

Critical Sections

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

Critical Sections – Example: Print Spooler

Process X

```
next_free_slot = in; (16)
```

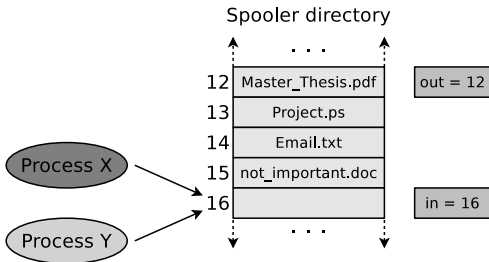
```
Store record in next_free_slot; (16)
in = next_free_slot + 1; (17)
```

Process Y

Process switch

```
next_free_slot = in; (16)
Store record in next_free_slot; (16)
in = next_free_slot + 1; (17)
```

Process switch



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

- **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 64)

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

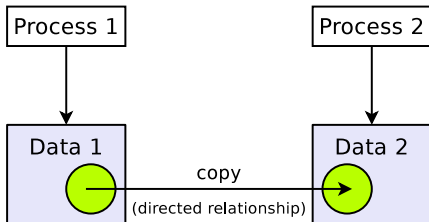
Image source: Google image search



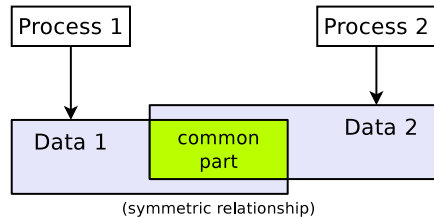
Communication vs. Cooperation

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

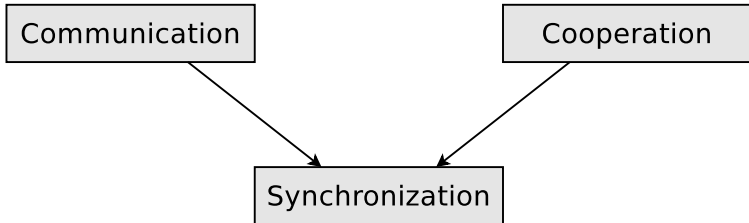
Communication
(= explicit data transport)



Cooperation
(= access to common data)



- 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**



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**)

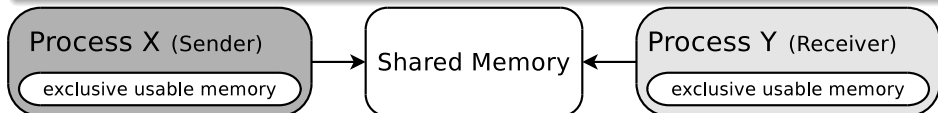
Summary: Difference between Signaling and Blocking

- **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

Shared Memory

- Interprocess communication via a shared memory is also called **memory-based communication**
- **Shared memory segments** are memory areas, which can be accessed by multiple processes
 - These memory areas are located in the address space of multiple processes
- The processes need to coordinate the access operations by themselves and ensure that their memory requests are mutually exclusive
 - A receiver process, cannot read data from the shared memory, before the sender process has finished its current write operation
 - If access operations are not coordinated carefully \implies inconsistencies

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



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

27 / 78

28/78

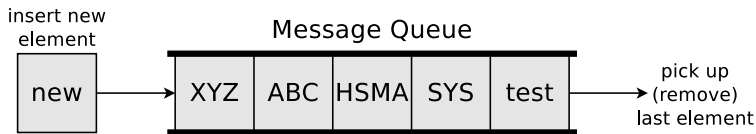
```

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, returncode_shmdt, returncode_sprintf;
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    // Attach shared memory segment
16    sharedmempointer = shmat(returncode_shmget, 0, 0);
17    ...
18
19    // Write a string into the shared memory segment
20    returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
21    if (returncode_sprintf < 0) {
22        printf("The write operation did fail.\n");
23    } else {
24        printf("%i chareacters written into the segment.\n", returncode_sprintf);
25    }
26
27    // Read the string from the shared memory segment
28    if (printf ("%s\n", sharedmempointer) < 0) {
29        printf("The read operation did fail.\n");
30    }
31    ...

```


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


```
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    }
```

33/78

```

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
```



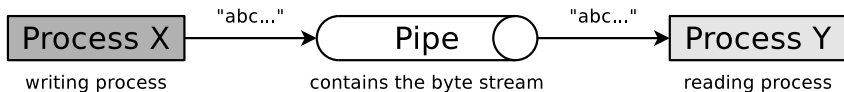
```
1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6
7 int main(int argc, char **argv) {
8     int returncode_msgget;
9     int returncode_msgctl;
10
11     // Create message queue or access an existing one
12     returncode_msgget = 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 }
```

37 / 78

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    }
```

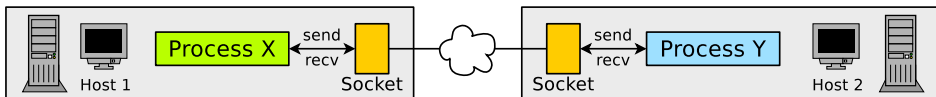

An Anonymous Pipe Example (in C) – Part 2/2

```
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

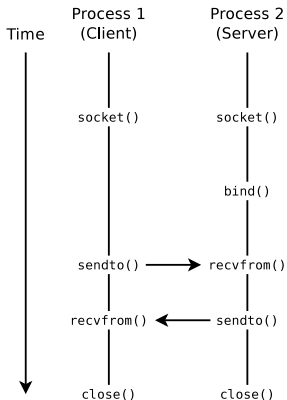
- **Connectionless sockets (= datagram sockets)**
 - Use the Transport Layer protocol UDP
 - Advantage: Better data rate as with TCP
 - Reason: Lesser overhead for the protocol
 - Drawback: Segments may arrive in wrong sequence or may get lost
- **Connection-oriented sockets (= stream sockets)**
 - Use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



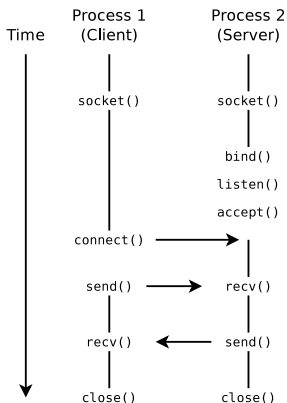
• Client

- 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);
```

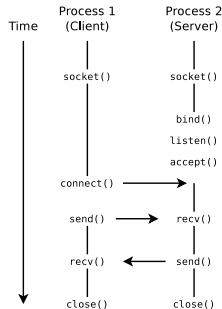
- A call of `socket()` returns an integer value
 - The value is called **socket descriptor** (*socket file descriptor*)
- `domain`: Specifies the protocol family
 - `PF_UNIX`: Local interprocess communication in Linux/UNIX
 - `PF_INET`: IPv4
 - `PF_INET6`: IPv6
- `type`: Specifies the type of the socket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

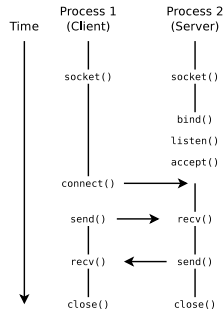
- `bind()` binds the newly created socket (`sd`) to the address (`address`) of the server
 - `sd` is the socket descriptor from the previous call of `socket()`
 - `address` is a data structure, which contains the IP address of the server and a port number
 - `addrlen` is the length of the data structure,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



Make a Server ready to receive Data: listen

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

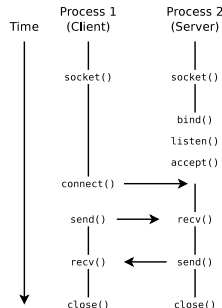
- `listen()` specifies how many connection requests can be buffered by the socket
 - If the `listen()` queue has no more free capacity, further connection requests from clients are rejected
 - `sd` is the socket descriptor from the previous call of `socket()`
 - `backlog` contains the number of possible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



Accept a Connection Request: `accept`

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

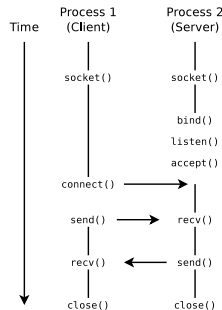
- `accept()` is used by the server to fetch the first connection request from the queue
- The return value is the socket descriptor of the new socket
- If the queue contains no connection requests, the process is blocked until a connection request arrives
- `address` contains the address of the client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

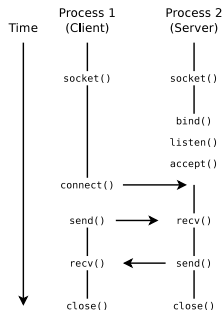
- Via `connect()`, the client tries to establish a connection to a server socket
- The client must know the address (hostname and port number) of the server
- `sd` is the socket descriptor
- `address` contains the address of the server
- `addrlen` is the length of the data structure, which contains the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

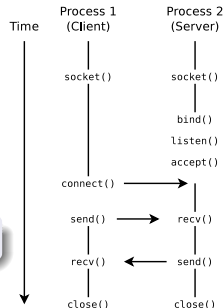

Close a Socket: close

```
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);
```

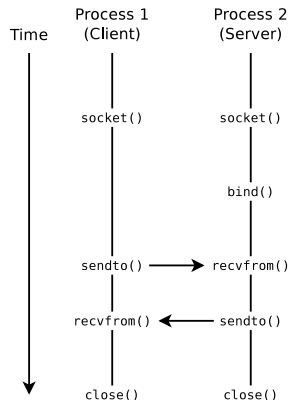
- If `close()` is used instead of `shutdown()`, this corresponds to a `shutdown(sd,2)`



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 deskriptor
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 out received data
19         print 'Received:', repr(data)
20 finally:
21     sd.close()               # Close socket
```

```
$ python udp_server.py
```

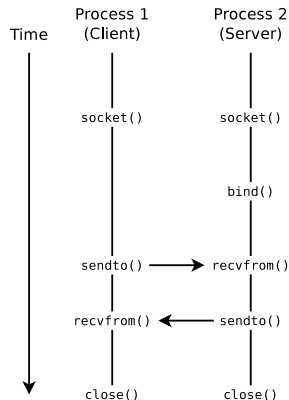


Sockets via UDP – Example (Client)

```
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))
```

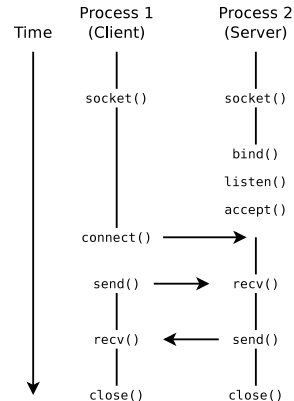


Sockets via TCP – Example (Server)

```

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

```



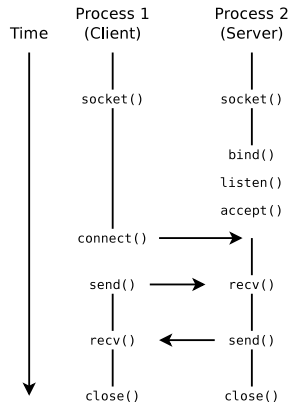
```
$ python tcp_server.py
```

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 out 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

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

- Advantages of message-based communication versus memory-based communication:
 - The operating system takes care about the synchronization of accesses
⇒ comfortable because the user processes do not need to take care about the synchronization
 - Can be used in distributed systems without a shared memory
 - Better portability of applications

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

Semaphore Access Operations (1/3)

A Semaphore consists of 2 Data Structures

- **COUNT:** An **integer, non-negative counter variable**.
Specifies how many processes can pass the semaphore now without getting blocked
 - A waiting room for the processes, which **wait** until they are allowed to pass the semaphore
The processes are in blocked state until they are transferred into ready state by the operating system when the semaphore allows to access the critical section
-
- **Initialization:** First, a new semaphore is created or an existing one is opened
 - For a new semaphore, the counter variable is initialized at the beginning with a non-negative initial value

```
1 // apply the INIT operation on semaphore SEM
2 SEM.INIT(unsigned int init_value) {
3
4     // initialize the variable COUNT of Semaphor SEM
5     // with a non-negative initial value
6     SEM.COUNT = init_value;
7 }
```


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 }
```

```

1 // Initialization of semaphores
2 s_init (Sema_Ping, 1);
3 s_init (Sema_Pong, 0);
4
5 task Ping is
6 begin
7     loop
8         P(Sema_Ping);
9         print("Ping");
10        V(Sema_Pong);
11    end loop;
12 end Ping;
13
14 task Pong is
15 begin
16     loop
17         P(Sema_Pong);
18         print("Pong, ");
19         V(Sema_Ping);
20    end loop;
21 end Pong;

```

- The two endless-running processes and Ping print out continuously PingPong, PingPong, PingPong,...

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`