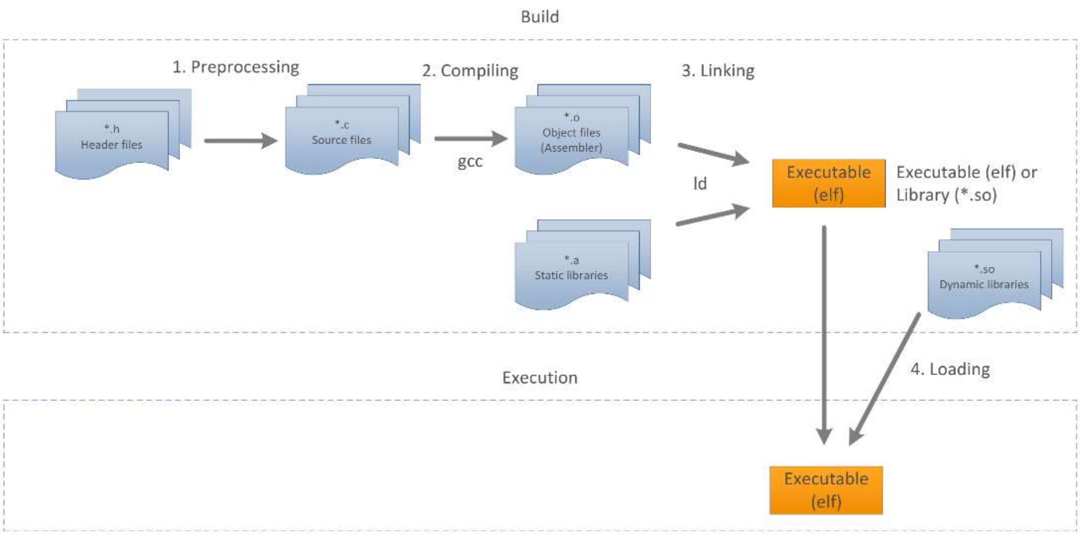
Betriebssysteme

Kapitel 2 – Build:

One Step:

- Build and execute: 'gcc -o hello_world main.c'
- Compile + Link into hello_world

Build Process:

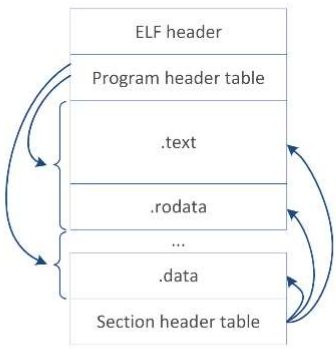


Separate Steps:

- 'gcc -c main.c' → compile main.c into main.o
- 'gcc -o hello_world main.o' → link main.o and deps into hello_world

ELF – Executable and Linking Format:

- Common commands:
  - 'strings hello_world': List all printable Strings in a binary file
  - 'ldd hello_world': List all shared libraries on which the object binary depends
  - 'nm hello_world': List all symbols from object file
  - 'strip hello_world': Delete the symbol table information
  - 'objdump':
    - '-t': Display symbols
    - '-d': Display disassembly
  - 'readelf -a hello_world': Display information about an ELF object file



Makefile:

- Build with: 'make'

Makefile example (1)

Makefile

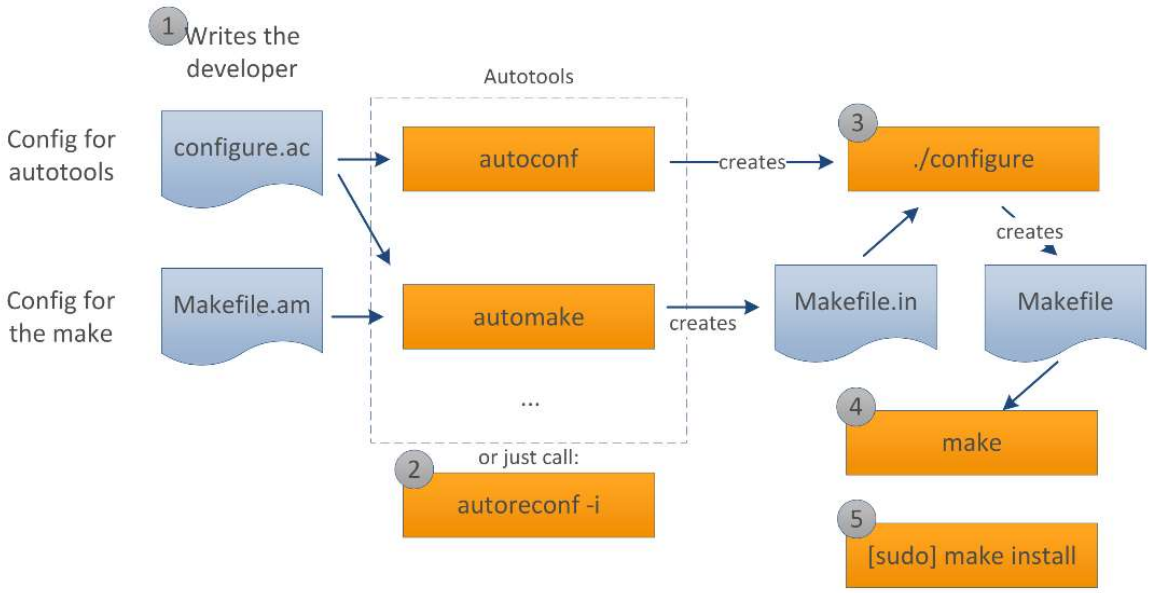
```
#target for the whole program
simple_prog: main.o mathfunctions.o
    gcc -o simple_prog main.o mathfunctions.o

#target for the main file
main.o: main.c
    gcc -c main.c -D USE_SPECIAL_ADD

#target for the mathfunctions file
mathfunctions.o: mathfunctions.c mathfunctions.h
    gcc -c mathfunctions.c

## syntax:
#target: depends_on_file_or_target
#    command

#Behavior: if depends_on has changed the command is executed
```

Build:

`make`

Autotools:

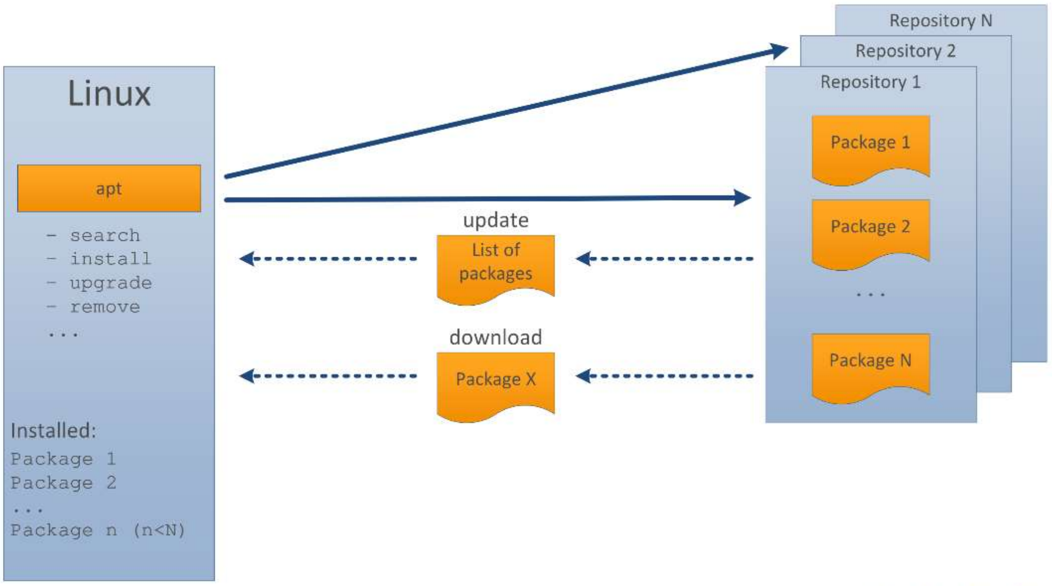
- Usage:
  - 'make -l' → make
  - 'sudo make install' → install
  - 'sudo make uninstall' → uninstall
  - 'make clean' → clean



Kapitel 3 – Package Management:

Centralized pack.man. with apt:

- Work with centralised (remote) repos and automatically download packages

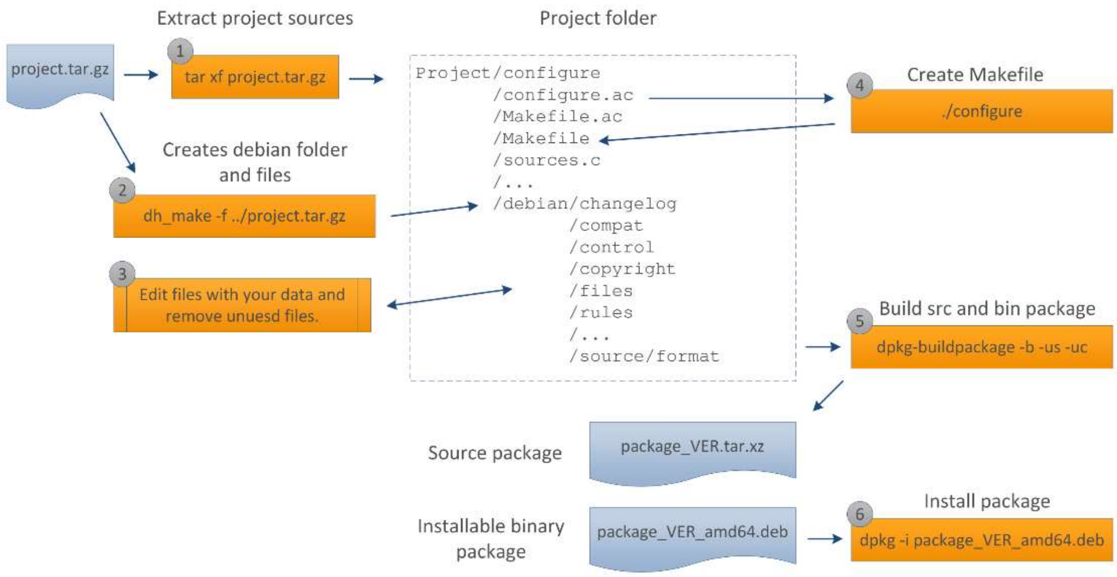


- Commands: 'apt '
    - Search: Searches for packages in the package repos
    - List: Lists packages with criteria (installed, upgradable)
    - Show: Shows information about packages
    - Depends: show dependencies of packages
    - Install: Install package
    - Remove: Remove the package
    - Update: Update the list of available package
    - Upgrade: Upgrades installed packages to the newest version
    - Contains: Find out which package a file is contained

Local pack.man with dpkg:

- Work with local packages on the system
- Commands: 'dpkg'
    - -S: Searches for files in packages
    - -l: Lists installed packages
    - -i: installs packages from file
    - -r: Removes installed packages
    - -L: Lists files in installed packages
    - --info: Shows information about packages
    - --contents: Lists files contained in packages
    - -reconfigure: Repairs installed packages
    - -buildpackage: Builds debian packages

Build Packages:

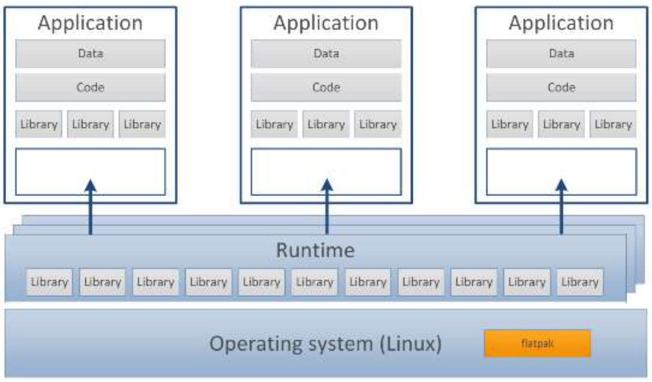


Flatpak:

- Packages for other Linux distros and with different release cycles of your distro?
- Commands: ‘flatpak’
  - Search: Searches for packages on flathub
  - List: Lists installed packages
  - Install: Installs package
  - Uninstall: Uninstalls package
  - Update: Updates installd packages
  - Info: Shows information
  - Run: Runs an installed application
  - -builder: Set of commands to build a package

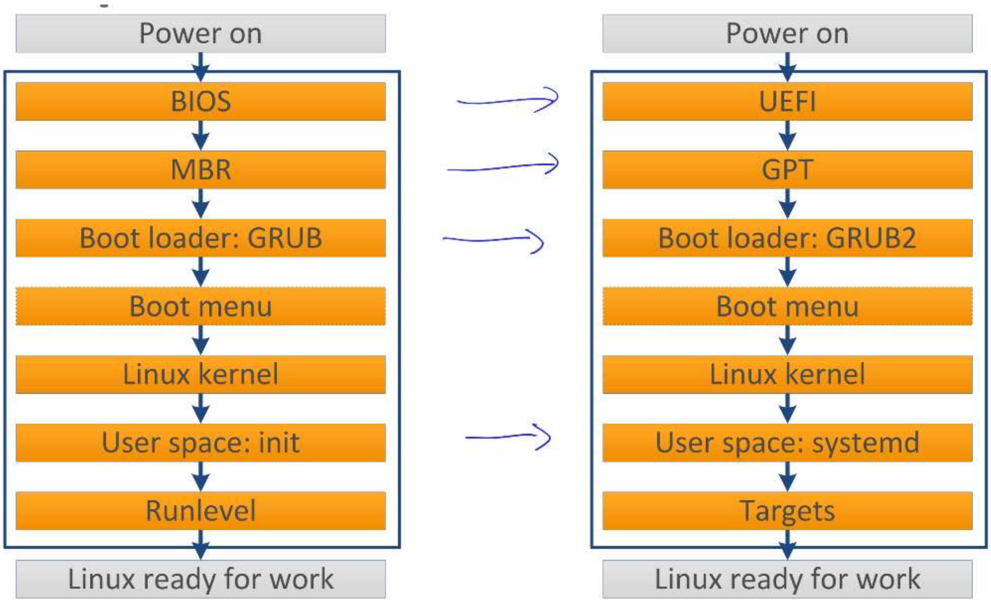
Flatpak Applications:

- Build for all Linux distros
- Runtimes
- Bundled Libraries
- Sandboxes
- Portals
- Repositories → Easy search/update



Kapitel 4 – OS Boot Architecture:

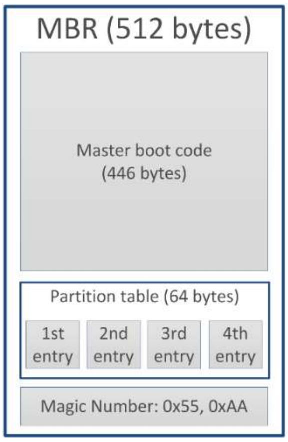
Boot Procedure - BIOS vs UEFI:



UEFI – Unified Extended Firmware Interface:

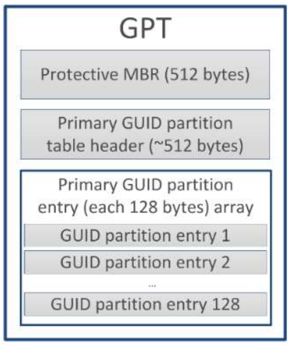
- Graphical User Interface (with mouse)
- Fast boot: cache + hibernation (Win only)
- Secure boot:
  - Protection against malware
  - Prevents against execution of unsigned code
- Network boot
- Modular interface for applications and devices
- Supported modes:
  - UEFI mode: Requires an EFI partition on boot device
  - BIOS mode: Old way of booting

Master Boot Record – MBR:



- Up to 2 TiB disks and partitions
- No safety (no checksum)
- Supports 4 primary partitions
- Supports one extended partition (not bootable)

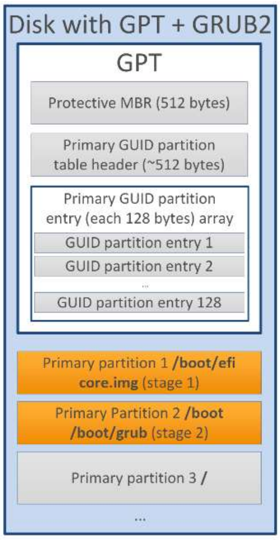
GUID Partition Table – GPT:



- Up to 18 EiB disks and partitions
- Safety (→ Checksums!)
- Version number
- Supports 128 primary partitions
- Does not have a boot code
- UEFI boots from an EFI partition

→ GPT > MBR !

Grand Unified Bootloader 2 – GRUB2:



- Boots in stages:
  - Stage1: Loads directly stage 2, usually from /boot partition
  - Stage2: Loads the default config file and other modules needed
- Themes graphical menus, scripting support
- Uses UUID to identify disks
- Automated search for other OS (like Win)
- Supports LLVM and RAID
- Boots live CD images from hard drive

User space - systemd:

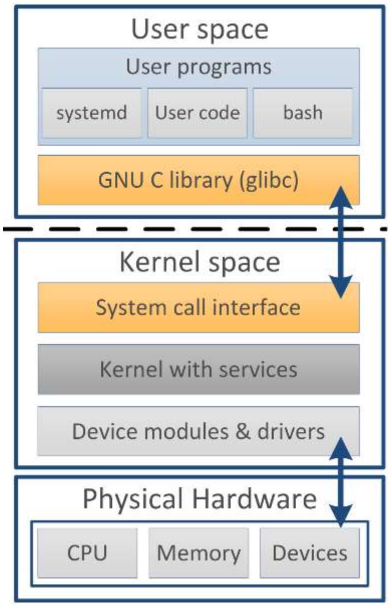
- First process started by the kernel
- Has always PID 1
- Looks in /etc/system/system/default.target for default target and executes it
- Starts user space processes on boot:
  - Daemons
  - Terminals
  - Graphical Desktop
- Speed up boot: Starts processes in parallel
- Commands:
  - Service daemon start
  - Service daemon stop
  - Service daemon reload
  - Service daemon restart
  - Service daemon status
- Enable/disable Daemon: systemctl enable/disable daemon.service

Linux High Level Overview:

OS Tasks:

- Execute graphical user applications
- Provide desktop environment
- Draw windows
- Provide shells
- Manage resources
- Support, abstract and virtualize hardware

User vs Kernel Space:



User space:

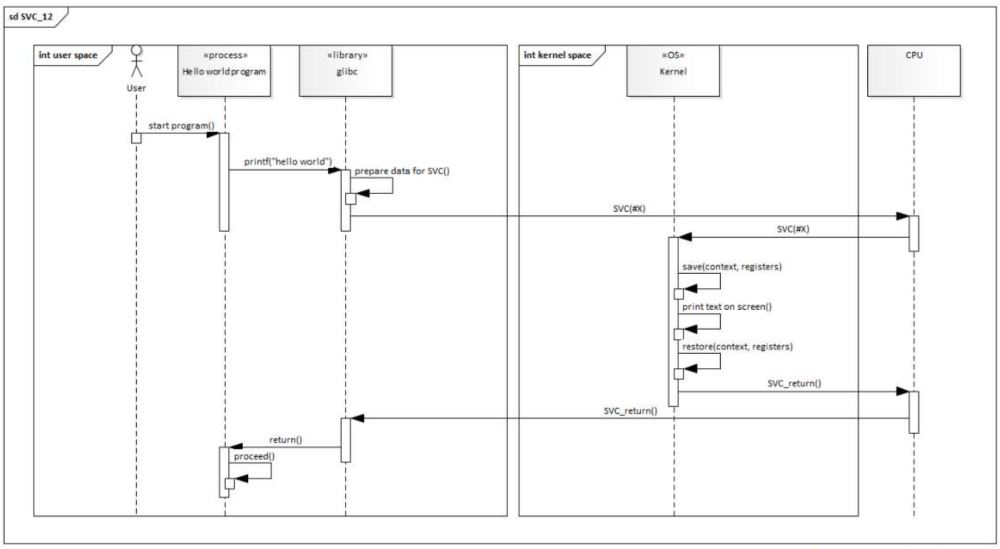
- All the code outside the kernel
- Restricted access to the hardware
- Only a subset of CPU instructions
- Crash in user process: Only stops the process

Kernel space:

- Complete and unrestricted access to the hardware
- Can execute and CPU instruction
- Crashes in kernel catastrophic: system stop!

Supervisor Call (SVC):

- CPU instruction to give control to the OS/kernel
- Requests for an OS service:
  - Start process
  - Allocate Memory
  - File open/read/…
  - Send data over network
  - …
- Example:

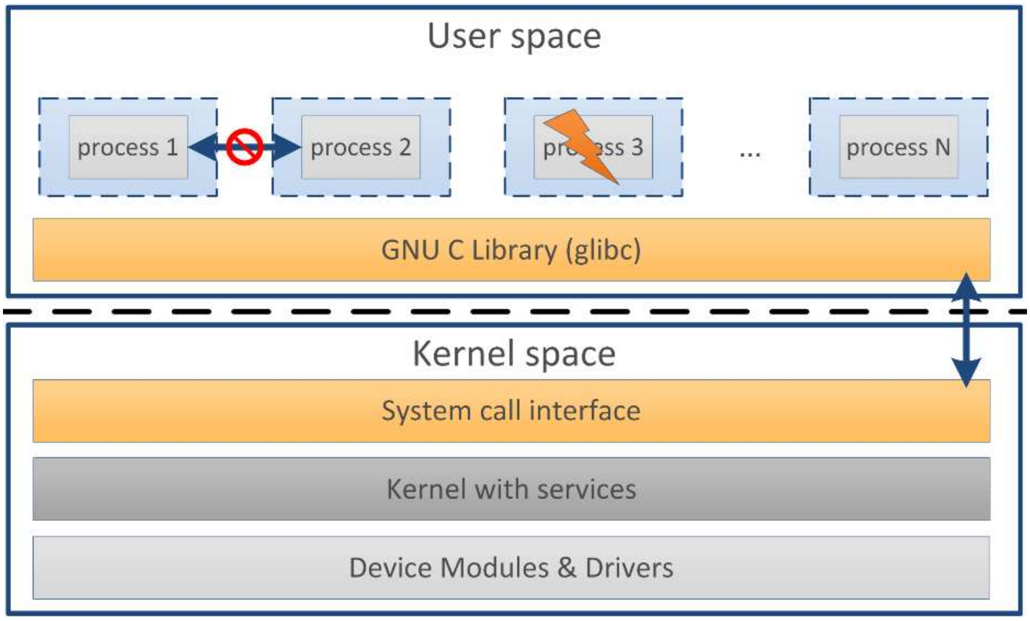


Kapitel 5 – Processes:

Process:

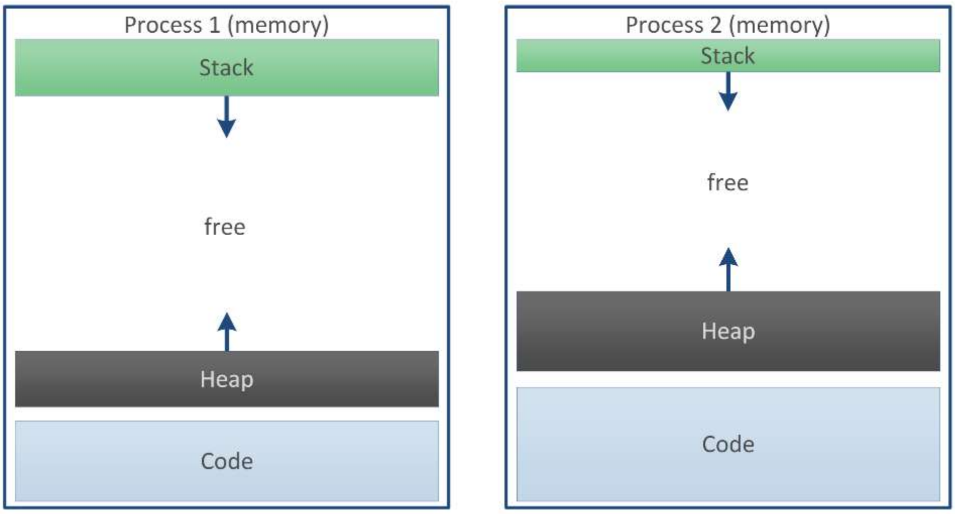
= Instance of a computer program that is being executed

Process isolation:

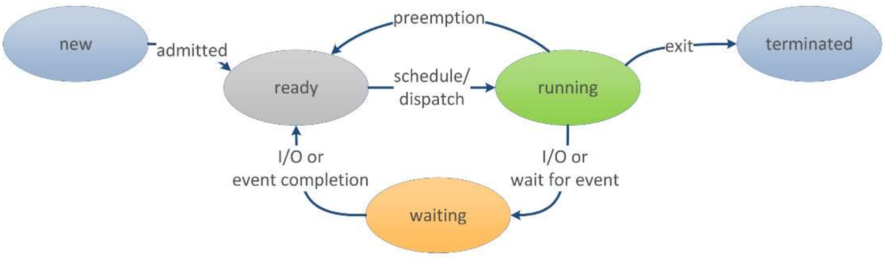


- Process cannot interfere with another process
- If one process stops the others will not be stopped by that

Process memory view:



Process states:



Daemon – a special kind of process:

- Daemon is a process
- Operates in the background
- A daemons parent PID is 1 → systemd
- Usually started from system as part of the boot procedure
- No direct interaction with the user
- Communication with a daemon: Network, signals, pipes, share memory
- Working directory: '/'
- Usually uses logfile to log events and errors

OS process table:

- The kernel manages the different processes
- Each process has its own process control block (=PCB)
    - Contains all process specific properties
    - The process table contains all PCBs
- In Linux: struct task_struct { … }

Advantages of different processes:

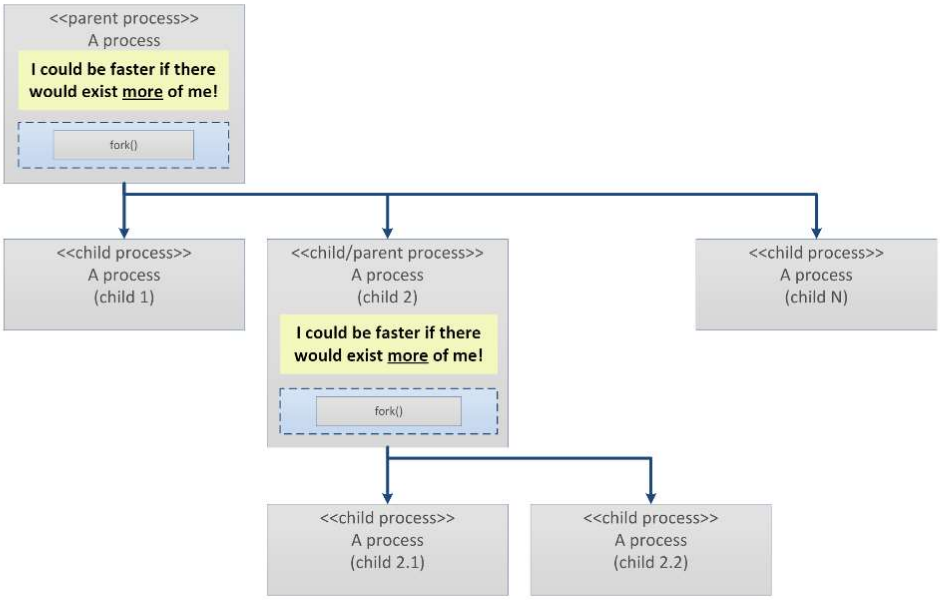
- Independent start of different processes
- Can be executed in parallel
- If a process crashes the others can continue their work
- No overwriting of the memory
- Security: No read of another process memory possible
- Independent development
- Independent dependencies
- Each user can have its own processes

Execute a command:

```
#include <stdio.h>     //printf
#include <stdlib.h>    //EXIT_SUCCESS, system

int main(int argc, char** argv)
{
    //executes a command specified in command by calling /bin/sh -c command
    const char* const command = "ls -l /";
    int exit_status = system(command);

    if(exit_status == -1) {
        printf("%s can't be started.\n", command);
    } else {
        printf("%s exited with status: %d.\n", command, exit_status);
    }

    return EXIT_SUCCESS;
}
```

system (man): http://man7.org/linux/man-pages/man3/system.3.html

Fork idea:



Example:

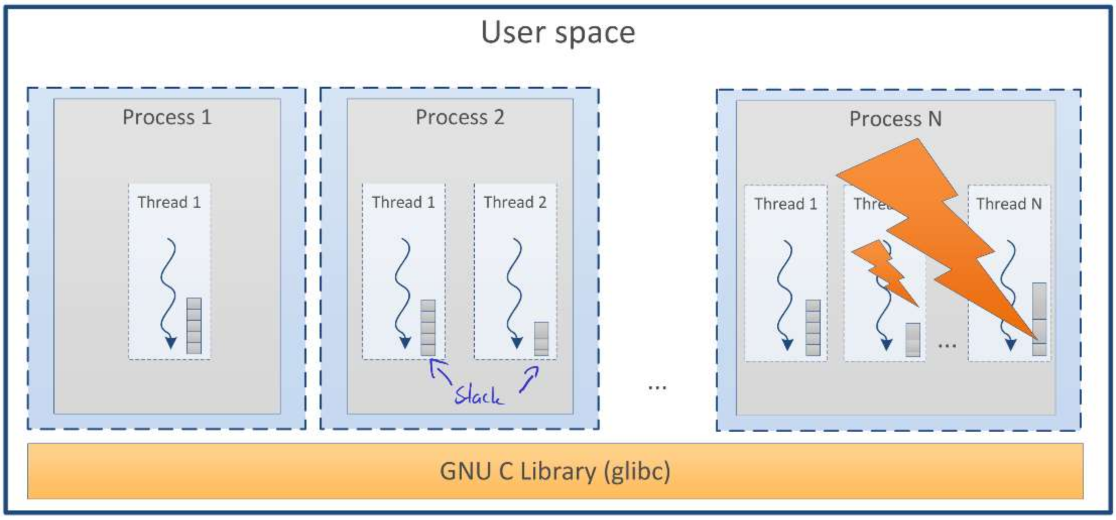
```c
#include <stdio.h>     //printf
#include <stdlib.h>    //EXIT_SUCCESS, EXIT_FAILURE, system
#include <unistd.h>    //fork
#include <sys/wait.h>  //waitpid

int main(int argc, char** argv)
{
    pid_t pid = fork();

    switch(pid){
        case -1: //error
            printf("Error: fork failed.\n");
            exit(EXIT_FAILURE);
            break;
        case 0:  //child
            printf("Hi, I'm the fork with the PID %d!\n", getpid());
            break;
        default: //parent
            printf("Parent waits until child process with PID %d ends.\n", pid);
            waitpid(pid, NULL, 0);
            printf("Child process with PID %d exited.\n", pid);
            break;
    }

    return EXIT_SUCCESS;
}
```

Process management on shell:

- './command': Start a process
- Kill: Stop (exit) a process
- Wait: Wait until a child process has stopped
- Ps aux: show information about the started process
- Top: Show live information about processes
- Pstree: Show the process hierarchy
- Renice: Change the priority of a process

In Linux: A thread is a lightweight process!

Thread illustration:



Properties of a thread:

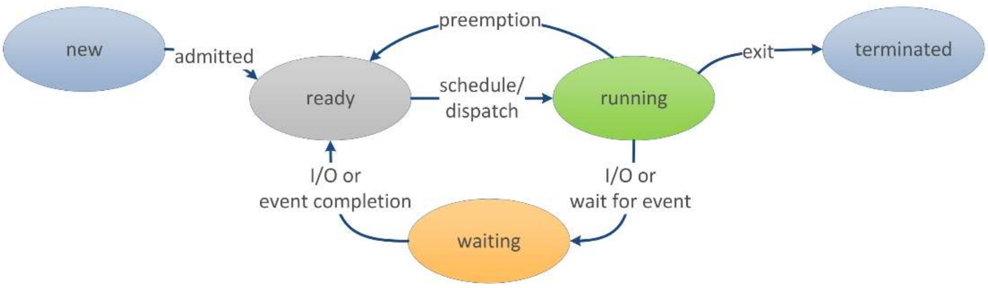
- Thread id (TID)
- State
- Register entries
- Stack (pointer)

Shared process properties:

- Address space
- Global variables
- Opened files
- Child processes
- Signals
- Working directory
- Environment variables

The kernel manages different threads

Thread states:



<u>Kapitel 6 – Synchronization 1:</u>

Problem with parallelization:

- Cause:
  - Parallel read/write
  - Parallel use
- Problem:
  - Read of unfinished data
  - (Partial) overwrite of data
- May occur sporadic: looks like undefined behavior
- These kind of bugs are often very hard to find

It is called race condition (Konkurrenzbedingung)

- Solution: Mutual exclusion “Gegenseitiger Ausschluss“
  - Only one process can access the critical section
  - Others must wait

Idea 1: Lock variables:

```c
int global_counter = 0;
int global_lock = 0;

void* thread1() {
  while(1) {
    while(lock == 1) {} //busy wait
    //thread1 see: lock==0
    //!! INTERRUPT: activate thread2 !!
    lock = 1;
    //...
  }
}
```

```c
void* thread2() {
  while(1) {
    while(lock == 1) {} //busy wait

    lock = 1;
      //increase counter
      int counter = global_counter;
      //!! INTERRUPT: activate thread1 !!
  }
}
```

Problem: Both threads are in a critical section. Solution useless.

Idea 2 – Disable interrupts:

- Easy solution, but …
- Only works on single core CPUs
- May disturb scheduling
- May disturb real time behavior
- Some interrupts cannot be deactivated
- A process does not activate interrupts again
- Program error in critical section

Conclusion:

- Only in some parts of the OS kernel possible

```c
int global_counter = 0;

void* thread1() {
  while(1) {
    disable_interrupts();

    //increase counter
    int counter = global_counter;
    counter = counter + 1;
    global_counter = counter;

    enable_interrupts();

    produce_something(counter);
  }
}
```

Semaphores:

Idea: Instead of busy wait, a process/thread blocks until the critical area is free.

Operations:

- 'seminit(s, value)': Creates and inits a semaphore with a value. Number = processes that can simultaneously enter the critical area.
- 'P(s)': Wait until the critical area is free (value--)
- 'V(s): Releases the critical area (value++)

Usage:

Basic usage

name of semaphore

```
seminit(s, 1);

P(s);
//critical area..,
V(s);
```

value : 1 process/thread can simultaneously enter the critical area

Types:

- Mutex ('seminit(s, 1)'): Used for mutual exclusion
- Binary ('seminit(s, 0)'): Used when there is only one shared resource
- Counting ('seminit(s, N)'): Used to handle more than one shared resource, possible with 0/N

Role of the OS:

- Provides semaphores
- Ensures that the P()/V() operations are atomic

…can reach this because:

- Disable process/thread changes
- Disable interrupts(temporarily)
- Use of a test-and-set CPU instruction

Example implementation:

Pseudo C code*

```
//Semaphore struct with a value and
//an internal list of waiting
//processes/threads
struct Semaphore
{
  int value;
  struct ProcessList process_list;
};
//initialises a semaphore with a value
void seminit(struct Semaphore* s, int value)
{
  s->value=value;
}
```

```
void P(struct Semaphore* s)
{
  if (s->value > 0) {
    s->value--;
  } else {
    append_to(pid, s->process_list);
    sleep(); //sleep indefinitely
  }
}
void V(struct Semaphore* s)
{
  if (is_empty(s->process_list)) {
    s->value++;
  } else {
    int pid=pop_any(s->process_list);
    wakeup(pid);
  }
}
```

Example:

Example with three threads and an critical area where 2 threads can enter simultaneously:

Step	Thread	Operation	Semaphore value	Comment
0		seminit(s, 2)	2	semaphore is initialised with 2
1	thread 1	P(s)	1	thread 1 can enter the critical area
2	thread 2	P(s)	0	thread 2 can enter the critical area
3	thread 3	P(s)	0	thread 3 has to wait
4	thread 2	V(s)	0	thread 2 leaves the critical area
5	thread 3		0	thread 3 wakes up and enters the critical area
6	thread 3	V(s)	1	thread 3 leaves the critical area
7	thread 1	V(s)	2	thread 1 leaves the critical area

Mutual exclusion:

```
int global_counter = 0;
seminit(s, 1); //declare and initialise semaphore

void* thread1() {
  while(1) {
    P(s);
      //increase counter
      int counter = global_counter;
      counter = counter + 1;
      global_counter = counter;
    V(s);

    produce_something(counter);
  }
}

void* thread2() {
   while(1) {
     P(s);
       //increase counter
       int counter = global_counter;
       counter = counter + 1;
       global_counter = counter;
     V(s);

     produce_something(counter);
   }
}

int main() {
    //start threads...
}
```

Mutual exclusion: Example C code:

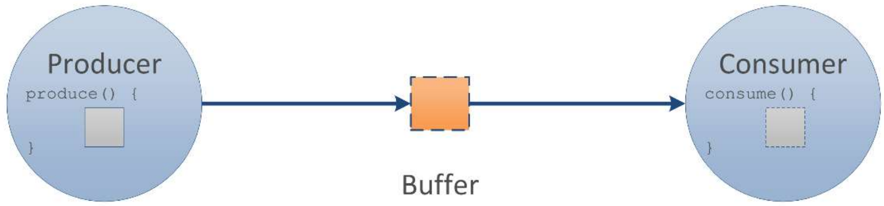
Mutual exclusion with lock files:

- Use a file to simulate P()/V() operations
- The process/thread that can acquire the file lock can enter the critical section

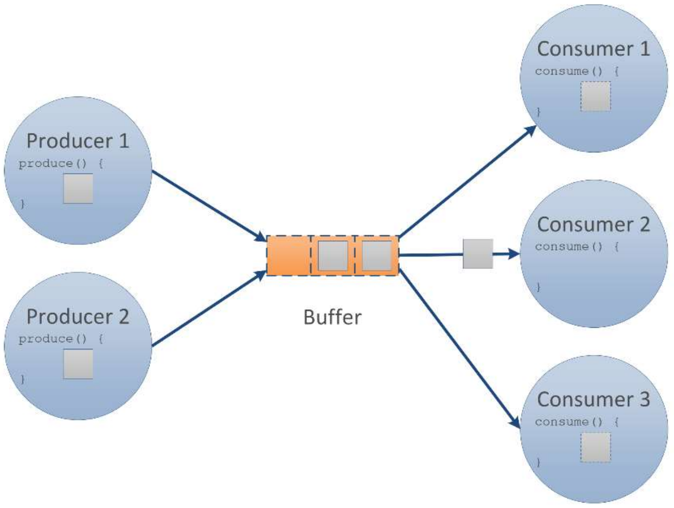
Named semaphores can be found on: ‘/dev/shm’

Kapitel 7 – Synchronisation 2:

Producer-Consumer problem:



- One or more processes produce something
- One or more processes consume something
- There is a buffer with one place to store the produces "artefact"
- Producer delivers to artifact and produces next, until artefact is full, then it waits until the artefact is completely free
- Consumer fetches the artefact until it is empty, then they wait until it is full again



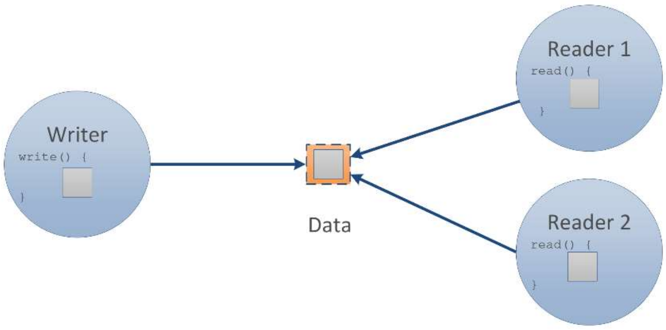
Here:

- Multiple producers and multiple consumers
- Producer delivers artefact until it is full, then it waits until the buffer has a free place
- Consumer fetches from artefact. If artefact is empty, wait until buffer contains at least one again

Reader-Writer problem:

- One or more writers "writes" something
- One or more Readers "reads" something
- Shared area for data
- Writer:
  - After data is written, a writer can immediately collect next set of data
  - If no readers currently read, it can write the new set of data
  - If readers currently read, it waits until all readers have finished reding
- Reader:

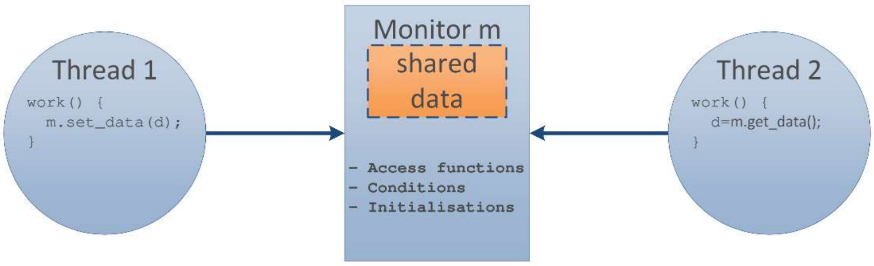
- After the data is fully read, it can work independently with the data
- If a writer is currently writing the readers must wait!
- It is not a consuming read, the data stay in the shared data area



Monitor Concept:

Problems with "pure" semaphores:

- Difficult implementation
- Depends on the correctness of all processes/threads
- Verification of correctness is difficult
- Difficult to determine which access functions read or change shared data
- Data is independent of access functions



Monitor M:

- Contains data and access functions
- Does all the initialization of data
- Checks the conditions internally
- Access to the shared data is only possible via the access functions
- Only one "active" process/thread can be inside a access function
- Pro:
  - Less error prone: Less todo for the users
  - Concentration on the difficult know-how inside the monitor

Mutex and condition:

Idea: A mutex controls the access functions of a monitor. The conditions help to to implement the waiting logic.

Operation	Description
`Mutex mutex`	Creates an instance of a mutex. A mutex is like a binary semaphore. The only difference is, that only the calling process/thread can unlock it.
`Condition cond`	Creates a condition variable. A condition variable is a synchronisation primitive that enables a process/thread to wait until a particular condition occurs.
`lock(mutex)`	Locks a mutex. The others wait.
`unlock(mutex)`	Unlocks a mutex.
`wait(cond, mutex)`	Waits until the condition is fullfilled. The mutex is free while waiting.
`signal(cond)`	Signals that the condition is fullfilled. Notifies one.

Kapitel 8 – Communication 1:

Process Communication:

- The communication channel is provided by the OS
- Different types of communication channels exist

Important concepts:

Important concepts

Function/concept	Description
`send(destination, message)`	**Send** a message **to** the **destination**.
`recv(source, &message)`	**Receive** a message **from** the **source**.
Blocking/synchron	`send()/recv()` **blocks** until the data is fully transferred.
Non-blocking/asynchron	`send()/recv()` **immediately returns** and the process can proceed.
Protocol required	A **protocol** defines the **order of `send()/recv()`** between processes and the message format.
Half-duplex/unidirectional	Communication over a "channel" **only in one** direction.
Full-duplex/bidirectional	Communication over a "channel" **in both** directions.

Signals: Are asynchronous events that interrupt a process. It is like an interrupt request at process level

- Overview:

List of signals: `kill -l`

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- Handling:
    - If a process receives a signal: Signal is saved in the PCB
    - If the process state changes to "running" the process will be interrupted
    - The OS looks if there is a registered handler for the signal
        - If there is a handler, this function will be called
        - IF there is no handler, the default function will be called
    - If the handler has not exited the process, it will proceed
- Shell:

Commands

Command	Description
`kill PID`	Sends the signal **15 (SIGTERM)** to the process.
`kill -1 PID`	Sends the signal **1 (SIGHUP)** to the process.
`kill -SIGHUP PID`	Sends the signal **1 (SIGHUP)** to the process.
`killall process_name`	Sends the signal **15 (SIGTERM)** to the process.
`killall -s HUP process_name`	Sends the signal **15 (SIGTERM)** to the process.

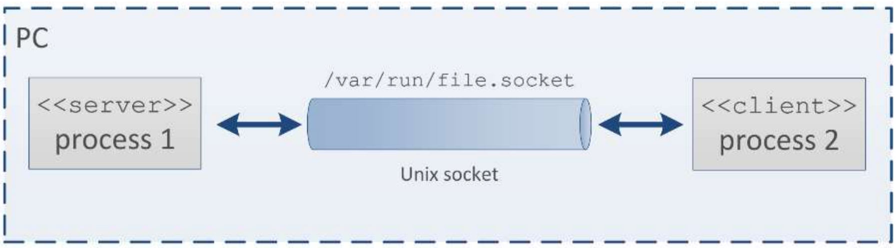
C function overview - Signals:

Function*	Description
`raise(int sig);`	**Sends** a **signal** to the calling process or thread.
`kill(pid_t pid, int sig);`	**Sends** a **signal** to the process with the specified pid.
`pause(void);`	Causes the calling process or thread to **sleep until a signal is delivered**.
`sleep(unsigned int seconds);`	**Sleeps** for the **specified seconds** or **until a signal delivered**.
`alarm(unsigned int seconds);`	**Sends an alarm** to the calling process or thread in the specified seconds.
`signal(int signum, sighandler_t handler);`	**Registers** a **signal handler** for signum.
`signal(int signum, SIG_IGN);`	**Ignores** signals for signum, by setting a `SIG_IGN` handler, which doesn't exits the process.
`signal(int signum, SIG_DFL);`	**Sets** the **default handler** for signum.

Sockets:

- Endpoint for sending or receiving data
- Inter-process communication (IPC)
- Byte oriented data transfer
- Full duplex → send()/recv() over the same socket

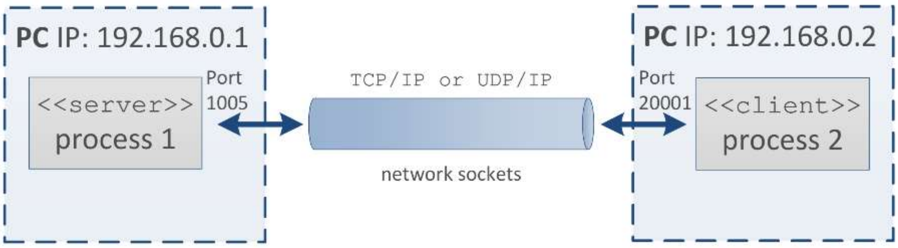
Unix Sockets:



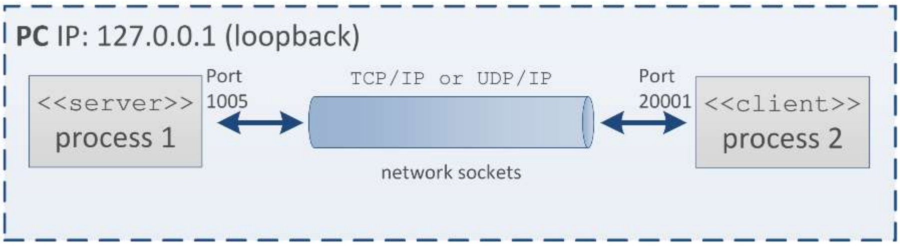
- Unix domain
- Communication only on the same PC
- Is faster than network socket
- Use file system as address name space
- User ID can be determined
- Access control via file system

Network Sockets:

Remote:



Local:

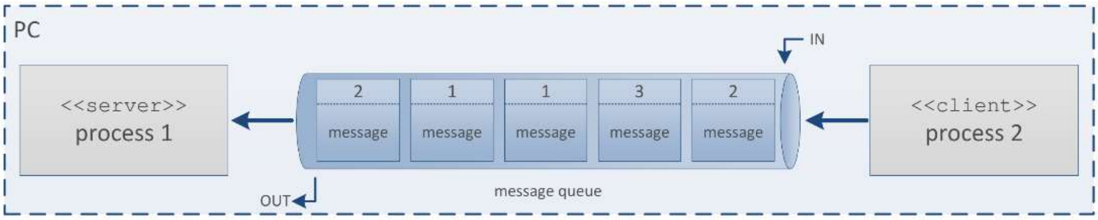


Concept:

- Internet/network domain
- Communication over the network
- Communication on same PC over loopback
- TCP/IP: Connection oriented
- UDP/IP: Simple connection communication
- Access control on packet filter level

Kapitel 9 – Communication 2:

Message Queue:



Concept:

- Queue to store messages
- Inter-process communication (IPC) between processes on one PC
- Messages have priority/type
- Internal stored as a linked list
- Send into queue does not require an active receiver
- Read from queue does not require and active sender
- Max queue size (default 16KiB)
- Max message size (default: 8KiB)

Structure:

```
struct message {
    long priority;    //priority or type
    char message[64]; //buffer for message bytes
};
```

- The lower the number, the higher the priority
- Priority can be interpreted as a type

Message queue: Linux commands

Command	Description
ipcs	Show information on IPC facilities
ipcs -q	Shows active message queues in the system
ipcmk	Make various IPC resources
ipcmk -Q	Create a message queue
ipcrm	Remove certain IPC resources
ipcrm -q 1	Remove message queue with id 1
ipcrm -Q 2	Remove message queue with key 2

Shared Memory:

- Area between processes
- Inter-process communication (IPC) between processes on one PC
- Plain memory area with certain size

- Access needs to be synchronized (e.g. semaphore)
- Access is very fast

Pseudo C Code:

```
seminit(READY_TO_WRITE, 1); //declare and initialise semaphore
seminit(READY_TO_READ,  0); //declare and initialise semaphore

void receiver() {
  //create shared memory
  shmget(...);
  //attach the shared memory
  shared_mem_address = shmat(...);




  //copy data from shared memory
  P(READY_TO_READ);
  copy(data, shared_mem_address); //data = sm
  V(READY_TO_WRITE);

  //... work with data
  work_with(data);

  //detach shared memory
  shmdt(...);
  //remove shared memory
  shmctl(...);
}

void sender() {
  //get existing shared memory
  shmget(...);
  //attach the shared memory
  shared_mem_address = shmat(...);

  //... prepare data
  data = prepare_data();

  //copy data into shared memory
  P(READY_TO_WRITE);
  copy(shared_mem_address, data); //sm = data
  V(READY_TO_READ);




  //detach shared memory
  shmdt(...);


}
```

Linux commands:

Command	Description
`ipcs`	Show information on IPC facilities
`ipcs -m`	Shows active shared memory in the system
`ipcmk`	Make various IPC resources
`ipcmk -M 8`	Create a shared memory with 8 bytes
`ipcrm`	Remove certain IPC resources
`ipcrm -m 1`	Remove shared memory with id 1
`ipcrm -M 2`	Remove shared memory with key 2