreiben
    write(testpipe[1], &nachricht, sizeof(nachricht));
  }

  // Kindprozess
  if (pid_des_Kindes == 0) {
    printf("Kindprozess: PID: %i\n", getpid());
    // Schreibkanal der Pipe testpipe blockieren
    close(testpipe[1]);
    // Einen Empfangspuffer mit 80 Zeichen Kapazität anlegen
    char puffer[80];
    // Daten aus dem Lesekanal der Pipe auslesen
    read(testpipe[0], puffer, sizeof(puffer));
    // Empfangene Daten ausgeben
    printf("Empfangene Daten: %s\n", puffer);
  }
}
```

```
$ gcc pipe_beispiel.c -o pipe_beispiel
$ ./pipe_beispiel
Die Pipe testpipe wurde angelegt.
Elternprozess: PID: 6363
Kindprozess: PID: 6364
Empfangene Daten: Testnachricht
```

Sockets

- Full duplex-ready alternative to pipes and shared memory
  - Allow interprocess communication in distributed systems
- An user process can request a socket from the operating system and afterwards send and receive data via the socket
  - The operating system maintains all used sockets and the related connection information



- Ports are used for the communication via sockets
  - Port numbers are randomly assigned during connection establishment
  - Port numbers are assigned randomly by the operating system
    - Exceptions are port numbers of well-known applications, such as HTTP (80) SMTP (25), Telnet (23), SSH (22), FTP (21),...
- Sockets can be used in a blocking (synchronous) and non-blocking (asynchronous) way

Different Types of Sockets

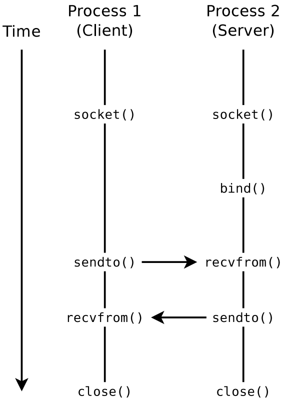
- **Connectionless sockets** (= **datagram sockets**)
  - Use the Transport Layer protocol UDP
  - Advantage: Better data rate as with TCP
    - Reason: Lesser overhead for the protocol
  - Drawback: Segments may arrive in wrong sequence or may get lost
- **Connection-oriented sockets** (= **stream sockets**)
  - 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()`

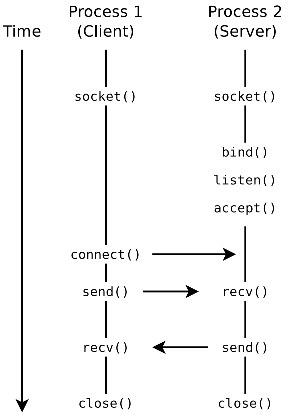
Overview of the sockets in Linux/UNIX: `netstat -n` or `lsof | grep socket`

Connection-less Communication via 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 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 connections with clients
  - Server accepts connections (`accept`)
  - Send (`send`) and receive data (`recv`)
  - Close socket (`close`)

Create a Socket: socket

```
int socket(int domain, int type, int protocol);
```

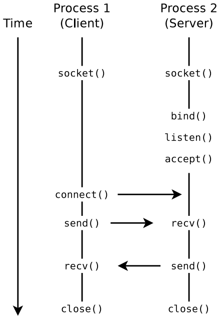
- A call of `socket()` returns an integer value
  - The value is called **socket descriptor** (*socket file descriptor*)
- `domain`: Specifies the protocol family
  - `PF_UNIX`: Local interprocess communication in Linux/UNIX
  - `PF_INET`: IPv4
  - `PF_INET6`: IPv6
- `type`: Specifies the type of the socket (and thus the protocol):
  - `SOCK_STREAM`: Stream socket (TCP)
  - `SOCK_DGRAM`: Datagram socket (UDP)
  - `SOCK_RAW`: RAW socket (IP)
- In most cases the `protocol` parameter is set to value zero
- Create a socket with `socket()`:

```c
sd = socket(PF_INET, SOCK_STREAM, 0);
    if (sd < 0) {
        perror("The socket could not be created");
        return 1;
    }
```

Bind Address and Port Number: bind

```
int bind(int sd, struct sockaddr *address, int addrlen);
```

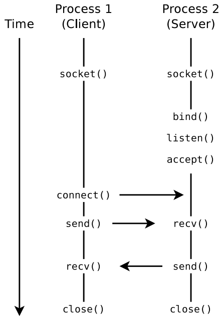
- `bind()` binds the newly created socket (`sd`) to the address (`address`) of the server
  - `sd` is the socket descriptor from the previous call of `socket()`
  - `address` is a data structure, which contains the IP address of the server and a port number
  - `addrlen` is the length of the data structure, which contains the IP address and port number



Make a Server ready to receive Data: listen

```
int listen(int sd, int backlog);
```

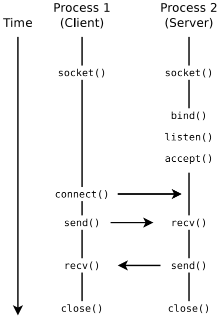
- `listen()` specifies how many connection requests can be buffered by the socket
  - If the `listen()` queue has no more free capacity, further connection requests from clients are rejected
  - `sd` is the socket descriptor from the previous call of `socket()`
  - `backlog` contains the number of possible connection requests, which can be stored in the queue
    - Default value: 5
  - A server for datagrams (UDP) does not need to call `listen()`, because it does not establish connections to clients



Accept a Connection Request: accept

```
int accept(int sd, struct sockaddr *address, int *addrlen);
```

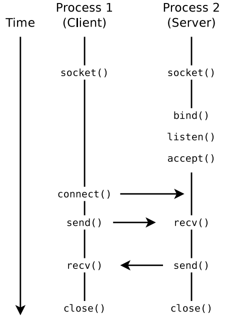
- `accept()` is used by the server to fetch the first connection request from the queue
- The return value is the socket descriptor of the new socket
- If the queue contains no connection requests, the process is blocked until a connection request arrives
- `address` contains the address of the client
- After a connection request was accepted with `accept()`, the connection with the client is established



Establish a Connection by the Client

```
int connect(int sd, struct sockaddr *servaddr,
            socklen_t addrlen);
```

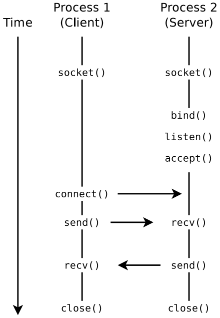
- Via `connect()`, the client tries to establish a connection to a server socket
- The client must know the address (hostname and port number) of the server
- `sd` is the socket descriptor
- `address` contains the address of the server
- `addrlen` is the length of the data structure, which contains the address of the server



Connection-oriented Exchange of Data: send and recv

```
int send(int sd, char *buffer, int nbytes, int flags);
int recv(int sd, char *buffer, int nbytes, int flags);
```

- Data are exchanged via `send()` and `recv()` over an existing connection
- `send()` sends a message (`buffer`) via the socket (`sd`)
- `recv()` receives a message from the socket `sd` and stores it in the buffer (`buffer`)
- `sd` is the socket descriptor
- `buffer` contains the data to be sent or received
- `nbytes` specifies the number of bytes in the buffer
- The value of `flags` is usually zero



Connection-oriented Exchange of Data: read and write

```
int read(int sd, char *buffer, int nbytes);
int write(int sd, char *buffer, int nbytes);
```

- In UNIX it is in normal case also possible to use `read()` and `write()` for receiving and sending data via a socket
  - „Normal case“ means, that `read()` and `write()` can be used, when the parameter `flags` of `send()` and `recv()` contains value zero
- The following calls have the same result

```
send(socket,"Hello World",11,0);
write(socket,"Hello World",11);
```

Connection-less Exchange of Data: `sendto` and `recvfrom`

```
int sendto(int sd, char *buffer, int nbytes, int flags,
           struct sockaddr *to, int addrlen);
int recvfrom(int sd, char *buffer, int nbytes, int flags,
             struct sockaddr *from, int addrlen);
```

- If a process knows the address of the socket (host and port), to which it should send data, it uses `sendto()`
- `sendto()` always transmits together with the data the local address
- `sd` is the socket descriptor
- `buffer` contains the data to be sent or received
- `nbytes` specifies the number of bytes in the buffer
- `to` contains the address of the receiver
- `from` contains the address of the sender
- `addrlen` is the length of the data structure, which contains the address

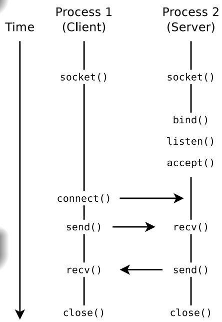
Close a Socket: close

```
int shutdown(int sd, int how);
```

- `shutdown()` closes a bidirectional socket connection
- The parameter `how` specifies whether no more data will be received (`how=0`), no more data will be send (`how=1`), or both (`how=2`)

```
int close(int sd);
```

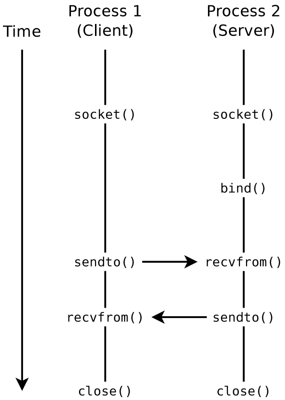
- If `close()` is used instead of `shutdown()`, this corresponds to a `shutdown(sd,2)`



Sockets via UDP – Example (Server)

```
#!/usr/bin/env python
# Server: Receives a message via UDP

import socket                # Import module socket

# For all interfaces of the host
HOST = ''                    # '' = all interfaces
PORT = 50000                 # Port number of server

# Create socket and return socket deskriptor
sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

try:
  sd.bind((HOST, PORT))        # Bind socket to port
  while True:
    # Receive data
    data = sd.recvfrom(1024)
    # Print received data
    print 'Received:', repr(data)
finally:
  sd.close()                 # Close socket
```

```
$ python udp_server.py
```

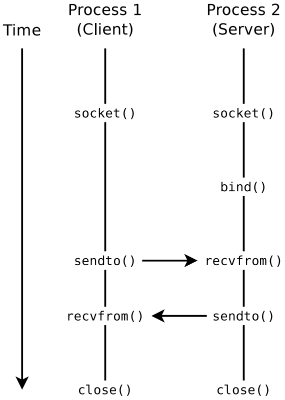


Sockets via UDP – Example (Client)

```
#!/usr/bin/env python
# Client: Sends a message via UDP

import socket                  # Import module socket

HOST = 'localhost'             # Hostname of Server
PORT = 50000                   # Port number of Server
MESSAGE = 'Hello World'        # Message

# Create socket and return socket deskriptor
sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Send message to socket
sd.sendto(MESSAGE, (HOST, PORT))

sd.close()                     # Close socket
```

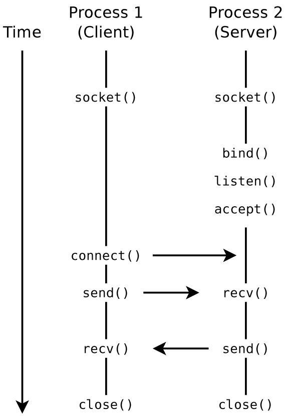
```
$ python udp_client.py
```

```
$ python udp_server.py
Received: ('Hello World', ('127.0.0.1', 39834))
```



Sockets via TCP – Example (Server)

```python
#!/usr/bin/env python
# Echo Server via TCP
import socket                   # Import module socket

HOST = ''                       # '' = all interfaces
PORT = 50007                    # Port number of server

# Create socket and return socket deskriptor
sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Bind socket to port
sd.bind((HOST, PORT))
# Make socket ready to receive
# Max. number of connections = 1
sd.listen(1)
# Socket accepts connections
conn, addr = sd.accept()

print 'Connected by', addr

while 1:                        # Infinite loop
    data = conn.recv(1024)      # Receive data
    if not data: break          # Break infinite loop
    conn.send(data)             # Send back received data

sd.close()                      # Close socket
```



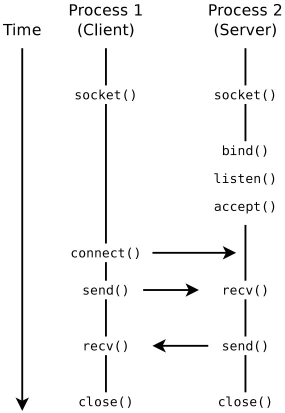

```
$ python tcp_server.py
```

Sockets via TCP – Example (Client)

```python
#!/usr/bin/env python
# Echo Client via TDP

import socket                     # Import module socket

HOST = 'localhost'                # Hostname of Server
PORT = 50007                      # Port number of server

# Create socket and return socket deskriptor
sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Connect with server socket
sd.connect((HOST, PORT))

sd.send('Hello, world')     # Send data
data = sd.recv(1024)        # Receive data
sd.close()                  # Close socket

# Print received data
print 'Empfangen:', repr(data)
```

```
$ python tcp_client.py
Empfangen: 'Hello, world'
```

```
$ python tcp_server.py
Connected by ('127.0.0.1', 49898)
```



Blocking and non-blocking Sockets

- If a socket is created, it per default in **blocking mode**
  - All method calls wait until the operation, they initiated, was carried out
    - e.g. a call of `recv()` blocks the process until data is received and can be read from the internal buffer of the socket
- The method `setblocking()` **modifies** the mode of a socket
  - `sd.setblocking(0)` $\Longrightarrow$ switches into non-blocking mode
  - `sd.setblocking(1)` $\Longrightarrow$ switches into blocking mode
- It is possible to switch between the modes **at any time** during process execution
  - e.g. the method `connect()` could be used in blocking mode and afterwards the method `read()` in non-blocking mode

Source: Peter Kaiser, Johannes Ernesti. Python – Das umfassende Handbuch. Galileo (2008)

Non-blocking Sockets - some Impacts

- `recv()` and `recvfrom()`
  - The method return data only, when they are already stored in the buffer
  - If the buffer does not contain any data, the method throws an **exception** and the program execution **continues**
- `send()` and `sendto()`
  - The methods send the specified data only, when they can be written directly in the send buffer
  - If the buffer has no more free capacity, the method throws an **exception** and the program execution **continues**
- `connect()`
  - The method sends a connection request to the destination socket and **does not wait** until this connection is established
  - If `connect()` is called, while the connection request is still in progress, an **exception** is thrown
    - By calling `connect()` several times, it can be checked, whether the operation is still carried out

Comparison of Communication Systems

	Shared Memory	Message Queues	(anon./named) Pipes	Sockets
Sort of communication	Memory-based	Message-based	Message-based	Message-based
Bidirectional	yes	no	no	yes
Platform independent	no	no	no	yes
Processes must be related with each other	no	no	for anon. pipes	no
Communication over computer boundaries	no	no	no	yes
Remain intact without a bound process	yes	yes	no	no
Automatic synchronization	no	yes	yes	yes

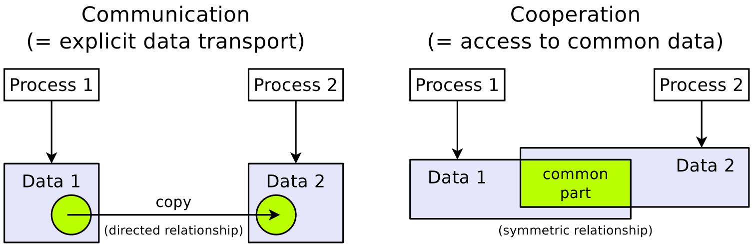
- Advantages of message-based communication versus memory-based communication:
  - The operating system takes care about the synchronization of accesses $\Longrightarrow$ comfortable
  - Can be used in distributed systems without a shared memory
  - Better portability of applications

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

Cooperation

- Cooperation
  - Semaphor
  - Mutex



Semaphore

- In order to protect (lock) critical sections, not only the already discussed locks can be used, but also **semaphores**
- 1965: Published 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** $\Longrightarrow$ can not be interrupted (indivisible)
- May allow multiple processes accessing the critical section
  - In contrast to semaphores, can locks ($\Longrightarrow$ slide 14) 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

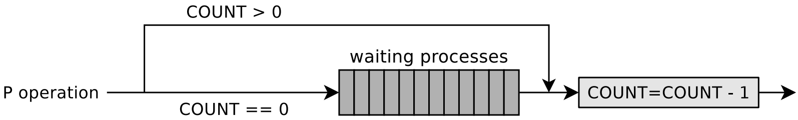
```
// apply the INIT operation on semaphore SEM
SEM.INIT(unsigned int init_value) {

    // initialize the variable COUNT of Semaphor SEM
    // with a non-negative initial value
    SEM.COUNT = init_value;
}
```

Semaphore Access Operations (2/3)

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

```
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;
}
```

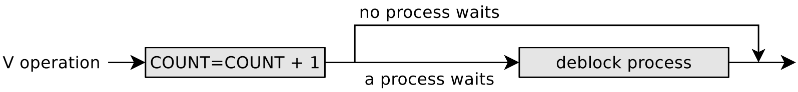


Semaphore Access Operations (3/3)

Image Source: Carsten Vogt

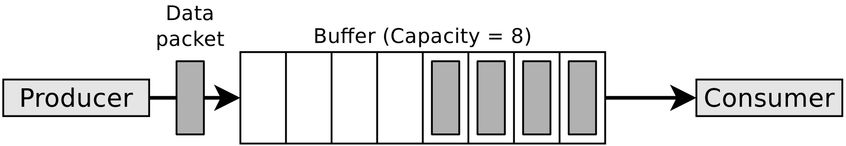
- **V operation** (*raise*): It first increases the counter variable by value 1
  - If processes are in the waiting room, one process gets deblocked
  - The process, which just got deblocked, continues its P operation and first reduces the counter variable

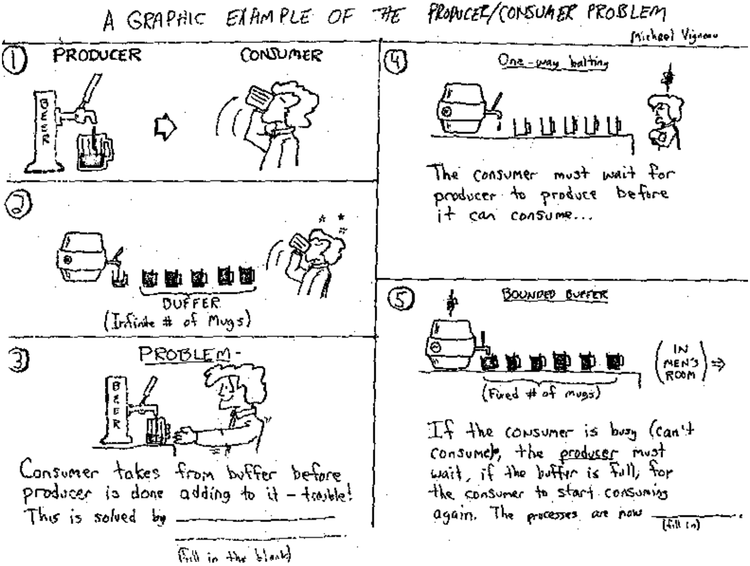
```
SEM.V() {
    // counter variable = counter variable + 1
    SEM.COUNT = SEM.COUNT + 1;

    // if processes are in the waiting room, one gets deblocked
    if ( < SEM waiting room is not empty > )
        < deblock a waiting process >
}
```



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
- Buffer = full $\Longrightarrow$ producer must be blocked
- Buffer = empty $\Longrightarrow$ consumer must be blocked





Source: Kenneth Baclawski (Northeastern University in Boston), Image source: Michael Vigneau (license: unknown)
`http://www.ccs.neu.edu/home/kenb/tutorial/example.gif`

Producer/Consumer Example (2/3)

- 3 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)
    - `empty` $= 0 \Longrightarrow$ puffer is completely filled $\Longrightarrow$ 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)
    - `filled` $= 0 \Longrightarrow$ puffer is empty $\Longrightarrow$ 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

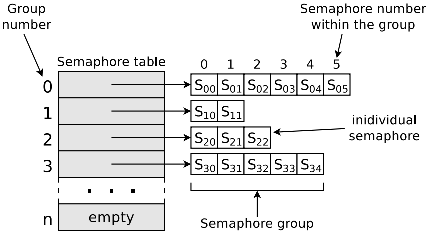
Producer/Consumer Example (3/3)

```
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

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 3 system calls for working with 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 provides the command `ipcs`

Mutexes

- Semaphores offer the feature of counting
- However, if this feature is not required, a simplified semaphore version, 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 2 states: **occupied** and **not occupied**
        - Mutexes have the same functionality as **binary semaphores**

2 functions for accessing a Mutex exist

`mutex_lock`	$\Longrightarrow$	corresponds to the P operation
`mutex_unlock`	$\Longrightarrow$	corresponds to the V operation

- If a process wants to access a critical section, it calls `mutex_lock`
    - If the critical section is **locked**, the process gets locked, until the process in the critical section is finished and calls `mutex_unlock`
    - If the critical section is **not locked**, the process can enter it

Monitor and erase IPC Objects

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

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

- Or alternatively just...
  - `ipcrm shm SharedMemoryID`
  - `ipcrm sem SemaphorID`
  - `ipcrm msg MessageQueueID`