

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









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

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

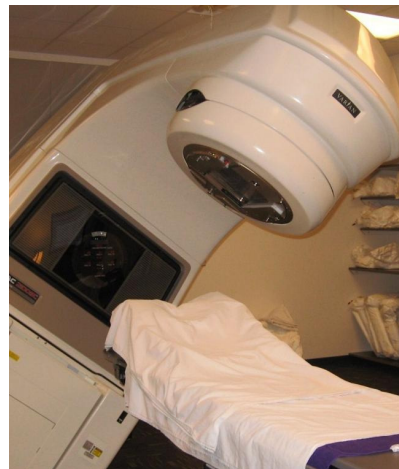


Image source: Google image search.  
Frequently shown picture in this context.  
(author and license: unknown)

*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](http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html)

- 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

<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

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/>

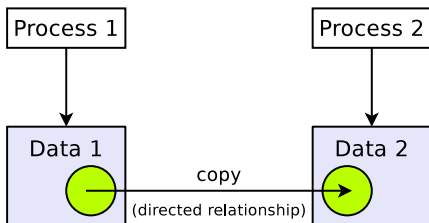


# Communication vs. Cooperation

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

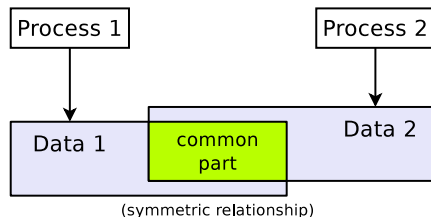
## Communication

(= explicit data transport)



## Cooperation

(= access to common data)



---

- \_\_\_\_\_

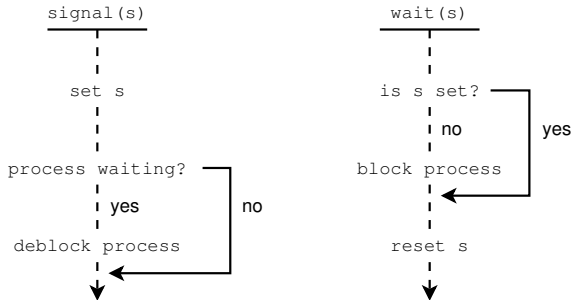




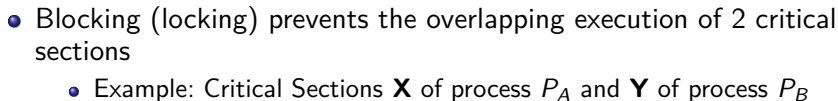


## 13/81

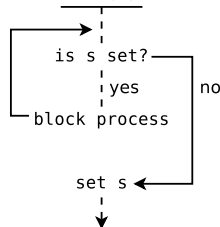
- Alternative system calls and function calls by which a process can block itself until it is woken up again by a system call are **pause** and **sleep**



- 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`



## Process 1



sigsuspend, kill, pause and sleep

- 15/81

## 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















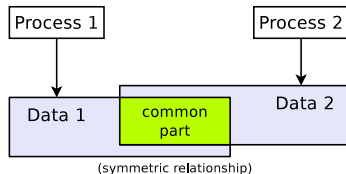
## Conclusion about Deadlocks

- 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)

---

- Shared Memory
- Message Queues
- Pipes
- Sockets

Cooperation  
(= access to common data)





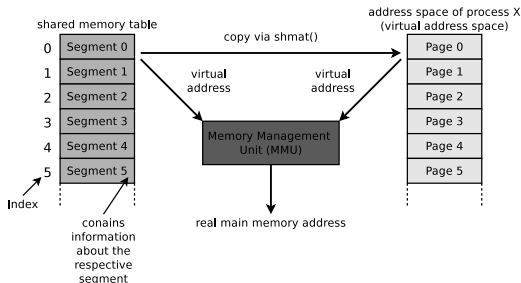
- Interprocess communication via a shared memory is also called **memory-based communication**
- **Shared memory segments** are memory areas, which can be accessed by multiple processes
  - These memory areas are located in the address space of multiple processes
- The processes need to coordinate the 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_mem(exclusive usable memory); end; subgraph Y_Box [ ] direction TB; Y; Y_mem(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 by 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.

# 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

1. *Journal of Management Studies*, 1997, 34, 1, 1-14.

\_\_\_\_\_

## Create a Shared Memory Segment (in C)

```

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

```

```
$ 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)

```

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	

1000

Prof. Dr. Christian Baun – 9th Slide Set Operating Systems – Frankfurt University of Applied Sciences – WS2122 30/81

## 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)

```

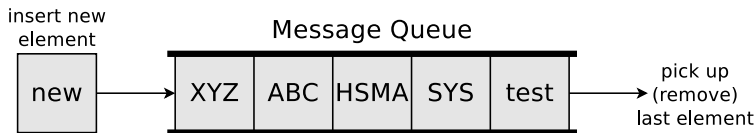
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget;
10    int returncode_shmctl;
11    char *sharedmempointer;
12
13    // Create shared 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 }

```



# Message Queues

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



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 ( $\Rightarrow$  send operation)
- `msgrcv()`: Read messages from message queues ( $\Rightarrow$  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)

```

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 }

```

- 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
```

```

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

- An **anonymous Pipe**...

- 
- The diagram illustrates a pipe as a communication channel. On the left, a box labeled "Process X" is identified as the "writing process". An arrow labeled "abc..." points from Process X to a central pipe icon, which is labeled "Pipe" and described as "contains the byte stream". Another arrow labeled "abc..." points from the pipe to a box on the right labeled "Process Y", which is identified as the "reading process".

## 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



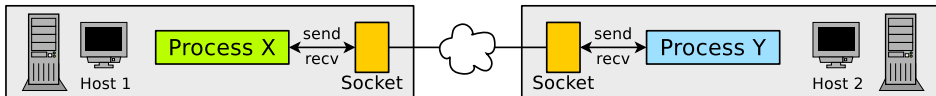
One example of working with named pipes in Linux can be found on the website of this course

41/81



# 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

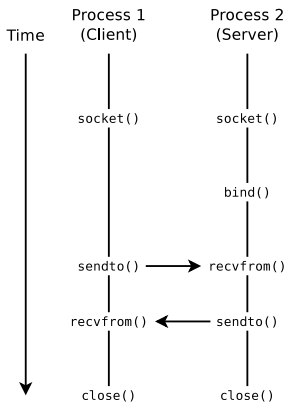
- **Connectionless sockets (= datagram sockets)**
  - Use the Transport Layer protocol UDP
  - Advantage: Better data rate as with TCP
    - Reason: Lesser overhead for the protocol
  - Drawback: Segments may arrive in wrong sequence or may get lost
- **Connection-oriented sockets (= stream sockets)**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



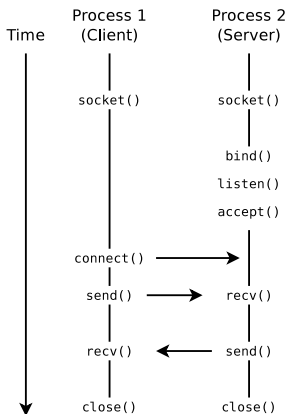
## • Client

- 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()`:

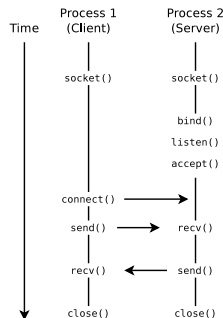
```
1 sd = socket(PF_INET, SOCK_STREAM, 0);
2   if (sd < 0) {
3       perror("The socket could not be created");
4       return 1;
5   }
```



# Bind Address and Port Number: bind

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

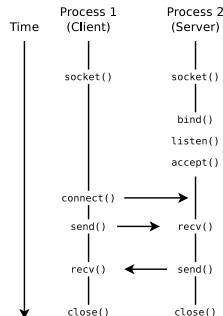
- `bind()` binds the newly created socket (`sd`) to the address (`address`) of the server
  - `sd` is the socket descriptor from the previous call of `socket()`
  - `address` is a data structure, which contains the IP address of the server and a port number
  - `addrlen` is the length of the data structure,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



# Make a Server ready to receive Data: listen

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

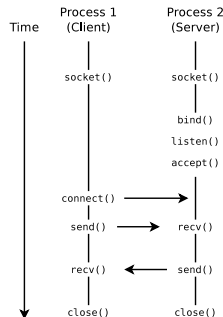
- `listen()` specifies how many connection requests can be buffered by the socket
  - If the `listen()` queue has no more free capacity, further connection requests from clients are rejected
  - `sd` is the socket descriptor from the previous call of `socket()`
  - `backlog` contains the number of possible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



# Accept a Connection Request: `accept`

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

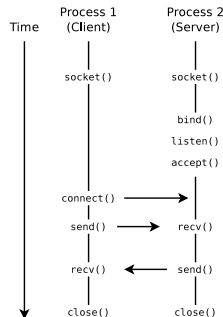
- `accept()` is used by the server to fetch the first connection request from the queue
- The return value is the socket descriptor of the new socket
- If the queue contains no connection requests, the process is blocked until a connection request arrives
- `address` contains the address of the client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

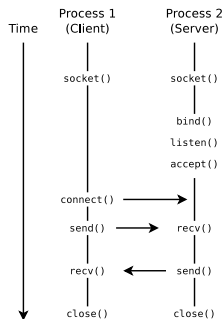
- Via `connect()`, the client tries to establish a connection to a server socket
- The client must know the address (hostname and port number) of the server
- `sd` is the socket descriptor
- `address` contains the address of the server
- `addrlen` is the length of the data structure, which contains the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

```
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

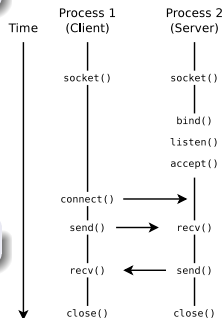
# Close a Socket: close

```
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)`





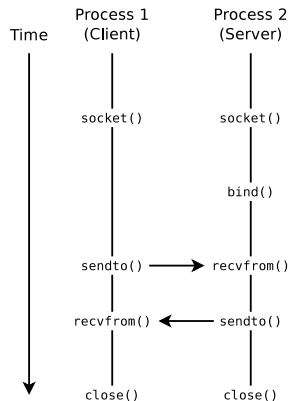
# Sockets via UDP – Example (Server)

```

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)

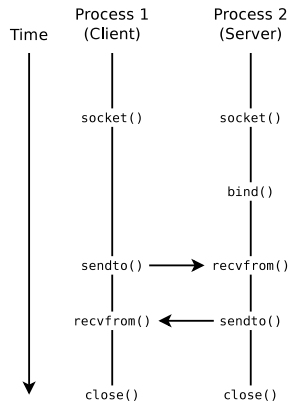
```

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))
```



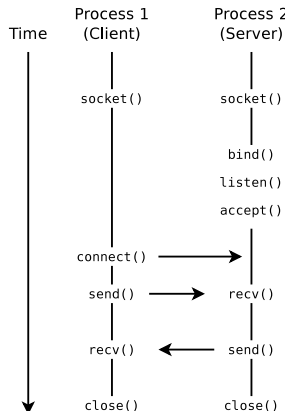
# Sockets via TCP – Example (Server)

```

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 sd.close()                   # Close socket

```

```
$ python tcp_server.py
```



## Sockets via TCP – Example (Client)

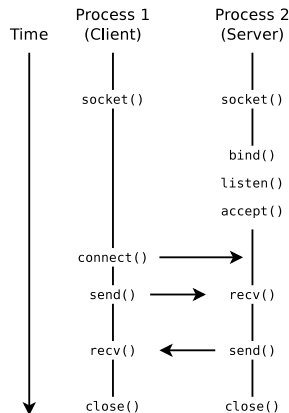
```

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)
```



## 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)`  $\implies$  switches into non-blocking mode
  - `sd.setblocking(1)`  $\implies$  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
<b>Sort of communication</b>	Memory-based	Message-based	Message-based	Message-based
<b>Bidirectional</b>	yes	no	no	yes
<b>Platform independent</b>	no	no	no	yes
<b>Processes must be related with each other</b>	no	no	for anon. pipes	no
<b>Communication over computer boundaries</b>	no	no	no	yes
<b>Remain intact without a bound process</b>	yes	yes	no	no
<b>Automatic synchronization</b>	no	yes	yes	yes

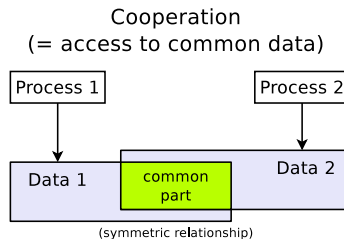
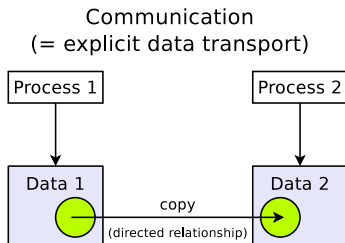
- Advantages of message-based communication versus memory-based communication:
  - The operating system takes care about the synchronization of accesses  
⇒ comfortable
  - Can be used in distributed systems without a shared memory
  - Better portability of applications

Storage can be integrated via network connections

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

# Cooperation

- Cooperation
  - Semaphore
  - 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**  $\implies$  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 }
```

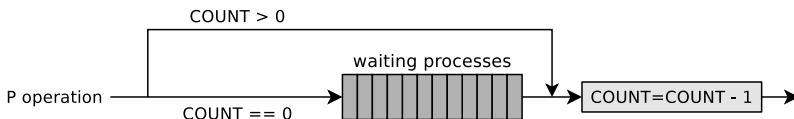
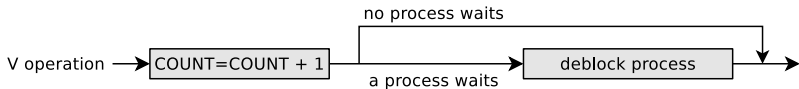


Image Source: Carsten Vogt

- ```
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 }
```



This program creates a child process. The parent process and the child process both try to print characters in the command line interface (critical section). Each process may print only one character at a time. Two semaphores are used to ensure mutual exclusion

Helpful documentation of `semget`

69/81

```

25 // Neue Semaphoregruppe 54321 mit einer Semaphore erstellen
26 returncode_semget2 = semget(sem_key2, 1, IPC_CREAT | IPC_EXCL | 0600);
27 if (returncode_semget2 < 0) {
28     printf("Die Semaphoregruppe %i konnte nicht erstellt werden.\n", sem_key2);
29     perror("semget");
30     exit(1);
31 }
32
33 // P-Operation definieren. Wert der Semaphore um eins dekrementieren
34 struct sembuf p_operation = {0, -1, 0};
35
36 // V-Operation definieren. Wert der Semaphore um eins inkrementieren
37 struct sembuf v_operation = {0, 1, 0};
38
39 // Erste Semaphore der Semaphoregruppe 12345 initial auf Wert 1 setzen
40 returncode_semctl = semctl(returncode_semget1, 0, SETVAL, 1);
41
42 // Erste Semaphore der Semaphoregruppe 54321 initial auf Wert 0 setzen
43 returncode_semctl = semctl(returncode_semget2, 0, SETVAL, 0);
44
45 // Initialen Wert der ersten Semaphore der Semaphoregruppe 12345 zur Kontrolle ausgeben
46 output = semctl(returncode_semget1, 0, GETVAL, 0);
47 printf("Wert der Semaphore mit ID %i und Key %i: %i\n", returncode_semget1, sem_key1, output);
48
49 // Initialen Wert der ersten Semaphore der Semaphoregruppe 54321 zur Kontrolle ausgeben
50 output = semctl(returncode_semget2, 0, GETVAL, 0);
51 printf("Wert der Semaphore mit ID %i und Key %i: %i\n", returncode_semget2, sem_key2, output);

```

Helpful documentation of `semctl`

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semctl/semctl.html>

```

52 // Einen Kindprozess erzeugen
53 pid_des_kindess = fork();
54
55 // Kindprozess
56 if (pid_des_kindess == 0) {
57     for (int i=0;i<5;i++) {
58         semop(returncode_semget2, &p_operation, 1); // P-Operation Semaphore 54321
59         // Kritischer Abschnitt (Anfang)
60         printf("B");
61         sleep(1);
62         // Kritischer Abschnitt (Ende)
63         semop(returncode_semget1, &v_operation, 1); // V-Operation Semaphore 12345
64     }
65     exit(0);
66 }
67
68 // Elternprozess
69 if (pid_des_kindess > 0) {
70     for (int i=0;i<5;i++) {
71         semop(returncode_semget1, &p_operation, 1); // P-Operation Semaphore 12345
72         // Kritischer Abschnitt (Anfang)
73         printf("A");
74         sleep(1);
75         // Kritischer Abschnitt (Ende)
76         semop(returncode_semget2, &v_operation, 1); // V-Operation Semaphore 54321
77     }
78 }

```

Helpful documentation of `semop`

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semop/semop.html>

```

79 // Warten auf die Beendigung des Kindprozesses
80 wait(NULL);
81
82 printf("\n");
83
84 // Semaphorgruppe 12345 entfernen
85 returncode_semctl = semctl(returncode_semget1, 0, IPC_RMID, 0);
86 if (returncode_semctl < 0) {
87     printf("Die Semaphorgruppe %i konnte nicht entfernt werden.\n", returncode_semget1);
88     exit(1);
89 } else {
90     printf("Die Semaphorgruppe mit ID %i und Key %i wurde entfernt.\n", returncode_semget1, sem_key1);
91 }
92
93 // Semaphorgruppe 54321 entfernen
94 returncode_semctl = semctl(returncode_semget2, 0, IPC_RMID, 0);
95 if (returncode_semctl < 0) {
96     printf("Die Semaphorgruppe %i konnte nicht entfernt werden.\n", returncode_semget2);
97     exit(1);
98 } else {
99     printf("Die Semaphorgruppe mit ID %i und Key %i wurde entfernt.\n", returncode_semget2, sem_key2);
100 }
101
102 exit(0);
103 }

```

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



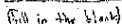
```
$ gcc semaphore_beispiel_systemv.c -o semaphore_beispiel_systemv
Wert der Semaphore mit ID 98362 und Key 12345: 1
Wert der Semaphore mit ID 98363 und Key 54321: 0
ABABABABAB
Die Semaphore mit ID 98362 und Key 12345 wurde entfernt.
Die Semaphore mit ID 98363 und Key 54321 wurde entfernt.
```

```
$ printf "%d\n" 0x00003039          # Convert from hexadecimal to decimal
12345
$ printf "%d\n" 0x0000d431
54321
```

- Without mutual exclusion by using the semaphores, the output sequence can be e.g. ABBABABABA or ABBAABABAB or ABABABABBA ...
- Without mutual exclusion by using the semaphores and without the sleep commands, the output sequence is usually AAAAABBBBBB and in rather seldom cases like AABAAABBBBB



Michael Vignaro



75/81



## Producer/Consumer Example (3/3)

```

1 typedef int semaphore;           // semaphores are of type integer
2 semaphore filled = 0;           // counts the number of occupied locations in the buffer
3 semaphore empty = 8;            // counts the number of empty locations in the buffer
4 semaphore mutex = 1;            // controls access to the critical sections
5
6 void producer (void) {
7     int data;
8
9     while (TRUE) {               // infinite loop
10        createDatapacket(data);   // create data packet
11        P(empty);                 // decrement the empty locations counter
12        P(mutex);                // enter the critical section
13        insertDatapacket(data);   // write data packet into the buffer
14        V(mutex);                // leave the critical section
15        V(filled);               // increment the occupied locations counter
16    }
17 }
18
19 void consumer (void) {
20     int data;
21
22     while (TRUE) {               // infinite loop
23        P(filled);                // decrement the occupied locations counter
24        P(mutex);                // enter the critical section
25        removeDatapacket(data);   // pick data packet from the buffer
26        V(mutex);                // leave the critical section
27        V(empty);                // increment the empty locations counter
28        consumeDatapacket(data); // consume data packet
29    }
30 }

```

Image Source: Carsten Vogt

- 
- Diagram illustrating the structure of a semaphore table:
- Group number:** Indexes the rows of the semaphore table (0, 1, 2, 3, ..., n).
  - Semaphore table:** A table where each row (group) points to a specific semaphore group.
  - Semaphore number within the group:** Indexes the individual semaphores within a group (0, 1, 2, 3, 4, 5).
  - Individual semaphore:** A single semaphore element, denoted as  $S_{ij}$  (e.g.,  $S_{00}$ ,  $S_{10}$ ,  $S_{20}$ ,  $S_{30}$ ).
  - Semaphore group:** A collection of individual semaphores belonging to a specific group (e.g.,  $S_{00}$  through  $S_{05}$ ).

- `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 (**SystemV-IPC**) existing semaphores provides the command `ipcs`

## Semaphores in Linux (System V vs. POSIX)

- The concept of protecting critical sections described in so far is also called **system V semaphores** in the literature
  - System V semaphores are implemented in kernel space
    - $\implies$  cause context switching (see slide set 8).
- Besides this, **POSIX semaphores** exist
  - They operate only in user space
    - $\implies$  do not cause context switching
    - $\implies$  consume fewer resources (they are „more lightweight“).
  - POSIX semaphore API is (perhaps) more intuitive and easier to learn

C function calls of the POSIX semaphores specified in the header file `semaphore.h`

- `sem_init`: Create a new **unnamed** semaphore and thereby specify the initial value
- `sem_open`: Create a new **named** semaphore and thereby specify the initial value
- `sem_post`: Increment the value of a semaphore (V operation)
- `sem_wait`: Decrement the value of a semaphore (P operation). Alternatives: `sem_trywait` or `sem_timedwait`
- `sem_getvalue`: Request the value of a semaphore
- `sem_destroy`: Erase an **unnamed** semaphore
- `sem_unlink`: Erase a **named** semaphore

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





