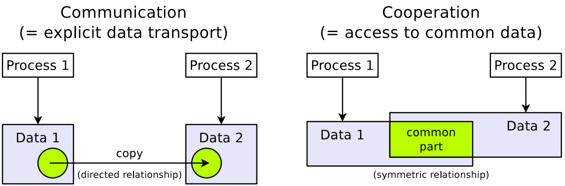
10th Slide Set
Operating Systems

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

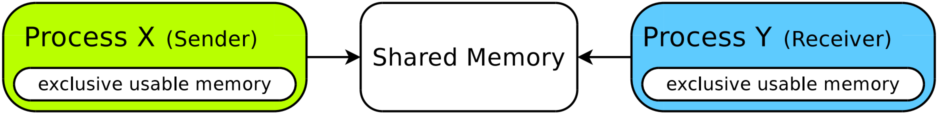
- At the end of this slide set You know/understand...
  - different options to implement **communication** between processes:
    - **Shared memory**
    - **Message queues**
    - **Pipes**
    - **Sockets**
  - different options to implement **cooperation** between processes
    - how critical sections can be protected via **semaphores**
    - the difference between **semaphore** and **mutex**



Exercise sheet 10 repeats the contents of this slide set which are relevant for these learning objectives

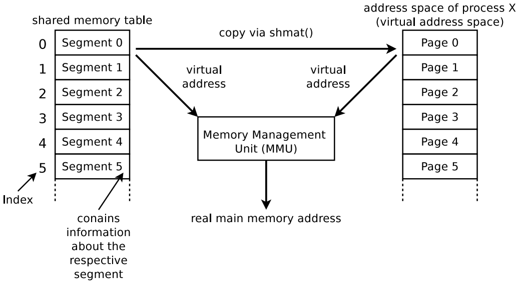
Shared Memory

- Inter-process 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 accesses themselves and to ensure that their memory accesses are mutually exclusive
  - A receiver process, cannot read data from the shared memory, before the sender process has finished its current write operation
  - If access operations are not coordinated carefully
    $\Longrightarrow$ inconsistencies occur



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

Working with Shared Memory

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

- `shmget()`: Create shared memory segments
- `shmat()`: Attach shared memory segments to processes
- `shmdt()`: Detach shared memory segments from processes
- `shmctl()`: Request status information (e.g. privileges) about shared memory segments, modify and erase shared memory segments

A well explained example about workng with shared memory provides. . .

`http://openbook.rheinwerk-verlag.de/unix_guru/node393.html`

`ipcs`

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

Create a Shared Memory Segment (in C)

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

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

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

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

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

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

Attach a Shared Memory Segment (in C)

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

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

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

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

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

Detach a Shared Memory Segment (in C)

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

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

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

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

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

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

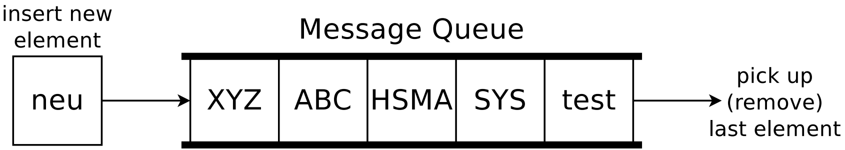
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

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

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

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

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

Erase a Shared Memory Segment (in C)

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

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

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

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

Message Queues

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



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

- `msgget()`: Create message queues
- `msgsnd()`: Send messages into message queues ($\Longrightarrow$ write operation)
- `msgrcv()`: Receive messages from message queues ($\Longrightarrow$ read operation)
- `msgctl()`: Request status information (e.g. privileges) of message queues, modify and erase message queues

Create Message Queues (in C)

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

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

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

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

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

Store Messages inside Message Queues (in C)

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

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

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

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

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

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

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

Result of storing a Message inside a Message Queue

- Before...

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

- Afterwards...

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

Pick a Message from a Message Queue (in C)

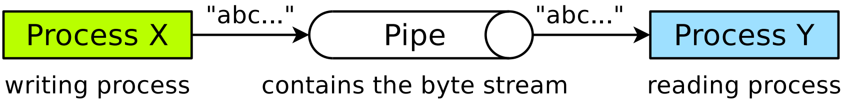
```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>                      // This header file is required for strcpy()
struct msgbuf {                          // Template of a buffer for msgsnd and msgrcv
    long mtype;    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

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

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

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

Erase a Message Queue (in C)

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

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

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

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

    exit(0);
}
```

Pipes (1/4)

- A pipe is like a channel or a tube, which allows a buffered unidirectional flow of data between 2 processes
  - Can always operate only between 2 processes
  - Operate according to the FIFO principle
  - Have limited capacity
  - Pipe = filled $\Longrightarrow$ the writing process gets blocked
  - Pipe = empty $\Longrightarrow$ the reading process gets blocked
  - Are created with system call `pipe()`
    - Creates an inode ($\Longrightarrow$ slide set 6) and 2 file descriptors (*handles*)
  - Processes carry out `read()` and `write()` system calls on the file descriptors to read data from the pipe and to write data into the pipe



Pipes (2/4)

- When child processes are created with `fork()`, the child processes also inherit access to the file descriptors
- 2 sorts of pipes exist:
  - **Anonymous pipes** and **named pipes**
- **Anonymous pipes** provide process communication only between closely related processes
  - Communication only works in one direction ($\Longrightarrow$ unidirectional)
  - 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

Pipes (3/4)

- Processes, which are not closely related with each other, can communicate via **named pipes**
  - These pipes can be accessed by using their names
  - Any process, which knows the name of a pipe, can use the name to access the pipe and communicate with other processes
- The operating system ensures **mutual exclusion**
  - At any time, only a single process can access a pipe

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

Pipes in the shell

A pipe forwards the output of a process into the input of another process and it is created in the shell with the | character. An example is:

```
cat /path/to/the/file.txt | grep search_pattern
```

Programming with Pipes (in C)

- Create a pipe:

```c
// Create the pipe testpipe
if (pipe(testpipe) < 0) {
    // It the pipe could not be created, the program is terminated
    printf("The pipe testpipe could not be created.\n");
    exit(1);
} else {
    printf("The pipe testpipe has been created.\n");
}
```

- Prepare a pipe for writing (after that the pipe can receive data):

```c
close(testpipe[0]);                  // Block the read channel of the pipe testpipe
open(testpipe[1]);                   // Open the write channel of the pipe testpipe
```

- Prepare a pipe for reading (after that the pipe can be read out):

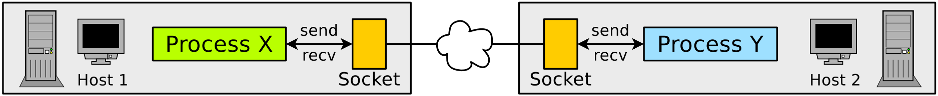
```c
close(testpipe[1]);                  // Block the write channel of the pipe testpipe
open(testpipe[0]);                   // Open the read channel of the pipe testpipe
```

- Read from a pipe and write into a pipe:

```c
read(testpipe[0], &buffervariable, sizeof(buffervariable));
write(testpipe[1], &buffervariable, sizeof(buffervariable));
```

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

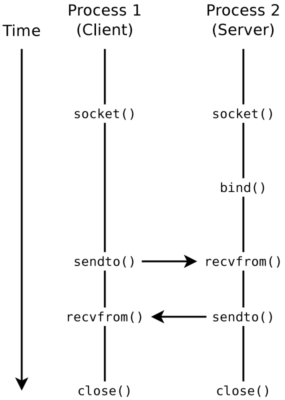
- **Connection-less sockets** (= **datagram sockets**)
  - Use the Transport Layer protocol UDP
  - Advantage: Better data rate as with TCP
    - Reason: Lesser overhead for the protocol
  - Drawback: Segments may arrive in wrong sequence or may get lost
- **Connection-oriented sockets** (= **stream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

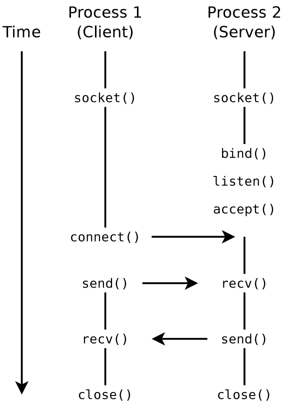
Overview of the sockets in Linux/UNIX: `netstat -n` or `lsof | grep socket`

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);
```

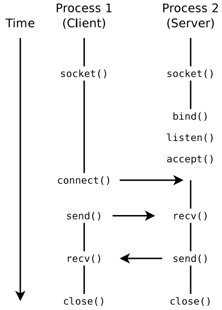
- A call of `socket()` returns an integer value
  - The value is called **socket descriptor** (*socket file descriptor*)
- `domain`: Specifies the protocol family
  - `PF_UNIX`: Local inter-process communication in Linux/UNIX
  - `PF_INET`: IPv4
  - `PF_INET6`: IPv6
- `type`: Specifies the type of the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

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

Bind Address and Port Number: bind

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

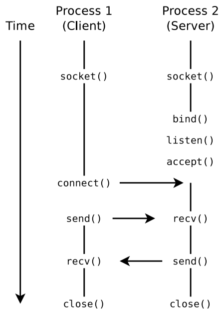
- bind() binds the newly created socket (sd) to the address (address) of the server
  - sd is the socket descriptor from the previous call of socket()
  - address is a data structure, which contains the IP address of the server and a port number
  - addrlen is the length of the data structure, which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



Make a Server ready to receive Data: listen

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

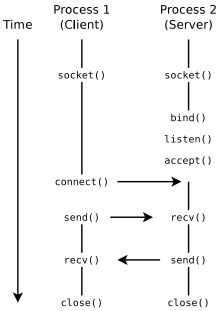
- `listen()` specifies how many connection requests can be buffered by the socket
  - If the `listen()` queue has no more free capacity, further connection requests from clients are rejected
  - `sd` is the socket descriptor from the previous call of `socket()`
  - `backlog` contains the number of possible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



Accept a Connection Request: accept

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

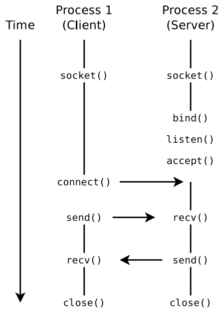
- `accept()` is used by the server to fetch the first connection request from the queue
- The return value is the socket descriptor of the new socket
- If the queue contains no connection requests, the process is blocked until a connection request arrives
- `address` contains the address of the client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



Establish a Connection by the Client

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

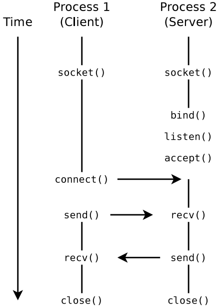
- Via connect(), the client tries to establish a connection to a server socket
- The client must know the address (hostname and port number) of the server
- sd is the socket descriptor
- address contains the address of the server
- addrlen is the length of the data structure, which contains the address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



Connection-oriented Exchange of Data: send and recv

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

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



Connection-oriented Exchange of Data: read and write

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

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

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

Connection-less Exchange of Data: sendto and recvfrom

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

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

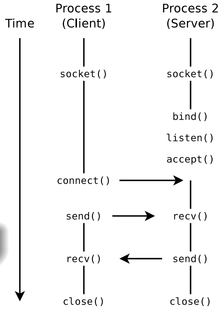
Close a Socket: close

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

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

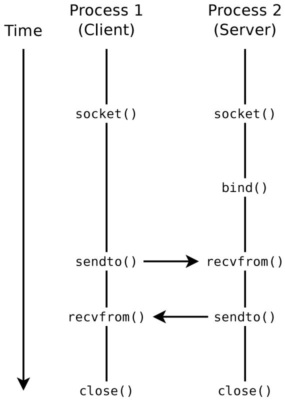
```
int close(int sd);
```

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



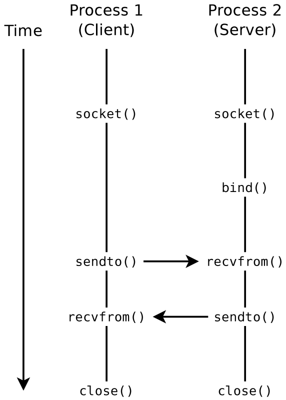
Sockets via UDP – Example (Server)

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

import socket                        # Import module socket

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

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

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



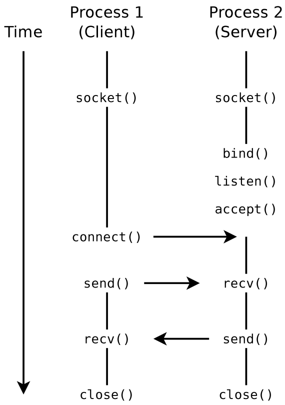
Sockets via UDP – Example (Client)

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

import socket                        # Import module socket

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

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

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

sd.close()                           # Close socket
```



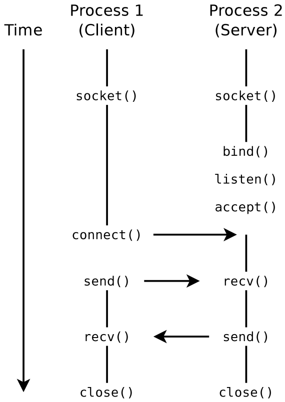
Sockets via TCP – Example (Server)

```
#!/usr/bin/env python
# Echo Server via TCP

import socket                 # Import module socket

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

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

sd.bind(HOST, PORT)           # Bind socket to port

sd.listen(1)                  # Make socket ready to receive
                              # Max. number of connections = 1

conn, addr = sd.accept()      # Socket accepts connections

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

conn.close()                  # Close socket
```



Sockets via TCP – Example (Client)

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

import socket                    # Import module socket

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

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

sd.connect(HOST, PORT)           # Connect with server socket

sd.send('Hello, world')          # Send data

data = sd.recv(1024)             # Receive data

sd.close()                       # Close socket

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



Blocking and non-blocking Sockets

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

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

Non-blocking Sockets - some Impacts

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

Comparison of Communication Systems

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

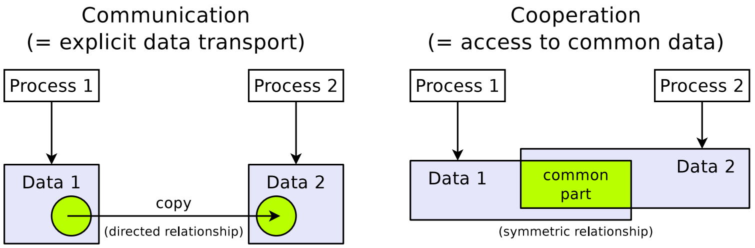
- Advantages of message-based communication versus memory-based communication:
  - The operating system takes care about the synchronization of accesses $\Longrightarrow$ comfortable because the user processes do not need to take care about the synchronization
  - Can be used in distributed systems without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

Storage can be integrated via network connections

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

Cooperation

- Cooperation
  - Semaphor
  - Mutex



Semaphore

- In order to protect (lock) critical sections, not only the already discussed locks can be used, but also **semaphores**
- 1965: Published by Edsger W. Dijkstra
- A semaphore is a counter lock **S** with operations **P(S)** and **V(S)**
  - **V** comes from the dutch *verhogen* = raise
  - **P** comes from the dutch *proberen* = try (to reduce)
- The **access operations are atomic** $\Longrightarrow$ can not be interrupted (indivisible)
- May also permit multiple processes accessing the critical section
  - In contrast to semaphores, can locks only be used to permit a single process entering the critical section at the same time

Cooperating sequential processes. *Edsger W. Dijkstra* (1965)

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

Semaphore: Functioning

- This scenario explains the **functioning**:
    - In front of a shop is a stack of shopping baskets
    - If a customer wants to enter the store, he must take a basket from the stack
    - If a customer has finished shopping, he must place back his basket on the stack
    - If the stack is empty ($\Longrightarrow$ all shopping baskets have been taken be customers), no new customers can enter the store as long as a basket becomes available and is placed on the stack

A Semaphore consists of 2 Data Structures

- `COUNT`: An integer, non-negative counter variable
  - Specifies how many processes are allowed to pass the semaphore currently without getting blocked

The value corresponds, according to the introductory example, with the number of shopping baskets, which are currently placed on the stack in front of the shop

- 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

The semaphore allows to access the critical section when baskets are available again

The length of the queue matches the number of customers, which wait in front of the store because no more baskets are available

3 Access Operations are possible (1/3)

- **Initialization**: First, a new semaphore is created or an existing one is opened
  - For a new semaphore, the count variable is initialized at the beginning with a non-negative initial value

This value is the number of baskets, which are in the queue in front of the store before opening

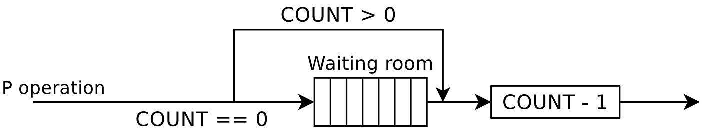
```
// apply the INIT operation on semaphore SEM
SEM.INIT(unsigned int init_value) {

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

3 Access Operations are possible (2/3)

Image Source: Carsten Vogt

- **P operation** (*reduce*): It checks the value of the counter variable
  - If the value is 0, the process becomes blocked
    - *The customer must wait in the waiting queue in front of the shop*
  - If the value > 0, it is reduced by 1
    - *The customer takes a basket*

```
SEM.P() {
    // if the counter variable = 0, the process becomes blocked
    if (SEM.COUNT == 0)
        < block >

    // if the counter variable is > 0, the counter variable
    // is decremented immediately by 1
    SEM.COUNT = SEM.COUNT - 1;
}
```

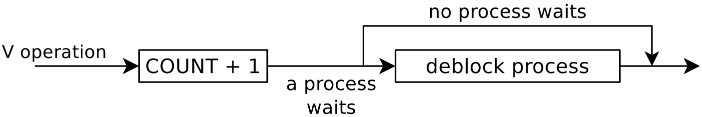


3 Access Operations are possible (3/3)

Image Source: Carsten Vogt

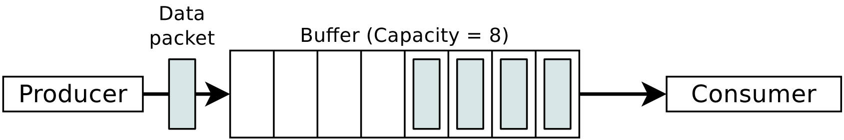
- **V operation** (*raise*): It first increases the counter variable by value 1
  - *A basket is placed back on the stack*
  - If processes are in the waiting room, one process gets deblocked
    - *A customer can now take a basket*
  - The process, which just got deblocked, continues its P operation and first reduces the counter variable
    - *The customer takes a basket*

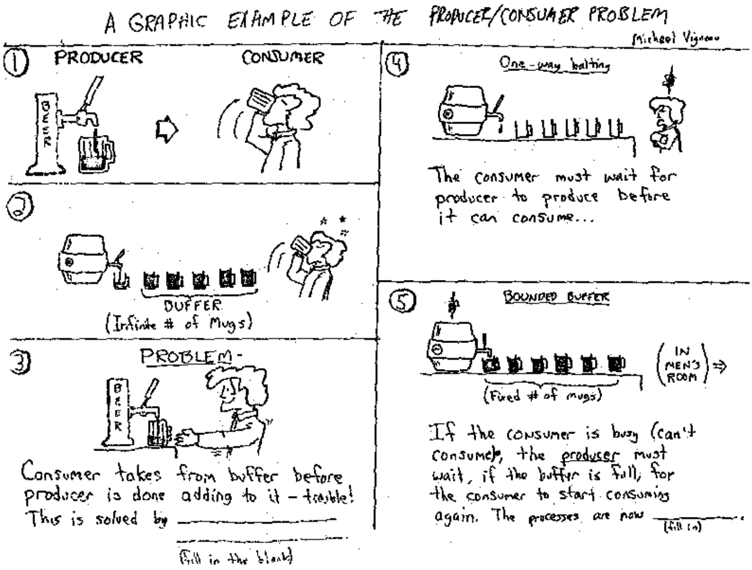
```
SEM.V() {
    // counter variable = counter variable + 1
    SEM.COUNT = SEM.COUNT + 1;

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



Producer/Consumer Example (1/3)

- A producer sends data to a consumer
- A buffer with limited capacity is used to minimize the waiting times of the consumer
- Data is placed into the buffer by the producer and the consumer removes data from the buffer
- Mutual exclusion is necessary in order to avoid inconsistencies
- Buffer = completely filled $\Longrightarrow$ producer must be blocked
- Buffer = empty $\Longrightarrow$ consumer must be blocked





Source: http://www.ccs.neu.edu/home/kenb/synchronize.html

Producer/Consumer Example (2/3)

- 3 semaphores are used for the synchronization of the accesses:
  - `empty`
  - `filled`
  - `mutex`
- The semaphores `filled` and `empty` are used in opposite to each other
  - `empty` counts the number of empty locations in the buffer and its value is reduced by the producer (P operation) and raised by the consumer (V operation)
    - `empty` $= 0 \Longrightarrow$ puffer is completely filled $\Longrightarrow$ producer is blocked
  - `filled` counts the number of data packets (occupied locations) in the buffer and its value is raised by the producer (V operation) and reduced by the consumer (P operation)
    - `filled` $= 0 \Longrightarrow$ puffer is empty $\Longrightarrow$ consumer is blocked
- The semaphore `mutex` is used to ensure for the mutual exclusion

Producer/Consumer Example (3/3)

```
typedef int semaphore;           // semaphores are of type integer
semaphore filled = 0;            // counts the number of occupied locations in the buffer
semaphore empty  = 8;            // counts the number of empty locations in the buffer
semaphore mutex  = 1;            // controls access to the critial sections

void producer (void) {
  int data;

  while (TRUE) {                 // infinite loop
    createDatapacket(data);      // create data packet
    P(empty);                    // decrement the empty locations counter
    P(mutex);                    // enter the critical section
    insertDatapacket(data);      // write data packet into the buffer
    V(mutex);                    // leave the critical section
    V(filled);                   // increment the occupied locations counter
  }
}

void consumer (void) {
  int data;

  while (TRUE) {                 // infinite loop
    P(filled);                   // decrement the occupied locations counter
    P(mutex);                    // enter the critical section
    removeDatapacket(data);      // pick data packet from the buffer
    V(mutex);                    // leave the critical section
    V(empty);                    // increment the empty locations counter
    consumeDatapacket(data);     // consume data packet
  }
}
```

Semaphore Example: PingPong

```
// Initialization of semaphores
s_init (Sema_Ping, 1);
s_init (Sema_Pong, 0);

task Ping is
begin
    loop
        P(Sema_Ping);
        print("Ping");
        V(Sema_Pong);
    end loop;
end Ping;

task Pong is
begin
    loop
        P(Sema_Pong);
        print("Pong, ");
        V(Sema_Ping);
    end loop;
end Pong;
```

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

Semaphore Example: 3 Runners (1/3)

- 3 runners should run a certain distance one behind the other
  - The 2^{nd} runner is not allowed to start before the 1^{nd} runner finished his run
  - The 3^{th} runner is not allowed to start before the 2^{nd} runner finished his run
- Is this solution correct?

```
// Initialization of semaphores
s_init (Sema, 0);

task First is
        < run >
        V(Sema);

task Second is
        P(Sema);
        < run >
        V(Sema);

task Third is
        P(Sema);
        < run >
```

Semaphore Example: 3 Runners (2/3)

- The solution is not correct!
- 2 sequence conditions exist:
  - Runner 1 prior runner 2
  - Runner 2 prior runner 3
- Both sequence conditions use the same semaphore
  - It can happen that runner 3 prior runner 2 decreases with its P operation the semaphore by value 1
- How could a correct solution look like?

```
// Initialization of semaphores
s_init (Sema, 0);

task First is
        < run >
        V(Sema);

task Second is
        P(Sema);
        < run >
        V(Sema);

task Third is
        P(Sema);
        < run >
```

Semaphore Example: 3 Runners (3/3)

- Possible solution:
  - Introduce a second semaphore
  - The second semaphore is also initialized with value 0
  - Runner 2 increases the second semaphore with its V operation
  - Runner 3 decreases the second semaphore with its P operation

```
// Initialization of semaphores
s_init (Sema1, 0);
s_init (Sema2, 0);

task First is
        < run >
        V(Sema1);

task Second is
        P(Sema1);
        < run >
        V(Sema2);

task Third is
        P(Sema2);
        < run >
```

Binary Semaphore

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

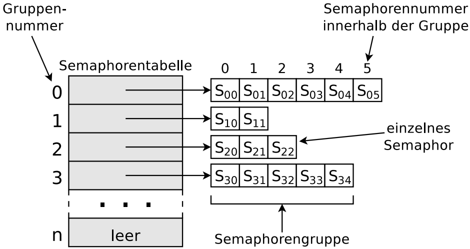
Strong and weak Semaphores

- For each semaphore or binary semaphore, a waiting queue exists, which stores the waiting processes
  - **Strong semaphores**
    - Processes are fetched in FIFO order from the queue
    - Typical sort of the semaphore, which is provided by operating systems
    - Advantage: Starvation is impossible
  - **Weak semaphores** do not set the order, in which the processes are fetched from the queue
    - Used for real-time operation, because there the deblocking of processes bases on their priority and not on the time when they became blocked

Semaphores in Linux/UNIX (1/2)

Image Source: Carsten Vogt

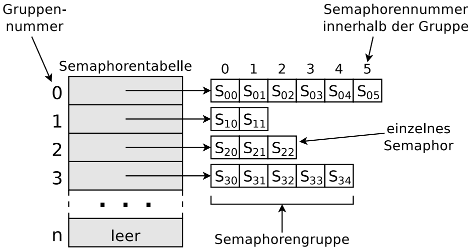
- The semaphore concept of Linux/UNIX differs from the semaphore concept of Dijkstra
  - In Linux/UNIX, the counter variable can be raised or reduced with a P or V operation by more than value 1
  - Multiple access operations on different semaphores can be carried out in an atomic way, which means that they are indivisible
    - Multiple P operations can, for example, be combined and they are only carried out, if none of the P operations causes a blocking
- Linux/UNIX systems maintain in the kernel a semaphore table, which contains references to arrays of semaphores
  - Each array contains a group of semaphores, which is identified by the index of the table



Semaphores in Linux/UNIX (2/2)

Image Source: Carsten Vogt

- Individual semaphores are addressed using the table index and the position in the group (starting from 0)
- Atomic operations on multiple semaphores can only be carried out when all semaphores belong to the same group



Linux/UNIX operating systems provide 3 system calls for working with semaphores

- `semget()`: Create new semaphore or a group of semaphores or open an existing semaphore
- `semctl()`: Request or modify the value of an existing semaphore value 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

- If the option of a semaphore count is not required, a simplified semaphore version, the mutex can be used instead
  - **Mutexes** (derived from **Mut**ual **Ex**clusion) are used to protect critical sections, which are allowed to be accessed by only **a single process** at any given moment
    - Mutexes can only have 2 states: **occupied** and **not occupied**
    - Mutexes have the same functionality as **binary semaphores**

2 functions for accessing a Mutex exist

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

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

Monitor and erase IPC Objects

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

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

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