

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 (⇒ 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.“

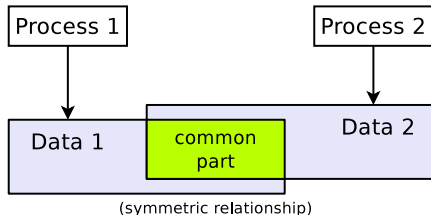
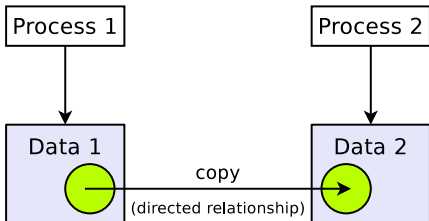
Other interesting sources

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/>

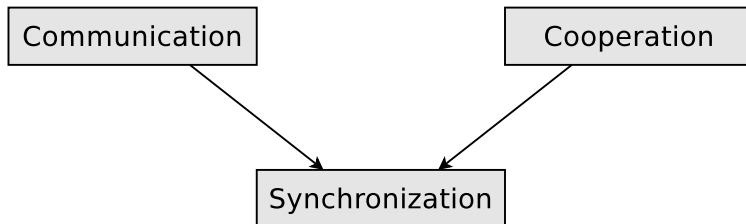
Killed by a machine: The Therac-25: <https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/>

1000

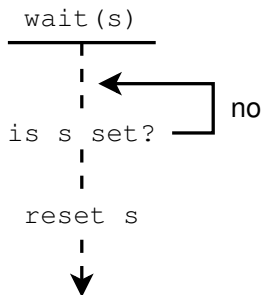


Forms of Interaction

- 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**



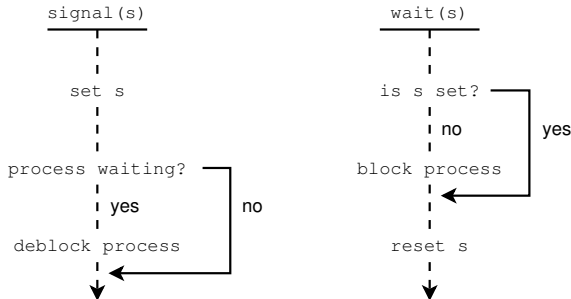
Most Simple Form of Signaling (Busy Waiting)



- 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**



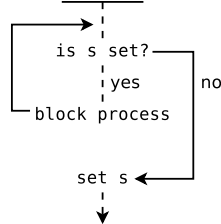
One way to specify in Linux an execution order with passive waiting, is by using the function `sigsuspend`. Thereby a process blocks itself until another process sends it an appropriate signal (usually `SIGUSR1` or `SIGUSR2`) with the command `kill` (or the system call of the same name) and in this way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

-
- The diagram illustrates the implementation of a semaphore using two mutexes and a counter variable s .
- Process 1 and Process 2:** Each process has a vertical timeline. Process 1's timeline shows a light blue shaded region labeled X between its `lock(s)` and `unlock(s)` operations. Process 2's timeline shows a light blue shaded region labeled Y between its `lock(s)` and `unlock(s)` operations. Arrows indicate that both processes can acquire either mutex, but only one can be held at a time. Specifically, Process 1 acquires its mutex first, then Process 2 acquires its mutex. After Process 2 finishes, it releases its mutex, and then Process 1 releases its mutex.
- Semaphore Operations:**
- unlock(s):** This operation is shown as a dashed line. It first performs a `reset s` (decrementing the counter). Then, it checks `process waiting?`. If the answer is `yes`, it performs `deblock process` (moving the process to the back of the queue). If the answer is `no`, it proceeds to the next step.
 - lock(s):** This operation is shown as a dashed line. It first checks `is s set?`. If the answer is `yes`, it performs `block process` (moving the process to the front of the queue). If the answer is `no`, it performs `set s` (incrementing the counter) and then proceeds to the next step.

- Prof. Dr. Christian Baun – 9th Slide Set Operating Systems – Frankfurt University of Applied Sciences – WS2021 14/75

Process 1



sigsuspend, kill, pause and sleep

- 15/75

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

- **Starvation**

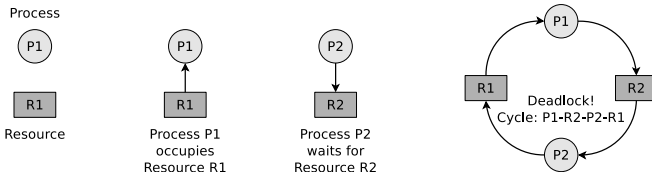
- **Deadlock**

-
- A diagram of a maze with a robot's path. The path is marked by red squares and black arrows. The path starts at the top-left, moves right, then down, then right again, ending at the bottom-right. Blue squares represent obstacles.



17/75

- 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 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 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 (2/2)

- If process 3 finished execution, it deallocates its resources

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

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

- 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
- 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 = $\begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$

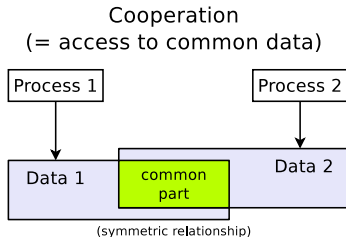
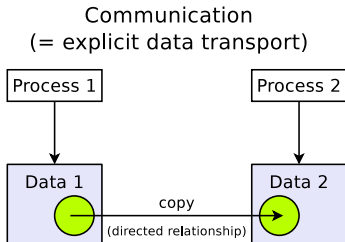
$$\text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

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

Communication of Processes

- Communication

- Shared Memory
- Message Queues
- Pipes
- Sockets



- 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 \implies inconsistencies

```
graph LR; X[Process X (Sender)] --> SM[Shared Memory]; Y[Process Y (Receiver)] --> SM; subgraph X_Box [ ] direction TB; X; X_UM[exclusive usable memory]; end; subgraph Y_Box [ ] direction TB; Y; Y_UM[exclusive usable memory]; end;
```

The diagram illustrates a Shared Memory architecture. It features three main components: Process X (Sender), Shared Memory, and Process Y (Receiver). Process X and Process Y are represented as rounded rectangles. Inside each process rectangle, there is a smaller rounded rectangle labeled "exclusive usable memory". Arrows point from both Process X and Process Y to the central Shared Memory block, which is also a rounded rectangle. This indicates that both processes can access the shared memory.

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

28/75

Attach a Shared Memory Segment (in C)

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget;
10    char *sharedmempointer;
11
12    // Create shared memory segment or access an existing one
13    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14    ...
15
16    // Attach shared memory segment
17    sharedmempointer = shmat(returncode_shmget, 0, 0);
18    if (sharedmempointer==(char *)-1) {
19        printf("Unable to attach the shared memory segment.\n");
20        perror("shmat");
21    } else {
22        printf("The shared memory segment has been attached %p\n", sharedmempointer);
23    }
24 }
25 }

```

```
$ ipcs -m
```

```
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00003039	56393780	bnc	600	20	1	

30/75

Detach a Shared Memory Segment (in C)

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget;
10    int returncode_shmdt;
11    char *sharedmempointer;
12
13    // Create shared memory segment or access an existing one
14    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
15    ...
16
17    // Attach the shared memory segment
18    sharedmempointer = shmat(returncode_shmget, 0, 0);
19    ...
20
21    // Detach the shared memory segment
22    returncode_shmdt = shmdt(sharedmempointer);
23    if (returncode_shmdt < 0) {
24        printf("Unable to detach the shared memory segment.\n");
25        perror("shmdt");
26    } else {
27        printf("The shared memory segment has been detached.\n");
28    }
29 }
30 }

```

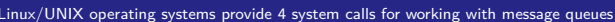
Erase a Shared Memory Segment (in C)

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget;
10    int returncode_shmctl;
11    char *sharedmempointer;
12
13    // Create shared memory segment or access an existing one
14    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
15    ...
16
17    // Erase shared memory segment
18    returncode_shmctl = shmctl(returncode_shmget, IPC_RMID, 0);
19    if (returncode_shmctl == -1) {
20        printf("Unable to erase the shared memory segment.\n");
21        perror("semctl");
22    } else {
23        printf("The shared memory segment has been erased.\n");
24    }
25 }
26 }

```


- 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



- The command `ipcs` provides information about existing message queues

Create Message Queues (in C)

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6
7 int main(int argc, char **argv) {
8     int returncode_msgget;
9
10    // Create message queue or access an existing one
11    // IPC_CREAT => create a message queue, if it does not still exist
12    // 0600 = Access privileges for the new message queue
13    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
14    if(returncode_msgget < 0) {
15        printf("Unable to create the message queue.\n");
16        exit(1);
17    } else {
18        printf("The message queue 12345 with the ID %i has been created.\n",
19              returncode_msgget);
20    }
21 }

```

```
$ 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
```

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6 #include <string.h>           // This header file is required for strcpy()
7
8 struct msgbuf {               // Template of a buffer for msgsnd and msgrcv
9     long mtype;               // Message type
10    char mtext[80];           // Send buffer
11 } msg;
12
13 int main(int argc, char **argv) {
14     int returncode_msgget;
15
16     // Create message queue or access an existing one
17     returncode_msgget = msgget(12345, IPC_CREAT | 0600);
18     ...
19
20     msg.mtype = 1;             // Specify the message type
21     strcpy(msg.mtext, "Testnachricht"); // Write the message into the send buffer
22
23     // Write a message into the message queue
24     if (msgsnd(returncode_msgget, &msg, strlen(msg.mtext), 0) == -1) {
25         printf("Unable to write the message into the message queue.\n");
26         exit(1);
27     }
28 }

```

- 35/75

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
```

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6 #include <string.h>           // This header file is required for strcpy()
7 struct msgbuf {              // Template of a buffer for msgsnd and msgrcv
8     long mtype;              // Message type
9     char mtext[80];          // Send buffer
10 } msg;
11
12 int main(int argc, char **argv) {
13     int returncode_msgget, returncode_msgrcv;
14     msg receivebuffer;        // Create a receive buffer
15
16     // Create message queue or access an existing one
17     returncode_msgget = msgget(12345, IPC_CREAT | 0600)
18
19     msg.mtype = 1;            // Pick the first message of type 1
20     // MSG_NOERROR => The message will be truncated when it is too long
21     // IPC_NOWAIT  => Do not block the process if no message exists
22     returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
23                                MSG_NOERROR | IPC_NOWAIT);
24     if (returncode_msgrcv < 0) {
25         printf("Unable to pick a message from the message queue.\n");
26         perror("msgrcv");
27     } else {
28         printf("This message was picked from the message queue: %s\n", msg.mtext);
29         printf("The received message is %i characters long.\n", returncode_msgrcv);
30     }
31 }

```

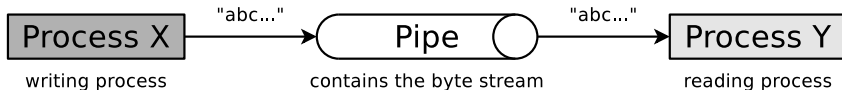
Erase a Message Queue (in C)

```
1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6
7 int main(int argc, char **argv) {
8     int returncode_msgget;
9     int returncode_msgctl;
10
11     // Create message queue or access an existing one
12     returncode_msgget = msgget(12345, IPC_CREAT | 0600);
13     ...
14
15     // Erase message queue
16     returncode_msgctl = msgctl(returncode_msgget, IPC_RMID, 0);
17     if (returncode_msgctl < 0) {
18         printf("Unable to erase the message queue with the ID %i.\n", returncode_msgget);
19         perror("msgctl");
20         exit(1);
21     } else {
22         printf("The message queue with the ID %i has been erased.\n", returncode_msgget);
23     }
24     exit(0);
25 }
```

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 \implies the writing process gets blocked
 - Pipe = empty \implies the reading process gets blocked
 - is created with the system call `pipe()`
 - During this process, the kernel of the operating system creates an Inode (\implies 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

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

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int pid_des_Kindes;
7     // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
8     int testpipe[2];
9
10    // Die Pipe testpipe anlegen
11    if (pipe(testpipe) < 0) {
12        printf("Das Anlegen der Pipe ist fehlgeschlagen.\n");
13        // Programmabbruch
14        exit(1);
15    } else {
16        printf("Die Pipe testpipe wurde angelegt.\n");
17    }
18
19    // Einen Kindprozess erzeugen
20    pid_des_Kindes = fork();
21
22    // Es kam beim fork zu einem Fehler
23    if (pid_des_Kindes < 0) {
24        perror("Es kam bei fork zu einem Fehler!\n");
25        // Programmabbruch
26        exit(1);
27    }
```

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

```

28 // Elternprozess
29 if (pid_des_Kindes > 0) {
30     printf("Elternprozess: PID: %i\n", getpid());
31     // Lesekanal der Pipe testpipe blockieren
32     close(testpipe[0]);
33     char nachricht[] = "Testnachricht";
34     // Daten in den Schreibkanal der Pipe schreiben
35     write(testpipe[1], &nachricht, sizeof(nachricht));
36 }
37
38 // Kindprozess
39 if (pid_des_Kindes == 0) {
40     printf("Kindprozess: PID: %i\n", getpid());
41     // Schreibkanal der Pipe testpipe blockieren
42     close(testpipe[1]);
43     // Einen Empfangspuffer mit 80 Zeichen Kapazität anlegen
44     char puffer[80];
45     // Daten aus dem Lesekanal der Pipe auslesen
46     read(testpipe[0], puffer, sizeof(puffer));
47     // Empfangene Daten ausgeben
48     printf("Empfangene Daten: %s\n", puffer);
49 }
50 }

```

```
$ gcc pipe_beispiel.c -o pipe_beispiel
```

```
$ ./pipe_beispiel
```

```
Die Pipe testpipe wurde angelegt.
```

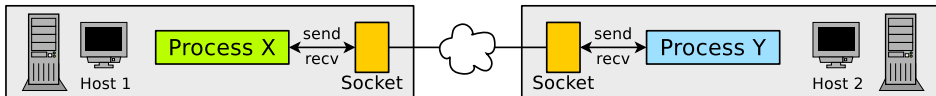
```
Elternprozess: PID: 6363
```

```
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

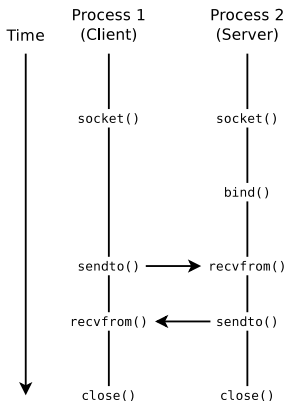
- **Connectionless sockets (= datagram sockets)**
 - Use the Transport Layer protocol UDP
 - Advantage: Better data rate as with TCP
 - Reason: Lesser overhead for the protocol
 - Drawback: Segments may arrive in wrong sequence or may get lost
- **Connection-oriented sockets (= stream sockets)**
 - Use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



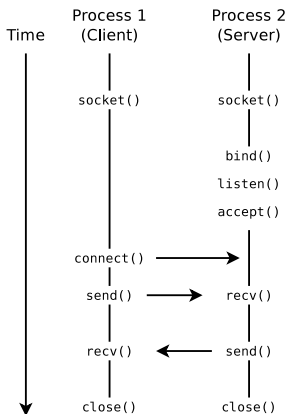
• Client

- 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);
```

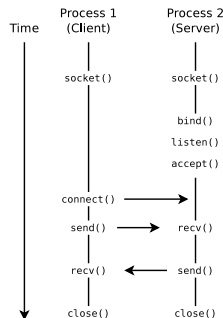
- A call of `socket()` returns an integer value
 - The value is called **socket descriptor** (*socket file descriptor*)
- `domain`: Specifies the protocol family
 - `PF_UNIX`: Local interprocess communication in Linux/UNIX
 - `PF_INET`: IPv4
 - `PF_INET6`: IPv6
- `type`: Specifies the type of the socket (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()`:

```
1 sd = socket(PF_INET, SOCK_STREAM, 0);  
2   if (sd < 0) {  
3       perror("The socket could not be created");  
4       return 1;  
5   }
```


Bind Address and Port Number: bind

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

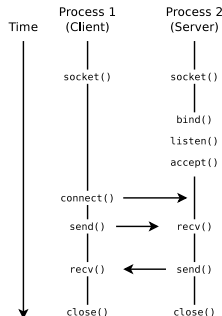
- `bind()` binds the newly created socket (`sd`) to the address (`address`) of the server
 - `sd` is the socket descriptor from the previous call of `socket()`
 - `address` is a data structure, which contains the IP address of the server and a port number
 - `addrlen` is the length of the data structure,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



Make a Server ready to receive Data: listen

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

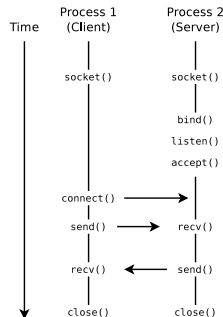
- `listen()` specifies how many connection requests can be buffered by the socket
 - If the `listen()` queue has no more free capacity, further connection requests from clients are rejected
 - `sd` is the socket descriptor from the previous call of `socket()`
 - `backlog` contains the number of possible 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



Establish a Connection by the Client

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

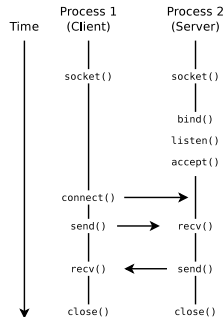
- Via `connect()`, the client tries to establish a connection to a server socket
- The client must know the address (hostname and port number) of the server
- `sd` is the socket descriptor
- `address` contains the address of the server
- `addrlen` is the length of the data structure, which contains the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

```
1 send(socket, "Hello World", 11, 0);  
2 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

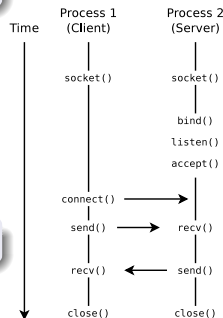
Close a Socket: close

```
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)`



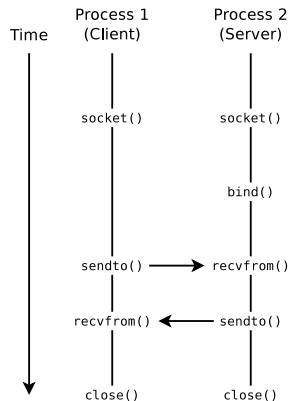
Sockets via UDP – Example (Server)

```

1  #!/usr/bin/env python
2  # Server: Receives a message via UDP
3
4  import socket                # Import module socket
5
6  # For all interfaces of the host
7  HOST = ''                    # '' = all interfaces
8  PORT = 50000                 # Port number of server
9
10 # Create socket and return socket descriptor
11 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
12
13 try:
14     sd.bind((HOST, PORT))     # Bind socket to port
15     while True:
16         # Receive data
17         data = sd.recvfrom(1024)
18         # Print received data
19         print 'Received:', repr(data)
20 finally:
21     sd.close()                # Close socket

```

```
$ python udp_server.py
```



Sockets via UDP – Example (Client)

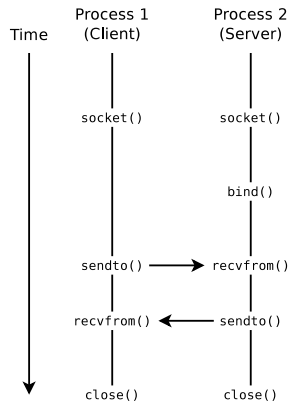
```

1  #!/usr/bin/env python
2  # Client: Sends a message via UDP
3
4  import socket                # Import module socket
5
6  HOST = 'localhost'           # Hostname of Server
7  PORT = 50000                 # Port number of Server
8  MESSAGE = 'Hello World'     # Message
9
10 # Create socket and return socket descriptor
11 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
12
13 # Send message to socket
14 sd.sendto(MESSAGE, (HOST, PORT))
15
16 sd.close()                   # Close socket

```

```
$ python udp_client.py
```

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



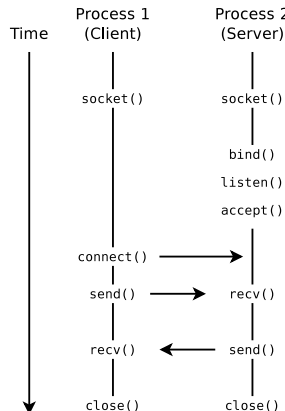
Sockets via TCP – Example (Server)

```

1  #!/usr/bin/env python
2  # Echo Server via TCP
3  import socket                # Import module socket
4
5  HOST = ''                    # '' = all interfaces
6  PORT = 50007                 # Port number of server
7
8  # Create socket and return socket descriptor
9  sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 # Bind socket to port
11 sd.bind((HOST, PORT))
12 # Make socket ready to receive
13 # Max. number of connections = 1
14 sd.listen(1)
15 # Socket accepts connections
16 conn, addr = sd.accept()
17
18 print 'Connected by', addr
19
20 while 1:                      # Infinite loop
21     data = conn.recv(1024)    # Receive data
22     if not data: break        # Break infinite loop
23     conn.send(data)           # Send back received data
24
25 conn.close()                  # Close socket

```

```
$ python tcp_server.py
```



Sockets via TCP – Example (Client)

```

1  #!/usr/bin/env python
2  # Echo Client via TDP
3
4  import socket                # Import module socket
5
6  HOST = 'localhost'           # Hostname of Server
7  PORT = 50007                 # Port number of server
8
9  # Create socket and return socket descriptor
10 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11 # Connect with server socket
12 sd.connect((HOST, PORT))
13
14 sd.send('Hello, world')       # Send data
15 data = sd.recv(1024)          # Receive data
16 sd.close()                   # Close socket
17
18 # Print received data
19 print 'Empfangen:', repr(data)

```

```

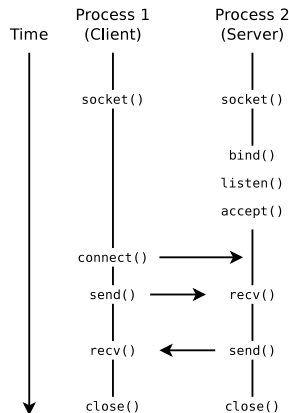
$ python tcp_client.py
Empfangen: 'Hello, world'

```

```

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

```



Non-blocking Sockets - some Impacts

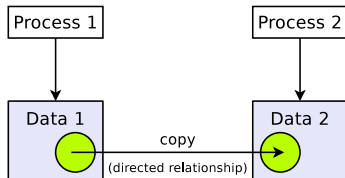
- `recv()` and `recvfrom()`
 - The method return data only, when they are already stored in the buffer
 - If the buffer does not contain any data, the method throws an **exception** and the program execution **continues**
- `send()` and `sendto()`
 - The methods send the specified data only, when they 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

Cooperation

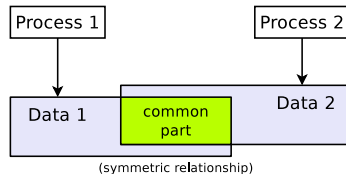
- Cooperation

- 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**
- 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** \implies can not be interrupted (indivisible)
- May allow multiple processes accessing the critical section
 - In contrast to semaphores, can locks (\implies 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>

Image Source: Carsten Vogt

- ```
1 SEM.P() {
2 // if the counter variable = 0, the process becomes blocked
3 if (SEM.COUNT == 0)
4 < block >
5
6 // if the counter variable is > 0, the counter variable
7 // is decremented immediately by 1
8 SEM.COUNT = SEM.COUNT - 1;
9 }
```

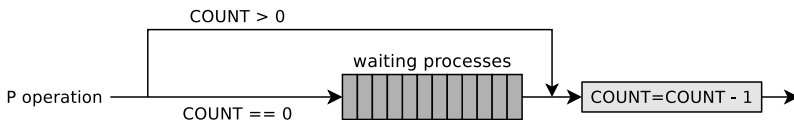


Image Source: Carsten Vogt

- 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
5 // if processes are in the waiting room, one gets deblocked
6 if (< SEM waiting room is not empty >)
7 < deblock a waiting process >
8 }
```

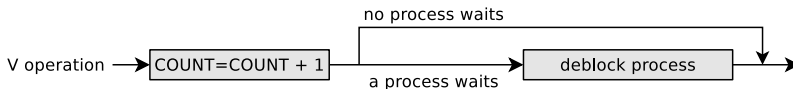












Image Source: Carsten Vogt

- 
- Diagram illustrating the structure of a semaphore table:
- The table is indexed by **Group number** (0, 1, 2, 3, ..., n).
  - Each entry in the table points to a **Semaphore group**.
  - The semaphore groups are organized as follows:
    - Group 0: Contains semaphores  $S_{00}, S_{01}, S_{02}, S_{03}, S_{04}, S_{05}$  (6 semaphores).
    - Group 1: Contains semaphores  $S_{10}, S_{11}$  (2 semaphores).
    - Group 2: Contains semaphores  $S_{20}, S_{21}, S_{22}$  (3 semaphores).
    - Group 3: Contains semaphores  $S_{30}, S_{31}, S_{32}, S_{33}, S_{34}$  (5 semaphores).
    - Group n: Is **empty**.
  - Each semaphore is labeled with its **Semaphore number within the group** (e.g., 0, 1, 2, 3, 4, 5 for Group 0).
  - An **individual semaphore** is highlighted within the semaphore groups.

- `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 **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 2 states: **occupied** and **not occupied**
    - Mutexes have the same functionality as **binary semaphores**

2 functions for accessing a Mutex exist

mutex\_lock  $\Rightarrow$  corresponds to the P operation

`mutex_unlock`  $\Rightarrow$  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 shmids] [-q msgids] [-s semids]
 [-M shmkeys] [-Q msgkeys] [-S semkeys]
```

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