

# **Betriebssysteme / Operating Systems**

## **Linux Basics I – Part I:**

### **Commands, Shell-Scripts, Compiler**

WS 2024

Prof. Dr.-Ing. Holger Gräßner

## Printed media (I)

Kernighan B. und Pike R.: Der UNIX-Werkzeugkasten.  
Carl Hanser Verlag, München · Wien, 2002.

Kernighan B. und Ritchie D.: Programmieren in C.  
Carl Hanser Verlag, München · Wien, 1990.  
2. Ausgabe (ANSI C).

Kerrisk M.: The Linux Programming Interface – A Linux and UNIX System Programming Handbook.  
No Starch Press Inc., San Francisco, 2010. ISBN 978-1-59327-220-3.

O'Sullivan B.: Answers to frequently asked questions for comp.os.research.  
<http://www.faqs.org/faqs/os-research>.

Quade J. und Kunst E. K.: Linux-Treiber entwickeln – Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung.  
dpunkt Verlag GmbH, Heidelberg, 2011. ISBN 978-3-89864-696-3.

## Printed media (II)

Stevens W. R. und Rago S.: Advanced Programming in the UNIX Environment.

Addison-Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 2005.

Stevens W. R., Rudoff A. und Fenner B.: Unix Network Programming Volume 1.

Addison-Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 2003.

Tanenbaum A. S.: Moderne Betriebssysteme.

Pearson Studium Prentice Hall Verlag, München, 2003.

Thompson R. B. und Thompson B.F.: PC Hardware in a Nutshell.

O'Reilly & Associates Inc., Sebastopol, 2003.

Williams G. J.: Debian GNU/Linux Desktop Survival Guide.

Togaware Pty Ltd., <http://www.togaware.com/linux/survivor/>, 2008.

## Digital media (I)

Mind the Copyright!

Linux:

- Richard Petersen: “Ubuntu the complete reference”. Available as PDF → seek the internet  
[https://archive.org/details/epdf.pub\\_linux-the-complete-reference](https://archive.org/details/epdf.pub_linux-the-complete-reference) (02.10.2024)
- Linux quick reference card: [https://danleff.net/downloads/linux/linux\\_quick\\_ref\\_card.pdf](https://danleff.net/downloads/linux/linux_quick_ref_card.pdf) (02.10.2024)
- A-Z Index of the Linux command line: bash + utilities: <https://ss64.com/bash/> (02.10.2024)

Linux-Programmierung, Rheinwerk-Verlag (german only):

[http://openbook.rheinwerk-verlag.de/linux\\_unix\\_programmierung](http://openbook.rheinwerk-verlag.de/linux_unix_programmierung) (02.10.2024)

Columbia University: <http://www.columbia.edu> → search for “Linux” (02.10.2024)

## Digital media (II)

Shells:

- Rheinwerk: [http://openbook.rheinwerk-verlag.de/shell\\_programmierung](http://openbook.rheinwerk-verlag.de/shell_programmierung) (02.10.2024, german)
- GNU, bash: <http://www.gnu.org/software/bash/manual/> (02.10.2024)

AWK:

- GNU: <https://www.gnu.org/software/bash/manual/> (02.10.2024)
- Wikibooks, awk-Skripting:  
[https://de.wikibooks.org/wiki/Awk:\\_Grundlagen:\\_Die\\_Grundstruktur\\_eines\\_awk-Skriptes](https://de.wikibooks.org/wiki/Awk:_Grundlagen:_Die_Grundstruktur_eines_awk-Skriptes)  
(02.10.2024, german)

## Digital media (III)

C-Library Reference Manual:

- GNU: <https://sourceware.org/glibc/manual/> (02.20.2024)

C Reference Manual:

- GNU: <https://www.gnu.org/software/gnu-c-manual/> (02.10.2024)

Make:

- GNU: <https://www.gnu.org/software/make/manual/> (02.10.2024)

POSIX:

- GNU: [https://www.gnu.org/software/guile/manual/html\\_node/POSIX.html](https://www.gnu.org/software/guile/manual/html_node/POSIX.html) (02.10.2024)

## Manipulating files and directories I:

<b>pwd</b>	output the name of current directory
<b>cd</b>	change current directory
<b>ls</b>	Ausgabe von Dateiverzeichnissen (Kurzform)...
<b>ls</b>	output of directories (short version)...
<b>ls -l</b>	...with file access rights (U:rwx G:rwx O:rwx)
<b>chmod</b>	change rights of files (U:rwx G:rwx O:rwx)
<b>cp</b>	copy files
<b>rm</b>	delete files
<b>rmdir</b>	delete directories
<b>mkdir</b>	create directories
<b>mv</b>	move or rename files

## Manipulating files and directories II:

<b>awk</b>	search and process patterns
<b>cat</b>	output of files
<b>diff</b>	find differences between two in files or directories
<b>expand</b>	substitute tabs by blanks
<b>find</b>	universal finding tool for files
<b>grep</b>	finding tool for file content
<b>head</b>	output of file begin
<b>kedit, ...</b>	Text editors (there are many of them)
<b>less, more</b>	page related output of files

## Manipulating files and directories III:

**lpr** print files on a printer

**sed** editor for streams

**sort** sort file content

**tail** output the end of a file

**unexpand** substitute blanks by tabs

### Commands for system control:

**htop, ps** output state of processes

**kill** send a signal to process #(PID)  
→ usually, the process will terminate

**login** operating system logon

**logout** operating system logout

**man** output an instruction manual for a command

**nohup** start process independently to terminal → no termination

**renice** change the priority of a running process

**sleep** suspend process for some time

**time** get the execution time of a command

### Other commands:

**git**, ... service for version management

**make** tool for program development

**tee** duplicate input (e. g. for documentation purpose)

→ [linux reference card.pdf](#) (brief outline)

→ [linux\\_command\\_reference.pdf](#) (detailed description)

→ Homework: Try the most common commands!

### Redirecting input and output in shell environment:

- Redirect standard output to file (substitution):  
`> cmd > ziel`
- Redirect standard output to file (append):  
`> cmd >> ziel`
- Redirect standard error output to file (substitution):  
`> cmd 2 > ziel`
- Redirect standard error output to file (append):  
`> cmd 2 >> ziel`
- Get standard input from file:  
`> cmd < quelle`
- Make standard output the standard input of another command (pipe):  
`> cmd1 | cmd2`

→ Homework: Test this!

## Variables

### Name

- Uppercase letters, lowercase letters,
- Numbers and underscores
- No numbers at the beginning!

### Type

### Value

assignment of a value by =

```
myVar=4711
```

Access to the value by \$

```
echo $myVar
```

## Variable substitution / variable expansion

String processing; the string contains wildcards.

Wildcards are given by a literal, e. g. %

Result is a new string. Here, the wildcards will be substituted by current values at runtime.

Example: `date +%Y_%m_%d` expands to `2024_10_08`

More examples with `date`:

`date +%Y` → 2024

`date +%Y_%m_%d` → 2024\_10\_08

## Simple backup skript

```
# A simple backup script
# Name : abackup

# datum looks like YYYY_MM_DD
datum=$(date +%Y_%m_%d)

# create a directory named backup_YYYY_MM_DD:
mkdir backup_$datum

# backup all C sourcefiles in the home directory there:
cp $HOME/*.c backup_$datum
```

→ [abackup](#)

# Shells

## History of shells

- 1978: sh = prime father of all shells
- csh = C-Shell
- ksh = Korn-Shell
- zsh = Z-Shell
- ash = A-Shell
- ...
- Bash = Bourne Again Shell – today's standard on Linux-Systems.  
Sometimes named **sh** (this is a symbolic link).

Choosing the shell by the first line of the script:

```
1 #! /bin/sh
```

## Predefined shell variables:

Variable	Bedeutung
<b>\$1 , \$2 , . . .</b>	1., 2., ... argument of programm call
<b>\$#</b>	number of program's arguments
<b>\$*</b>	all arguments (interpreted); if "\$*" is in the string
<b>\$@</b>	like \$*, but no interpretation if using "\$@"
<b>\$-</b>	Shell options
<b>\$?</b>	result of the last command
<b>\$\$</b>	process number of the shell
<b>\$!</b>	process number of the last command, started with '&'
<b>\$PATH</b>	list of directories to search commands

### conditional command execution

```
if condition  
then  
    cmd1  
  
else  
    cmd2  
  
fi
```

→ Homework:

Write a script **exam** to determine, if a student passed the exam or not.  
Transfer the percentage of points [0...100] to your script.  
The script should print „passed“ if the percentage is above 50 and „failed“ otherwise.

→ [signtest](#)

## Important conditions

condition	explanation
<b>test -f Filename</b>	true, if Filename exists
<b>test -d Directoryname</b>	true, if Directoryname exists
<b>test zk1 = zk2</b>	true, if strings <b>zk1</b> and <b>zk2</b> are identical
<b>test n1 -eq n2</b>	true, if numbers <b>n1</b> and <b>n2</b> are identical. More operators: <b>-lt, -le, -ne, -ge and -gt</b>
<b>test ! expression</b>	true, if negation of <b>expression</b> is true
<b>test ... -a ...</b>	AND composition
<b>test ... -o ...</b>	OR composition

Homework /  
Demo „exam“

## Case statement (I)

```
case Variable in
    word1) cmd1 ;;
    word2) cmd2 ;;
    word3) cmd3 ;;
    ...
esac
```

→ Homework:

Write a **casescript** to tell the number of remaining working days.

Your script should first determine the day of the week.

It should then print something like „2 days to go“ (in case of a working day)  
or „weekend“.

Use **case** the command only!

## For loop (I)

```
for i in ListOfWords  
do  
    cmd1  
    cmd2  
    ...  
done
```

## For loop (II)

```
k=".c"  
for i in *$k  
do  
    j=${i%$k}  
    make $j  
    rm -fv ${j}.o  
    . . .  
done
```

→ Homework:

Find out, what's going on here!

## For loop (II)

What's the effect of this Skript?!?

Hint: `echo *.txt` is to list all text files.

```
k=".c"  
for i in *$k  
do  
    j=${i%$k}  
    make $j  
    rm -fv ${j}.o  
    ...  
done
```

→ Homework: Find out!

→ [forloopsscript](#)

# for loop, while loop, until loop

→ Homework:

Write 3 different scripts for loops by using one of the following commands:

**for**  
**while**  
**until**

Your script should print all numbers between 0 and 10.

## Running shell scripts: Useful hints (I)

Open a new terminal window (Ubuntu desktop, not applicable for Windows 10 subsystem):

- Click the icon „terminal“ or
- <Alt><Ctrl><t>

Open / create a text file using the editor **nano** (terminal window):

- **nano myscriptfile**

Type and save a text file using the editor **nano** (editor window):

- **echo „hello world“**
- <Ctrl><o> → change / acknowledge filename → <return>
- <Ctrl><x>

Make a file (e. g. a script file) executable for the current user (not necessary for Windows 10 subsystem):

- **chmod u+x myscriptfile**

Run an executable script file in current directory (terminal window):

- **./myscriptfile**

## Running C programs: Useful hints (II)

Open a new terminal window (Ubuntu desktop):

- Click the icon „terminal“ or
- <Alt><Ctrl><t>

Open / create a C source file using the editor **gedit** (terminal window):

- **gedit mycfile.c**

Type and save a C source file file using the editor **gedit** (editor window):

- ```
#include <stdio.h>
int main (void) {
    printf(„hello world!\n“);
    return (0);
}
```
- <Ctrl><s>
- <Alt><F4>

### Running C programs: Useful hints (III)

Run the C compiler `gcc` to compile `mycfile.c` to `mycfile` (terminal window):

- `gcc -o mycfile mycfile.c`
- Without the specification `-o mycfile` the executable program will be named `a.out` (default).

Run this program (terminal window):

- `./mycfile`

Note: More sophisticated hints will be given later:

- Linking modules
- Using makefiles

## Command line options in C:

```
/* argvdemo.c */  
#include <stdio.h>  
  
int main(int argc, char *argv[]) {  
    int i;  
  
    for (i=1; i<argc; i++) {  
        printf("command line argument no. %d is %s\n",  
               i, argv[i]);  
    }  
}
```

→ Homework:

- 1) Get this running in your Linux environment at home.
- 2) Become familiar with the **gcc** compiler.

Usage: **./argvdemo one two three**

→ [argvdemo.c](#)

# **Betriebssysteme / Operating Systems**

## **Files + shared memory**

WS 2024

Prof. Dr.-Ing. Holger Gräßner

[08 OS-BS 2024 files - shared memory]



### fopen(), fclose()

```
FILE *fp = fopen(char *Name, char *Mode);
```

**fp:** file pointer

**Name:** file name

**Mode:** access mode "r", "w", "a", "r+", "w+" or "a+"

```
int fclose(FILE *fp);
```

### fread(), fwrite()

```
size_t fread(void *buf, size_t elsize, size_t nelem,  
            FILE *fp);
```

```
size_t fwrite(void *buf, size_t elsize, size_t nelem,  
             FILE *fp);
```

**buf:** Data buffer

**Elsize:** Block size in Bytes

**nelem:** Number of Blocks to read or to write

**fp:** File pointer

### open(), close()

```
int fd = open(char *Name, int Flags);
```

**fd:** File descriptor

**Name:** Filename

**Flags:** Access mode

"O\_RDONLY", "O\_WRONLY", "O\_RDWR"

"O\_APPEND", "O\_CREAT", "O\_NONBLOCK"

...

```
int close(int fd);
```

### read(), write()

```
size_t read(int fd, void *buf, size_t bytes);
```

```
size_t write(int fd, void *buf, size_t bytes);
```

**fd:** File descriptor

**buf:** Buffer

**bytes:** Number of Bytes

### Conversions: fdopen(), fileno()

```
FILE *fp = fdopen(int fd, char *Mode);
```

**fp:** File pointer

**fd:** File descriptor

**Mode:** Access mode "r", "w", "a", "r+", "w+" or "a+,,

```
int fd = fileno(FILE *fp);
```

**fd:** File descriptor

**fp:** File pointer

## Init of shared memory

```
shm_des = shm_open(shm_name, oflag, rights);
```

**shm\_des:** descriptor to access the memory object

**shm\_name:** Name of the memory object

**oflag:** Access mode

**rights:** Read or write rights

### Defining the size of shared memory

```
ftruncate(shm_des, shm_len);
```

**shm\_des:** descriptor to access the memory object

**shm\_len:** Size of the memory object in bytes

## Assigning shared memory

```
ptr = mmap(0, len, prot, MAP_SHARED, shm_des, offset);
```

**ptr:** Pointer to begin of memory block

**len:** size of memory block in bytes

**prot:** access rights

**shm\_des:** descriptor to access the memory object

**offset:** Offset of the memory block

## Message transmission with shared memory (I)

```
#include <stdio.h>  #include <unistd.h>    #include <stdlib.h>    #include <fcntl.h>
#include <string.h>  #include <sys/wait.h>   #include <sys/stat.h>  #include <sys/mman.h>

#define MODE  (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP)
#define MSGLEN 100

int main(void)
{
    char *memfile = "shtrans";
    int fd;
    int wstat;
    pid_t npid;
    char *ptr=NULL, *hptr=NULL;
```

→ [shared\\_memory.c](#)

## Message transmission with shared memory (II)

```
if ((fd=shm_open(memfile, O_RDWR|O_CREAT, MODE)) == -1) {
    printf ("Error: 'shm_open(create)' doesn't work!\n");
    return EXIT_FAILURE;
}

if ( ftruncate(fd, sizeof(*ptr)+MSGLEN) == -1) {
    printf ("Error: 'ftruncate()' doesn't work!\n");
    return EXIT_FAILURE;
}

if ((ptr = (char*) mmap(0, sizeof(*ptr)+MSGLEN,
PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0))== (char*) -1) {
    printf ("Error: 'mapping' impossible!\n");
    return EXIT_FAILURE;
}
```

→ [shared\\_memory.c](#)

## Message transmission with shared memory (III)

```
close(fd) ;  
  
hptr = ptr+1;  
  
*ptr = 0; /* ugly hack: first byte == 0 indicates */  
         /* 'freedom to write to memory block' */  
  
npid = fork() ;  
  
if (npid) {  
    usleep(100);  
  
    printf ("Parent process: Please type a message:\n");  
  
    *hptr = '\0' ;  
  
    fflush (stdin) ;  
  
    scanf("%[^\\n]",hptr);  
  
    *ptr = 1; /* ugly hack: first byte == 1 indicates */  
              /* 'Memory contains new data' */  
  
    wait(&wstat);
```

→ [shared\\_memory.c](#)

## Message transmission with shared memory (IV)

```
    printf ("Parent process: EXIT\n");
    return EXIT_SUCCESS;
} else {
    printf ("Child process: Waiting for a message...\\n");
    while (*ptr == 0) ; /* infinite loop */
    printf("Child process: I received this: \\n %s\\n", hptr);
    munmap(ptr, sizeof(*ptr)+MSGLEN);
    if (shm_unlink(memfile) == -1) {
        printf("Error: 'Unlink' impossible!\\n");
        return EXIT_FAILURE;
    }
    printf ("Child process: EXIT\\n");
    return EXIT_SUCCESS;
}
```

→ [shared\\_memory.c](#)

# Betriebssysteme / Operating Systems

## Computer system overview

SS 2022

Prof. Dr.-Ing. Holger Gräßner

03 OS-BS 2018 Computer system overview.pptx

## Computer System Overview

- Basic Elements
- Processor Registers
- Instruction Execution
- Interrupts
- Memory Hierarchy
- Cache Memory
- I/O Communication Techniques

The purpose of this section is to provide a common understanding of the underlying computer system hardware.

Since I assume that we all already have some experience with computers this section is brief.

# Computer System Overview

## Basic Elements

Processor Registers

Instruction Execution

Interrupts

Memory Hierarchy

Cache Memory

I/O Communication Techniques

# Basic Elements

| Element     | Description                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Processor   | Controls the operation of the computer and performs its data processing functions. In “ancient” times there was only one processor per system. Hence, it was called Central Processing Unit (CPU). Often specified by clock frequency like 3 GHz |
| Main Memory | Stores both data and programs. Contents are not permanent. Typical unit (today) is GB (Giga Byte)                                                                                                                                                |
| I/O modules | Move data between computer and external environment.                                                                                                                                                                                             |
| System bus  | Provides for communication among processors, main memory and I/O modules.                                                                                                                                                                        |

# Basic Elements

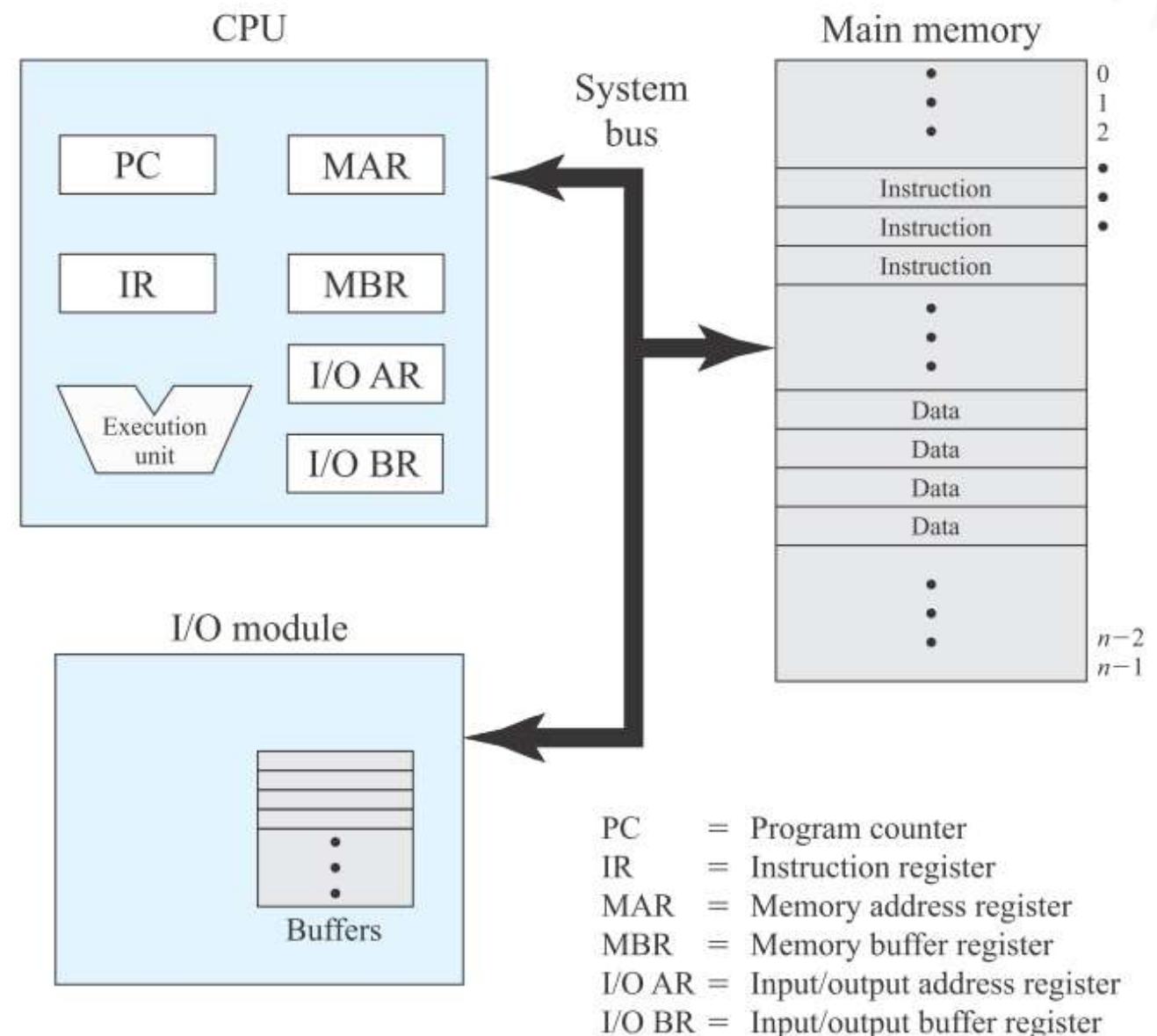


Figure 1.1 Computer Components: Top-Level View

# Computer System Overview

Basic Elements

Processor Registers

Instruction Execution

Interrupts

Memory Hierarchy

Cache Memory

I/O Communication Techniques

## Processor's registers

| Element        | Description                                                                                             |
|----------------|---------------------------------------------------------------------------------------------------------|
| PC             | (Program counter) Pointer to an instruction position in the memory.                                     |
| IR             | (Instruction register) contains the current instruction to be executed.                                 |
| MAR            | (Memory Address Register) Pointer to a data position in the memory.                                     |
| MBR            | (Memory Buffer Register) contains data to be written to or being read from memory.                      |
| I/O AR and BR  | Similar registers for I/O modules.                                                                      |
| Execution unit | Performs the command, numerical calculations (integer and floating point), logical operations, and more |

# Processor Registers

**Registers** is a kind of storage that is built into a processor and hence it has very fast access to it. However, this kind of memory is small (in contrast to the main memory).

**User-visible registers** Usable by machine code to minimize accesses to the main memory (slow!). Usually optimizing compilers try to exploit registers to speed up the execution of code.

**Control and status registers** Used by the processor to control the operation of the processor and by privileged OS routines.

## User-visible registers

| Element          | Description                                                                                                                               |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Data register    | General purpose and accessible to the programmer. Sometimes (depending on the processor) restrictions (e.g. floating point)               |
| Address register | contain main memory addresses of data or instructions. Here, we have again various types like index registers, segment or stack pointers. |
|                  |                                                                                                                                           |
|                  | Often, user-visible registers are saved in case of a procedure call.                                                                      |

## Control and Status Register

Usually not visible to the ordinary machine code but visible in modes which are considered to have a higher status (“kernel mode”, “control mode”).

Examples:

| Element | Description                                                               |
|---------|---------------------------------------------------------------------------|
| PC      | (program counter) Pointer to an instruction position in the memory        |
| IR      | IR (instruction register) contains the current instruction to be executed |

## Condition Codes (Flags) and Other Registers

Besides the above examples, condition codes are used. They encode information as bits that is necessary to control the execution.

Examples:

| Element | Description                                                                                                                                            |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| ZERO    | A flag that signals whether the last arithmetic operation returned the value 0. Similar flags for overflows, positive and negative results are common. |
| KERNEL  | A flag showing whether the processor is in kernel mode or not.                                                                                         |

There are also registers for **interrupts** (pointing to the interrupt handling routine) or **stack pointers**.

# Computer System Overview

Basic Elements

Processor Registers

Instruction Execution

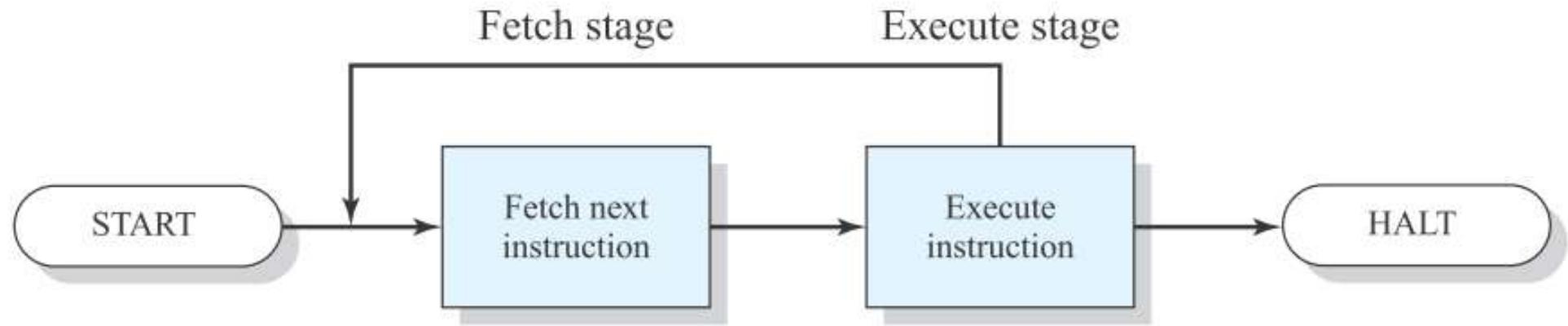
Interrupts

Memory Hierarchy

Cache Memory

I/O Communication Techniques

## Instruction Execution



**Figure 1.2 Basic Instruction Cycle**

A processor executes instructions sequentially in the above cycle (well, I hope that you know that this is not the whole truth...):

- The instruction execution cycle fetches the next instruction first. The information where the next instruction is located in the memory is stored in the IR.
- Then the instruction is executed. Here, the program counter is incremented in the default case or changed (“jump instruction” or “branching instruction”).

→ Pipeline

## Instructions



(a) Instruction format

Instructions consists of two parts:

| Element | Description                                                                                                                                             |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| Opcode  | This part <b>encodes</b> the <b>operation</b> . The width of the opcode is fixed but varies with the processor. (“What is to do?”)                      |
| Address | This part specifies an <b>object</b> of the operation. For instance, it can be an address where a number is stored which should be added to a register. |

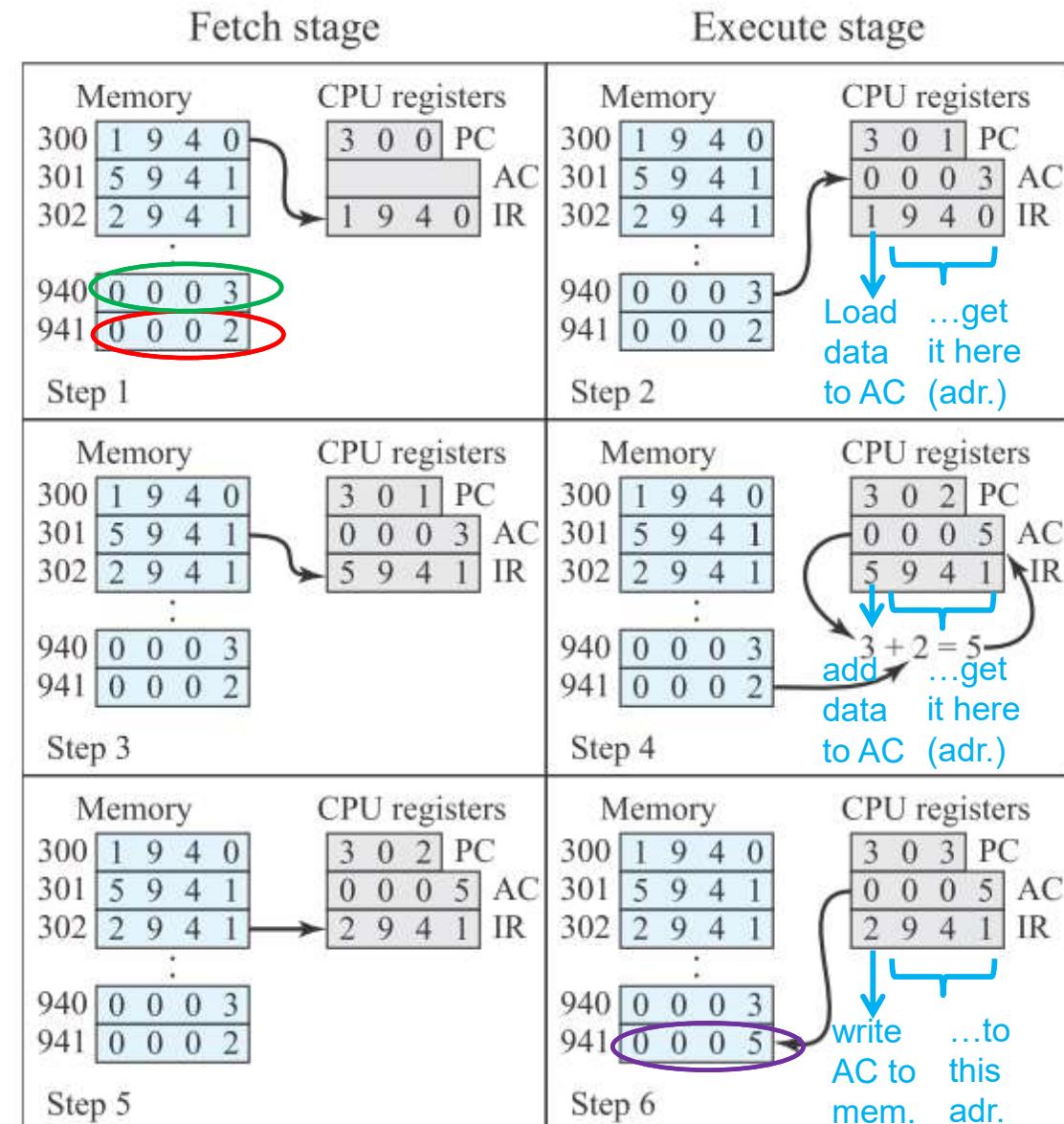
## Instruction Categories

| Element         | Description                                                                                                     |
|-----------------|-----------------------------------------------------------------------------------------------------------------|
| Process memory  | Data may be transferred from processor to memory or vice versa.                                                 |
| Processor I/O   | Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module. |
| Data processing | The processor may perform some arithmetic or logic operation on data.                                           |
| Control         | An instruction may specify that the sequence of execution be altered.                                           |

## Example of Program Execution

Purpose of this code fragment:

- Add the data (0003) at adress #940
- to the data (0002) at adress #941
- and store the result (0005) at adress #941



**Figure 1.4 Example of Program Execution** (contents of memory and registers in hexadecimal)

# Computer System Overview

Basic Elements

Processor Registers

Instruction Execution

**Interrupts**

Memory Hierarchy

Cache Memory

I/O Communication Techniques

## Interrupts

**Interrupts** are a mechanism to react to certain events in a computer system that require an urgent handling.

| Element          | Description                                                                                                                                                                                                                             |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Program          | Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space. |
| Timer            | Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.                                                                                                            |
| I/O              | Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.                                                                                                                 |
| Hardware failure | Generated by a failure, such as power failure or memory parity error.                                                                                                                                                                   |

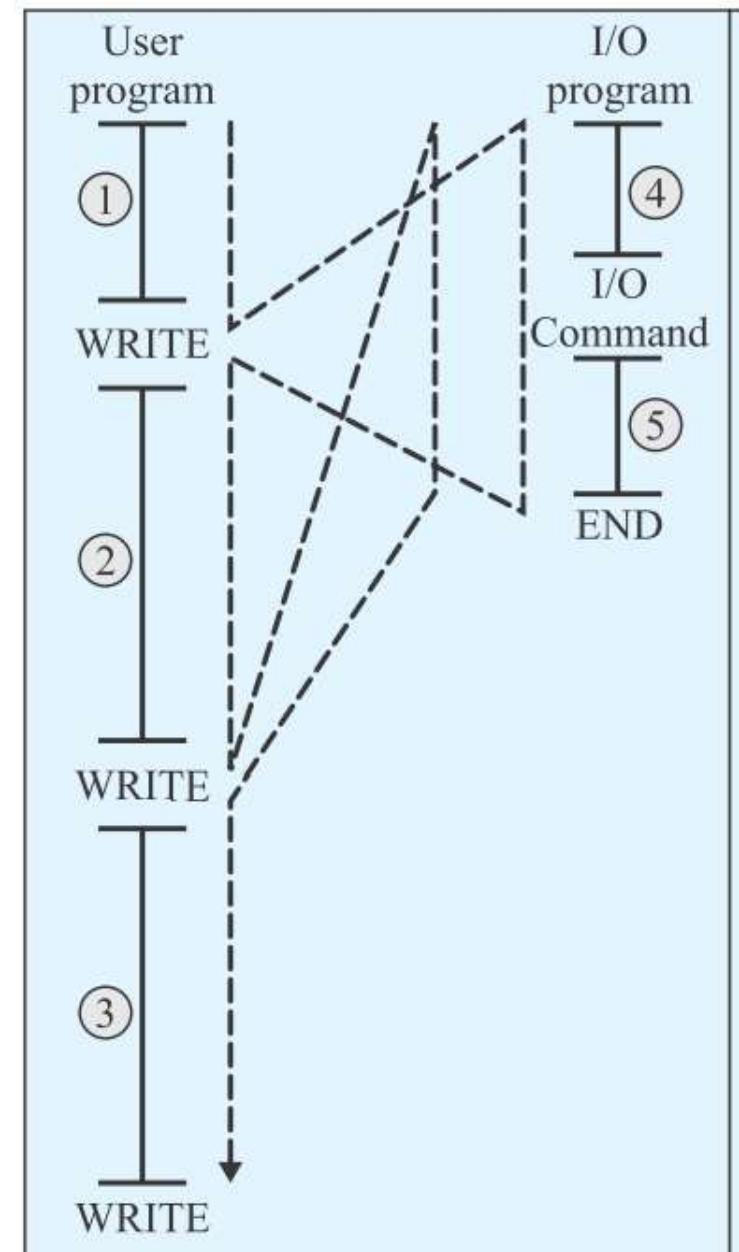
In case of I/O interrupts the benefit is that the processor is allowed to exploit the waiting time (I/O is slow) until the I/O operation is possible.

## Control Flow without Interrupts

**No interrupts:** The user program requires some I/O operation which itself require (for instance)

- ▶ a preparation part (filling a buffer) (4)
- ▶ a command to start the I/O operation
- ▶ a finishing part (setting flags, freeing of device...) (5)

**Consequence:** Long waiting times (note that I/O operations are million times slower than the execution cycle of the processor).



(a) No interrupts

## Control Flow with Interrupts

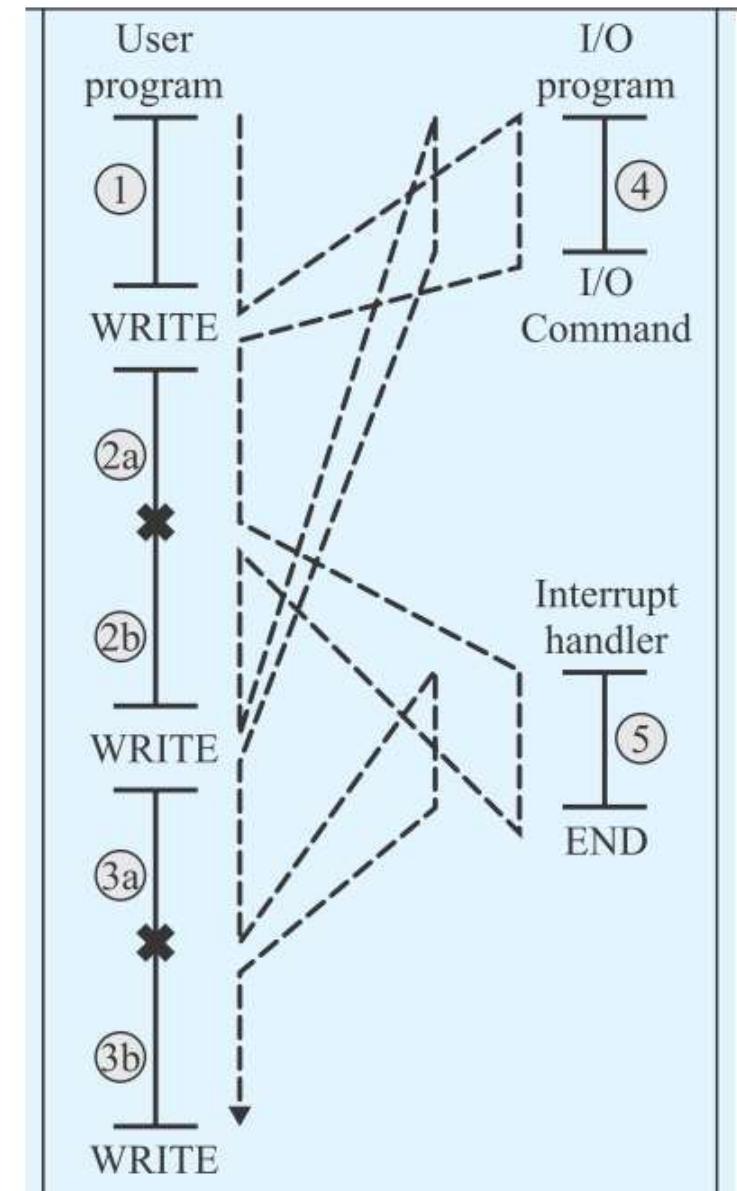
With interrupts: The user program requires some I/O operation which itself require (for instance)

- ▶ a preparation part (filling a buffer) (4)
- ▶ a command to start the I/O operation.

After this the user program can continue.

- ▶ a finishing part (setting flags, freeing of device...) (5). This is now started with setting an interrupt flag, hence the user program is suspended. Afterwards, the user program is resumed.

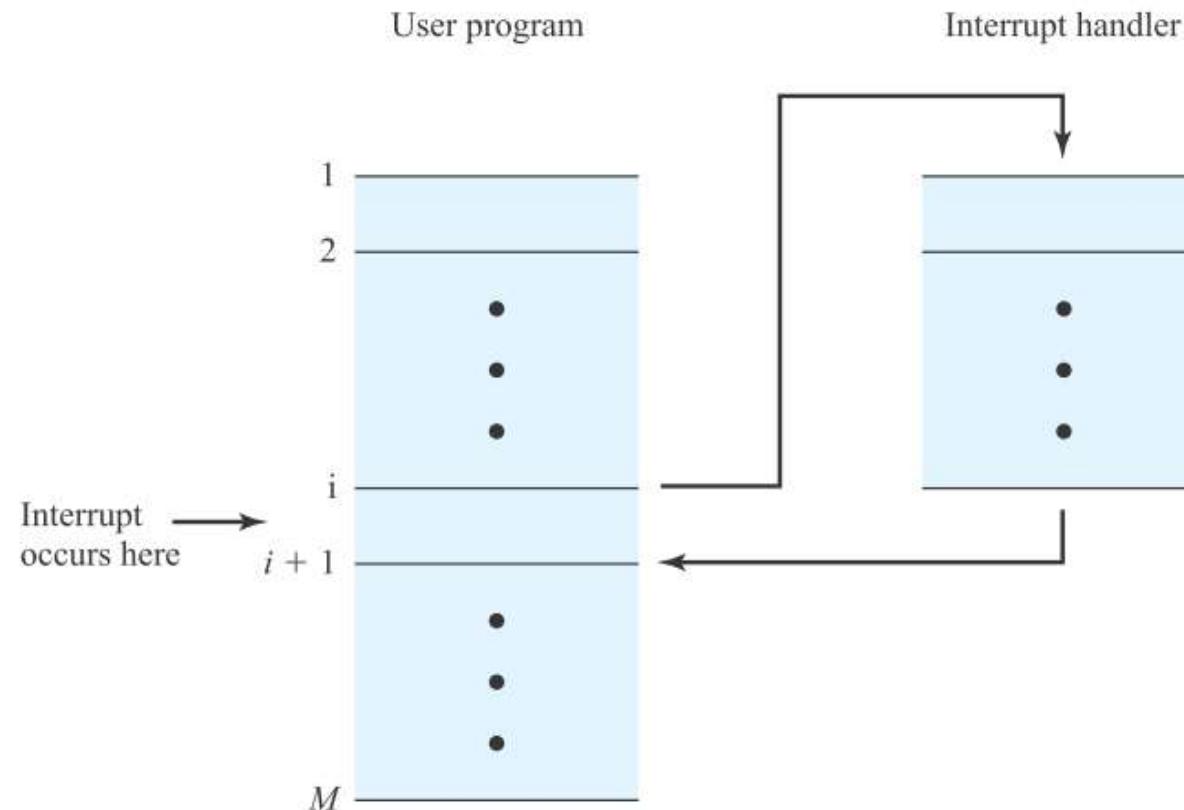
Consequence: No waiting times provided that the I/O operation are finished early enough.



(b) Interrupts; short I/O wait

## Handling of Interrupts (I)

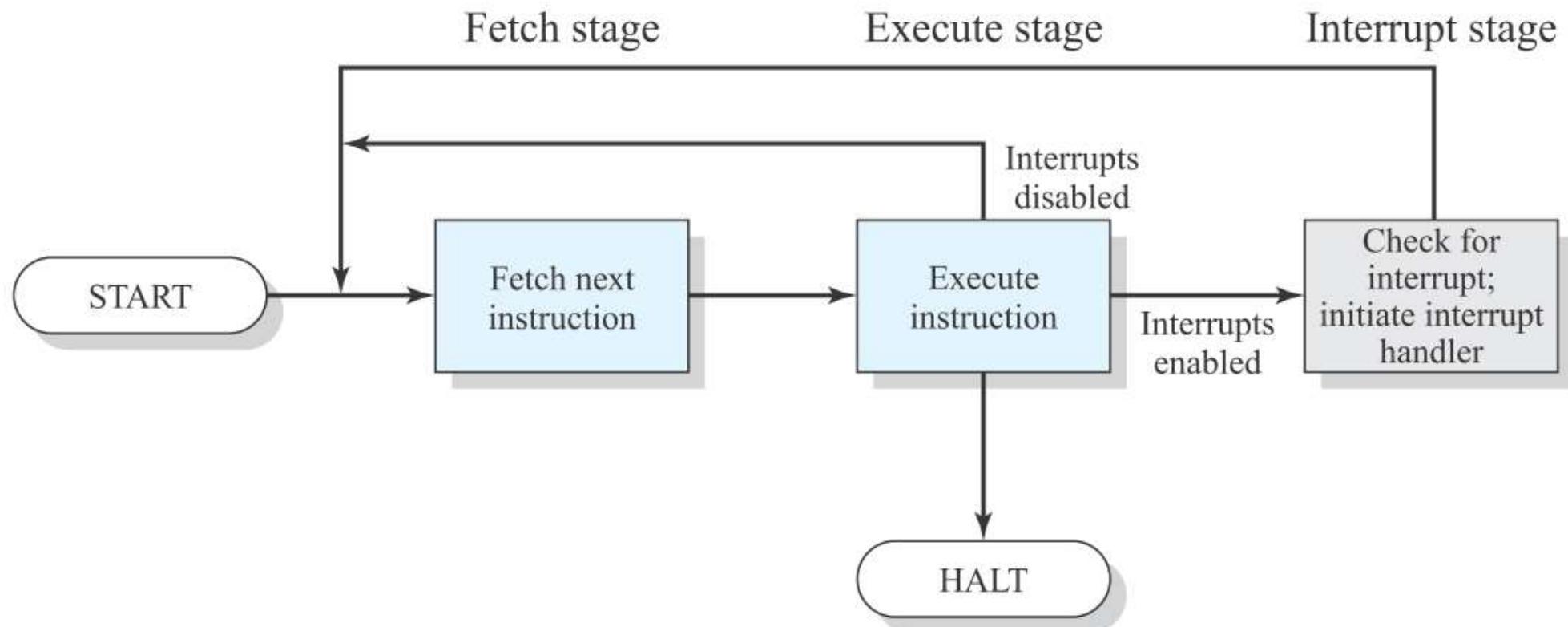
The user program does **not** have to handle the interrupts itself, both the processor and the OS are responsible for that.



**Figure 1.6** Transfer of Control via Interrupts

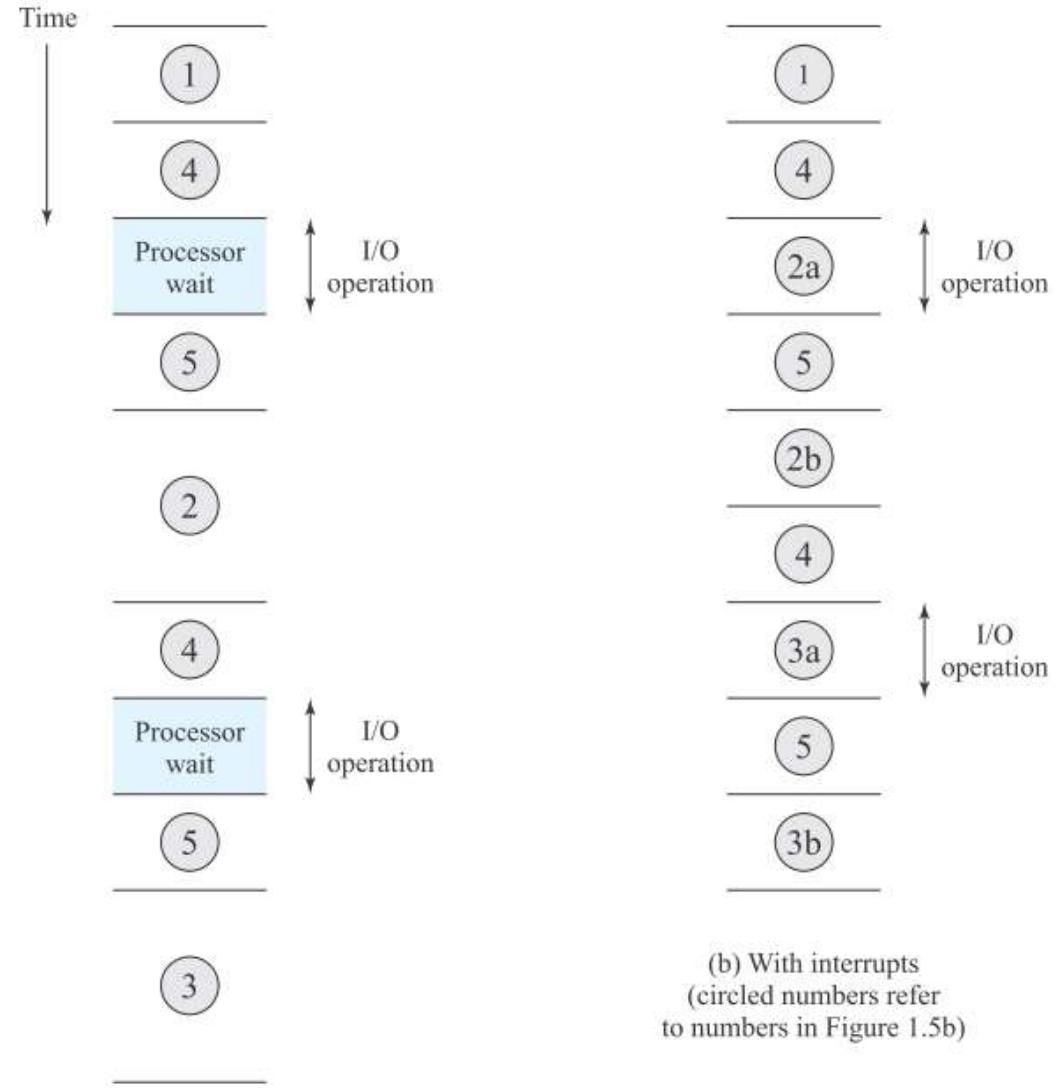
## Handling of Interrupts (II)

The user program does **not** have to handle the interrupts itself, both the processor and the OS are responsible for that.



**Figure 1.7** Instruction Cycle with Interrupts

## Control flow with interrupts

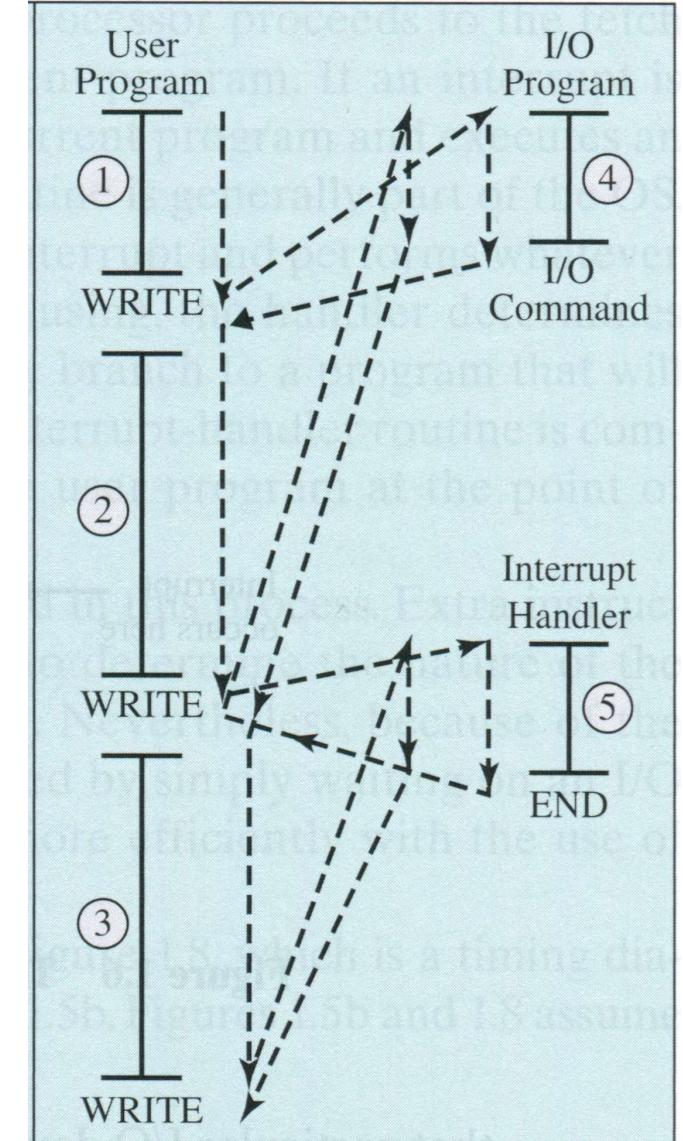


## Control flow with interrupts (revisited)

With interrupts and SLOW I/O.

Consequence: Waiting times in case of I/O operation even with interrupts.

Does it still pay off?



(c) Interrupts; long I/O wait

## Control flow with interrupts (revisited II)

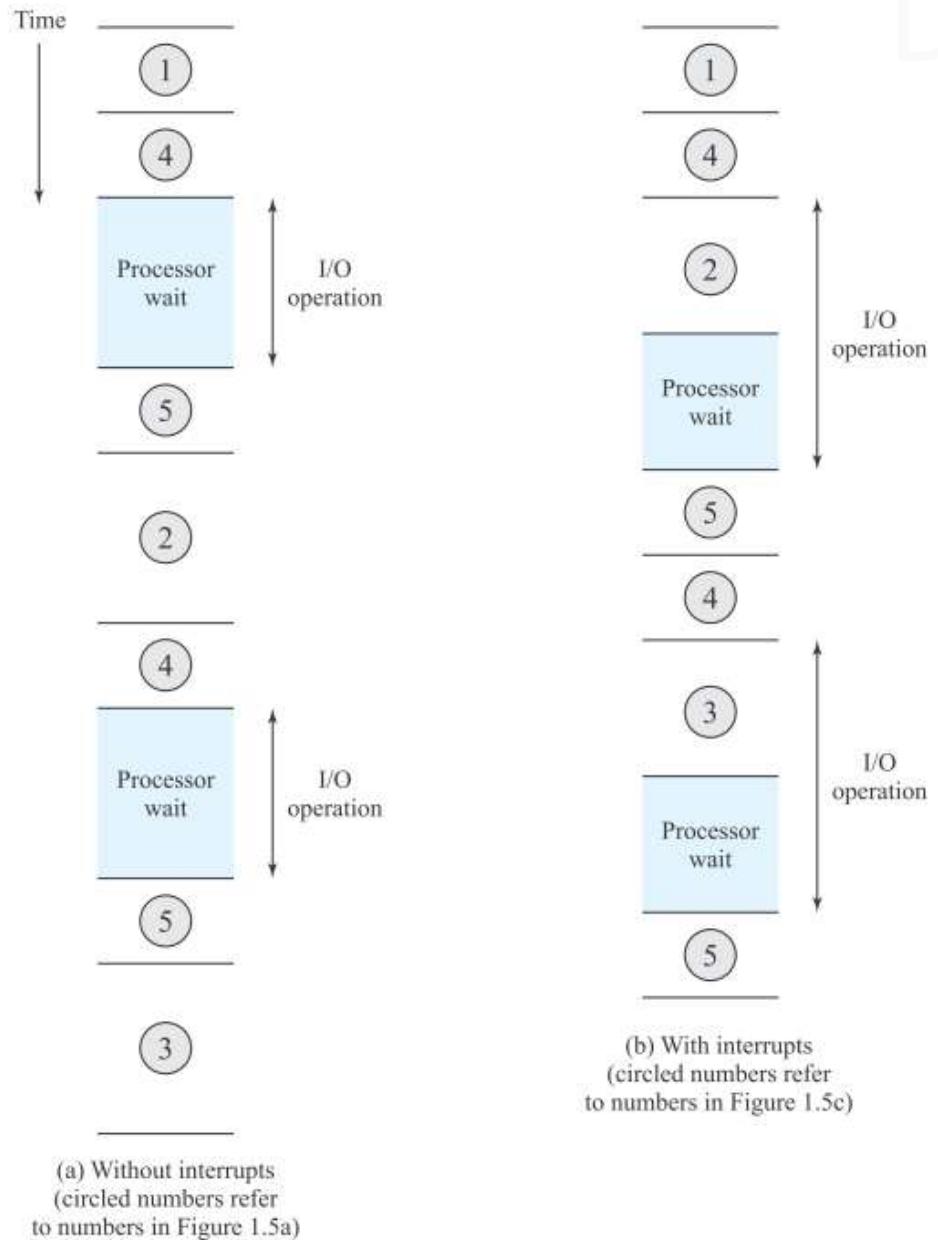


Figure 1.9 Program Timing: Long I/O Wait

## Exercise

1. Build groups consisting of 3 members.
2. Collect actions to be done while processing an interrupt completely.
3. Sequentialize these actions.
4. Which of these actions are under the control of the HW  
and which of them are SW controlled?

## Exercise

1. Build groups consisting of 3 members.
2. Collect actions to be done while processing an interrupt completely.
3. Sequentialize these actions.
4. Which of these actions are under the control of the HW  
and which of them are SW controlled?

## Interrupts

## Interrupt Processing (processor's job)

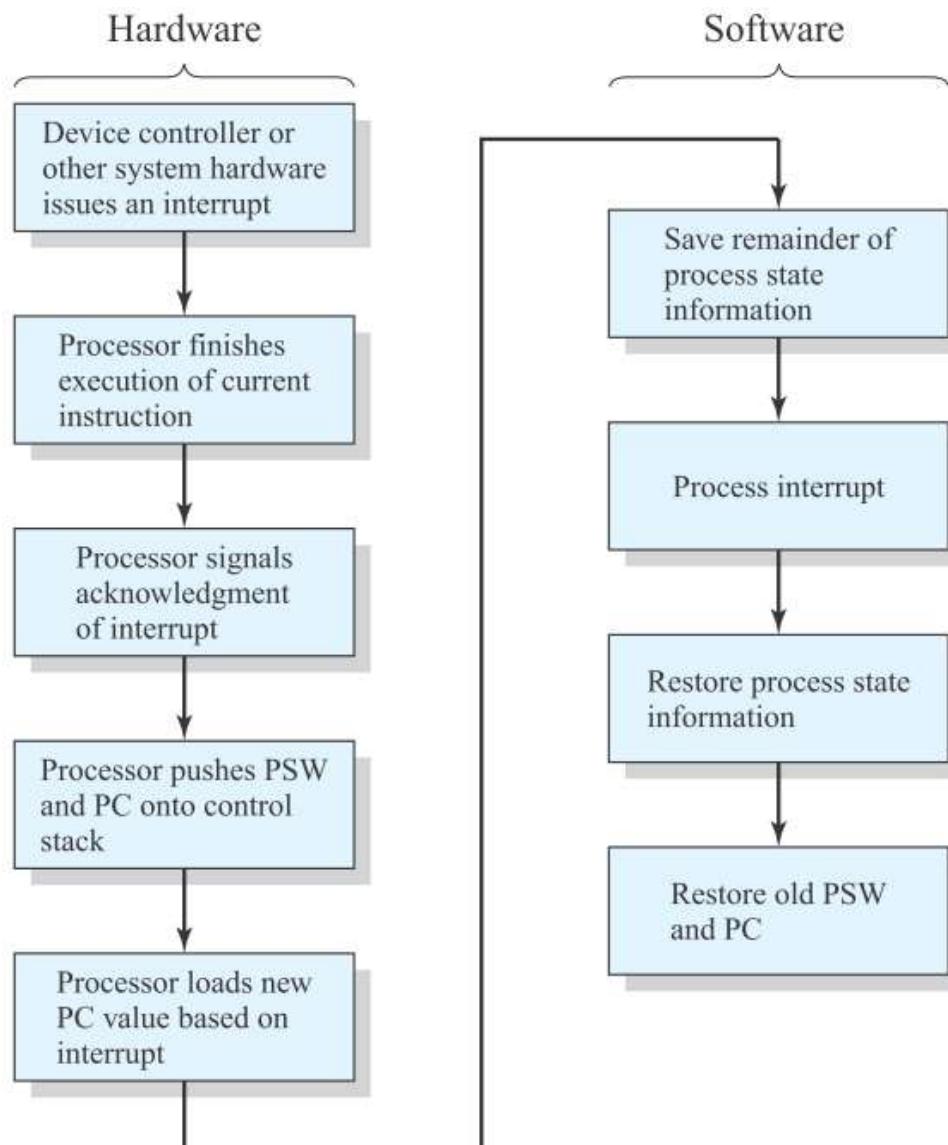
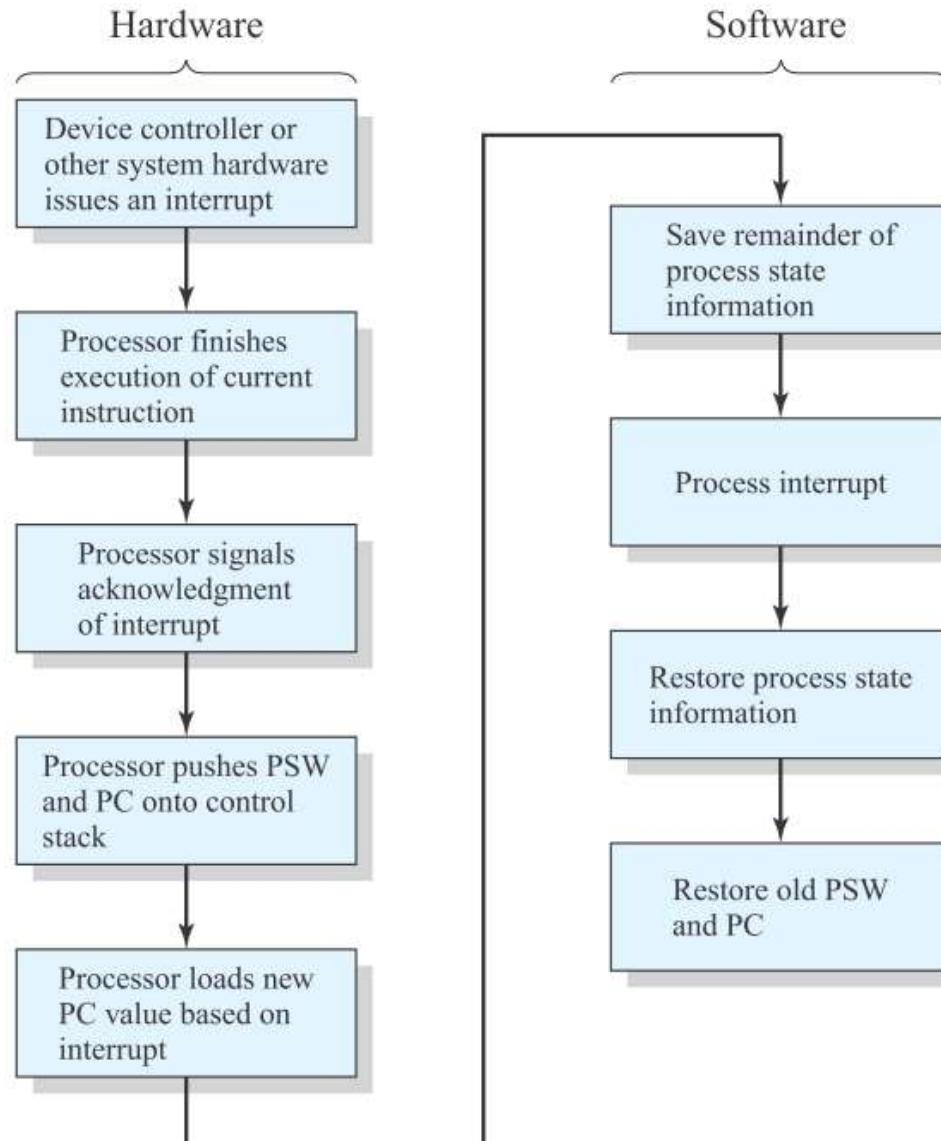


Figure 1.10 Simple Interrupt Processing

1. A device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor detects an interrupt signal and sends an acknowledgement signal to the interrupting device. The device resets the signal.
4. The processor pushes the PC and PSW (program status word) onto a control stack.
5. The processor now starts an interrupt handling routine.  
Depending on the system it can be
  - a fixed routine,
  - a routine dedicated to that particular interrupt or
  - a routine dedicated to that particular device.



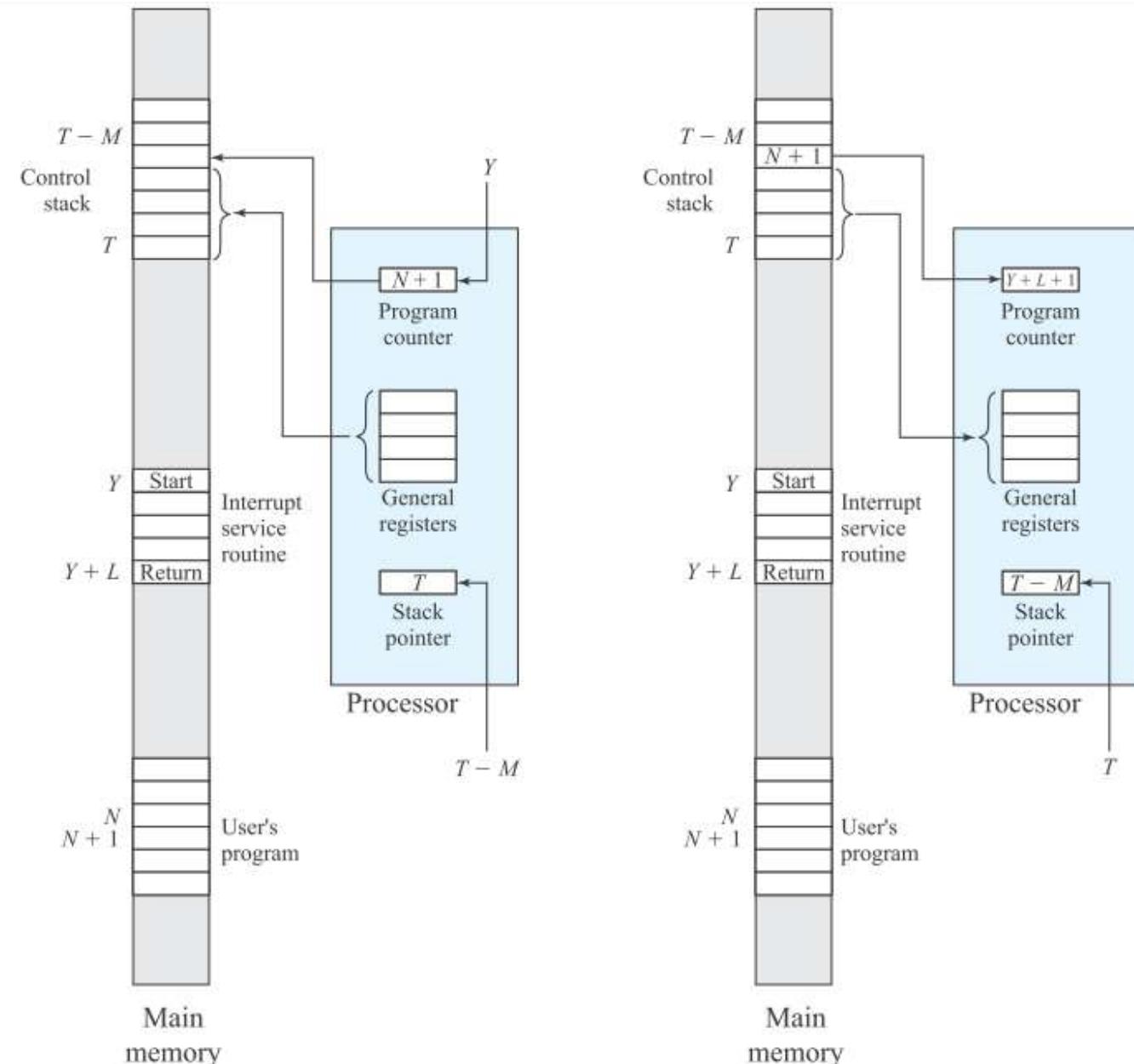
## Interrupt Processing (software's job)

Now control is handed over to the **interrupt routine!**

6. The remaining state of the **interrupted program** is saved (registers etc.).
7. Now the interrupt handling routine is able to do its work, i. e. to deal with the interrupting device.
8. If finished, then the saved status information of the interrupted user program is restored to the processor registers.
9. Finally, PC and PSW are restored.

Figure 1.10 Simple Interrupt Processing

# Interrupt Processing



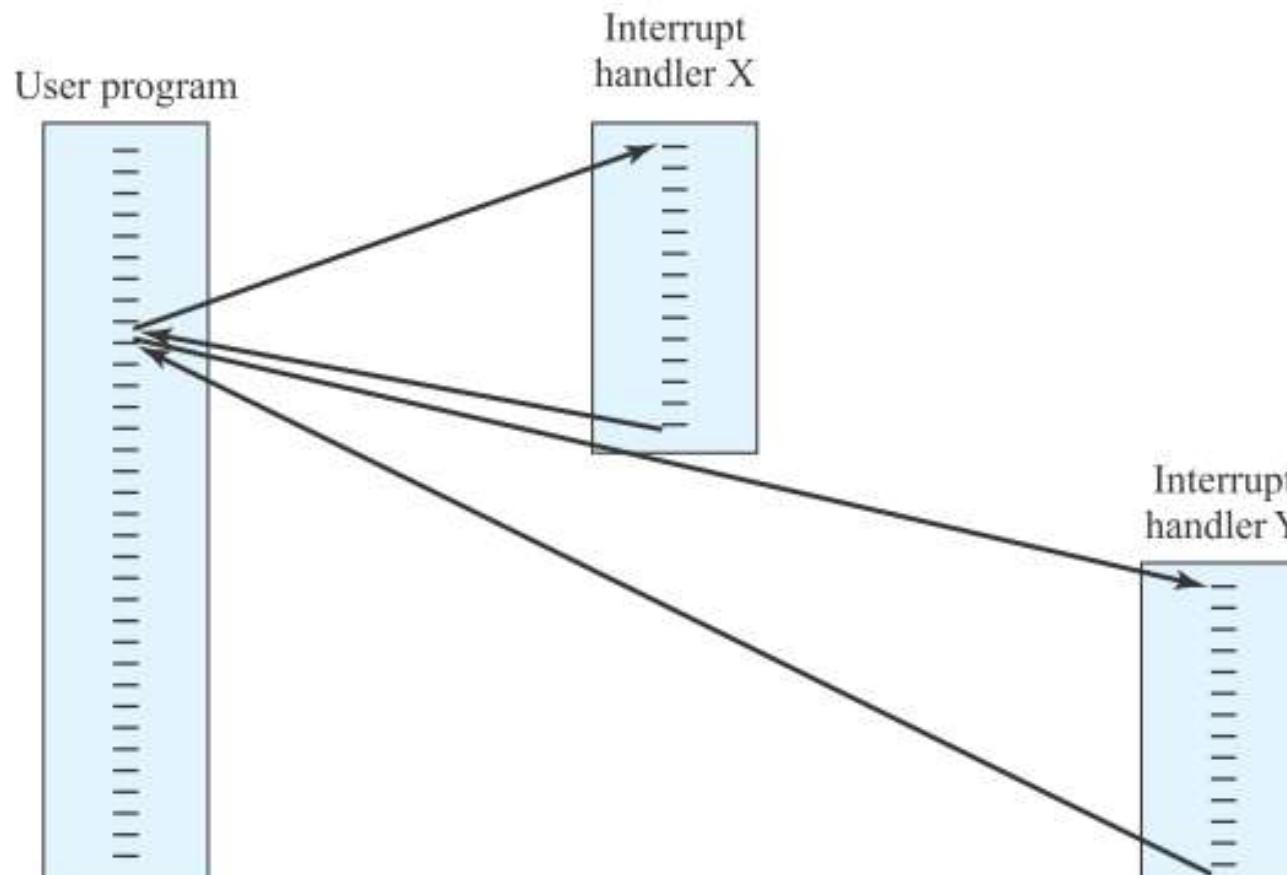
Any problems?

Does anyone observe a problem?

What happens if an interrupt occurs during an ISR?

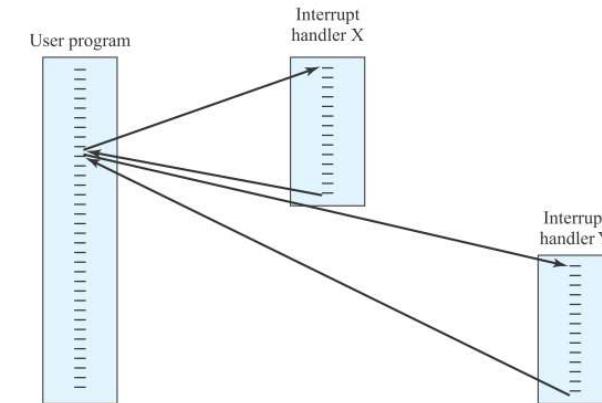
## Multiple interrupts

**Simple option:** Handle all interrupts sequentially.



(a) Sequential interrupt processing

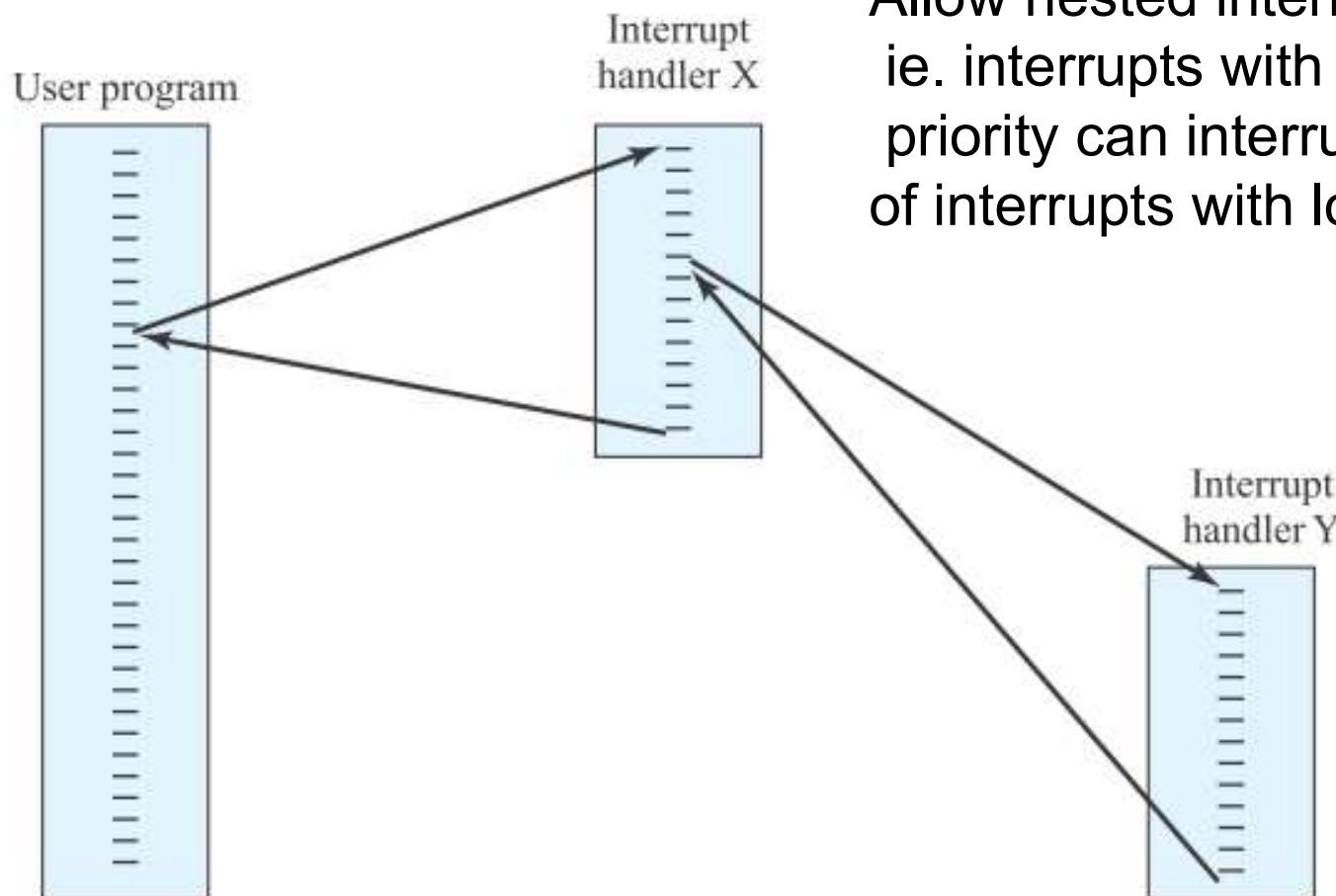
## Multiple interrupts



(a) Sequential interrupt processing

| PLUS         | MINUS                  |
|--------------|------------------------|
| Simple       | No priorities possible |
| Strict order |                        |

## Multiple interrupts



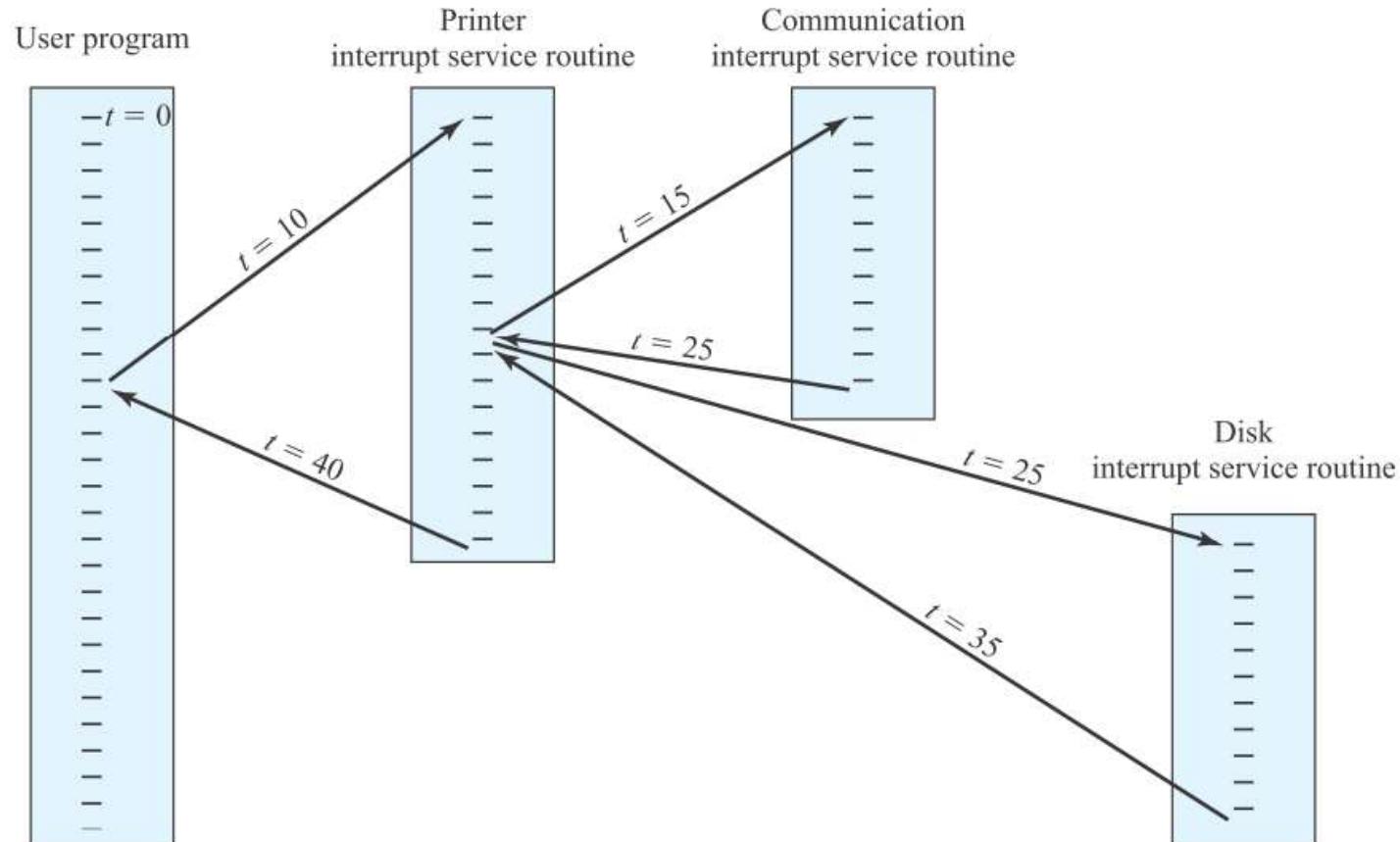
(b) Nested interrupt processing

**Priority based option:**  
Allow nested interrupts,  
ie. interrupts with higher  
priority can interrupt ISRs  
of interrupts with lower priority.

## Multiple Interrupts: Example

Priorities: Printer < Disk < Communication

Interrupts: Printer ( $t = 10$ ), Communication ( $t = 15$ ), Disk ( $t = 20$ )



**Figure 1.13** Example Time Sequence of Multiple Interrupts

## Computer System Overview

Basic Elements

Processor Registers

Instruction Execution

Interrupts

**Memory Hierarchy**

Cache Memory

I/O Communication Techniques

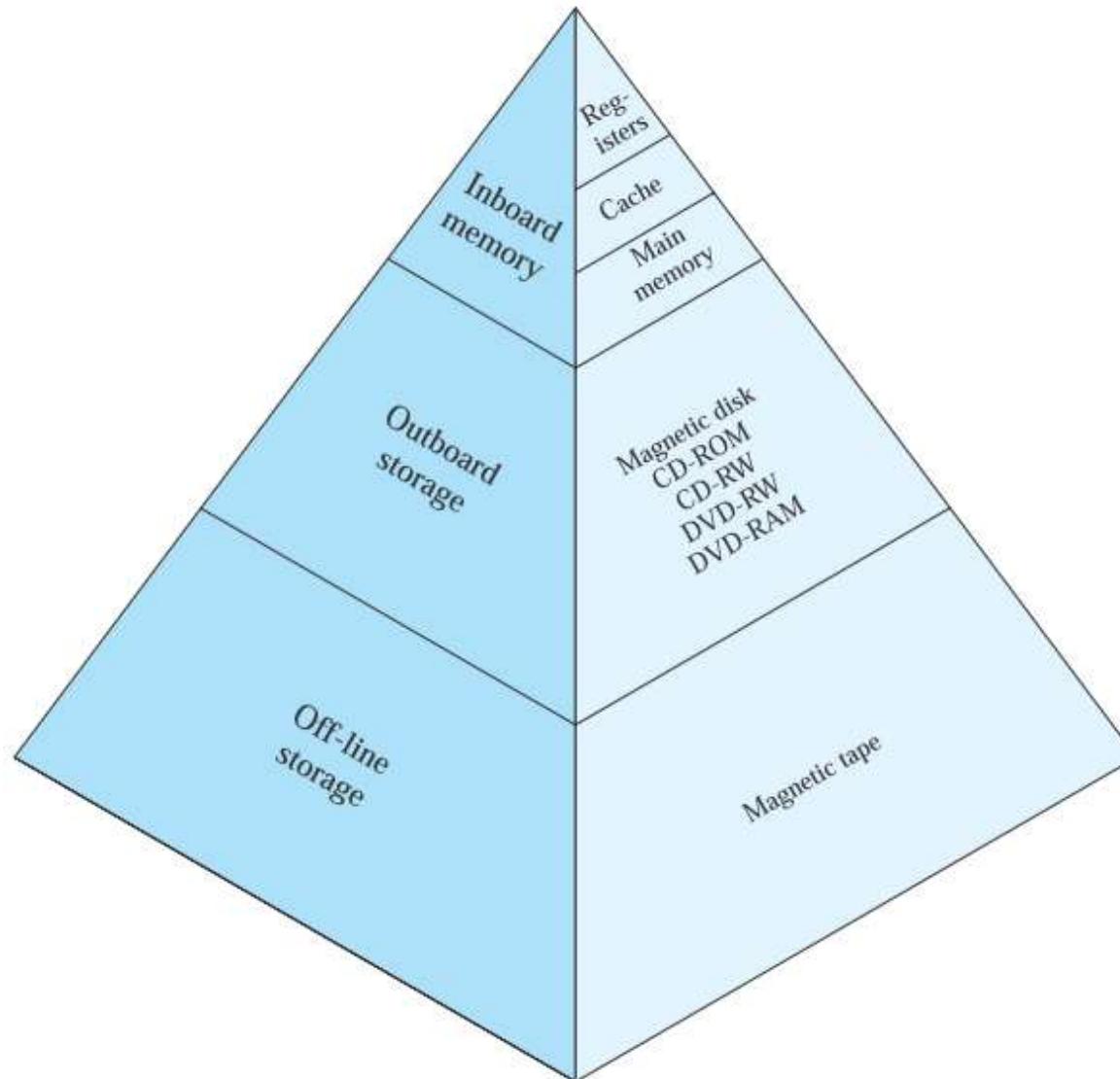
# Memory Hierarchy

Regarding memory we have to make a trade-off between the following objectives:

- Fast access
- Capacity
- Costs (per bit)

Since there is no technology that is optimal in all three aspects, computer system designer construct a **memory hierarchy**.

## Memory Hierarchy



Going **down** the hierarchy we have:

- decreasing costs per bit,
- increasing capacity,
- increasing access time, and
- **decreasing frequency of access** (hopefully)

### Example

Example (A simple 2-level memory hierarchy)

Consider a memory hierarchy with 2 levels only. The access time for the first level is  $0.1 \mu s$  and for the second level it is  $1 \mu s$ .

Suppose that we have a hitratio  $H$  of 95%. Then we get the following average access time:

$$0.95 * (0.1 \mu s) + 0.05 * (1 \mu s + 0.1 \mu s) = 0.15 \mu s$$

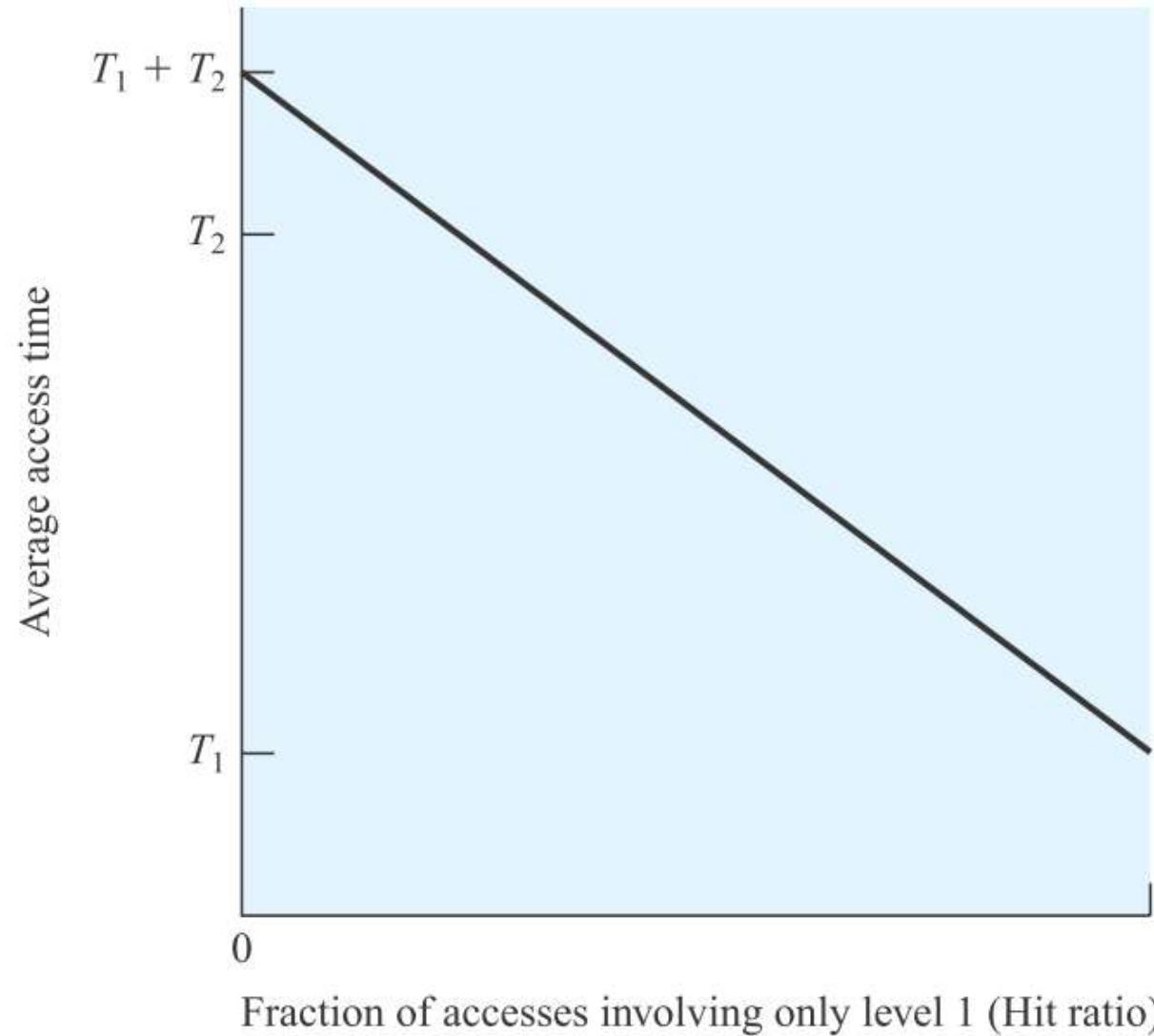
Hence, having an excellent hitratio leads to a memory system that has

- an access time that is close to the fastest level and
- a capacity that is the sum of both levels.

“hit” the accessed word is found in the faster memory

“miss” otherwise.

## Hit Ratio



### Locality of Reference

Why do we have high hit ratios ( $> 90\%$ ) even if the faster memory has a low capacity ( $< 10\%$ ) in contrast to the slower memory?

During execution of a program, memory references by the processor are likely to be close to the previous references:

- Instructions are executed (and stored) sequentially. Jumps are often “small” jumps.
- Data is also accessed in a clustered way.

This principle – **knows as locality of reference** – holds if we consider the reference of a short term. Over a long term this does not hold, ie. there are always clusters, but they change dynamically.

As a consequence, the computer system should react dynamically to the change of these cluster for optimal performance.

One well known example: **Caches**.

# Computer System Overview

Basic Elements

Processor Registers

Instruction Execution

Interrupts

Memory Hierarchy

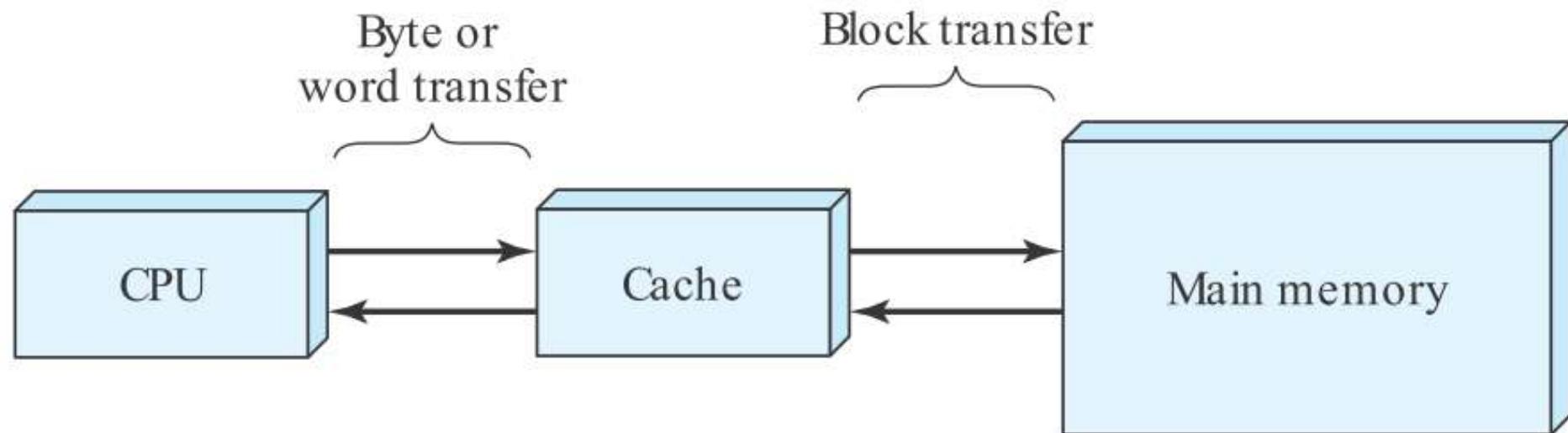
**Cache Memory**

I/O Communication Techniques

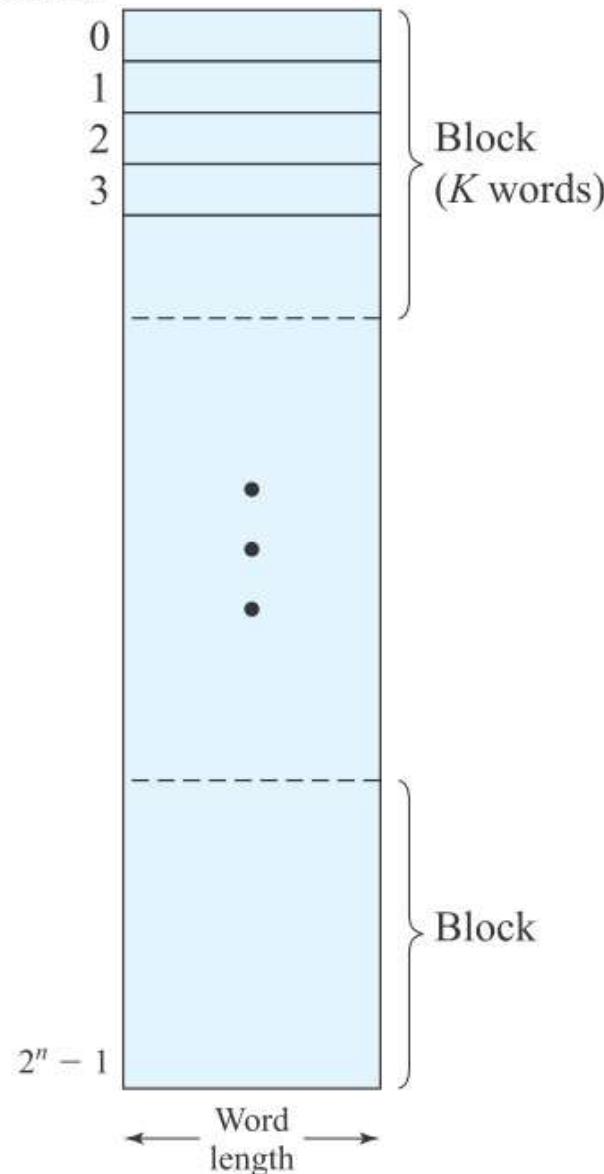
## Cache memory

Faster than main memory, slower than registers.

- Less capacity than main memory, more than registers.
- Not visible to the programmer nor to the OS.
- Controlled by dedicated memory management HW.
- Urgently needed because processor speed is limited by the memory access speed. The latter did not increase in the same way in the last decades.



Memory address



## Structure of a Cache-Memory System (I)

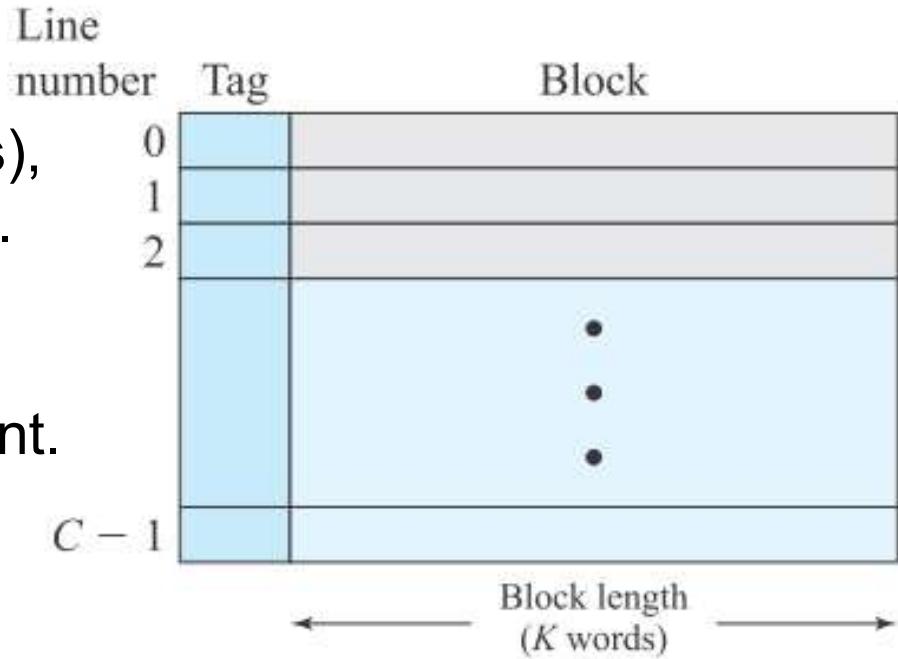
### Main Memory:

- Consists of  $2^n$  addressable words, each word has a unique  $n$ -bit address.
- It is separated into blocks of  $K$  words ( $K$  is typically a number of the form  $2^m$ ). Hence, there are  $M = 2^n / K$  many blocks.

## Structure of a Cache-Memory System (II)

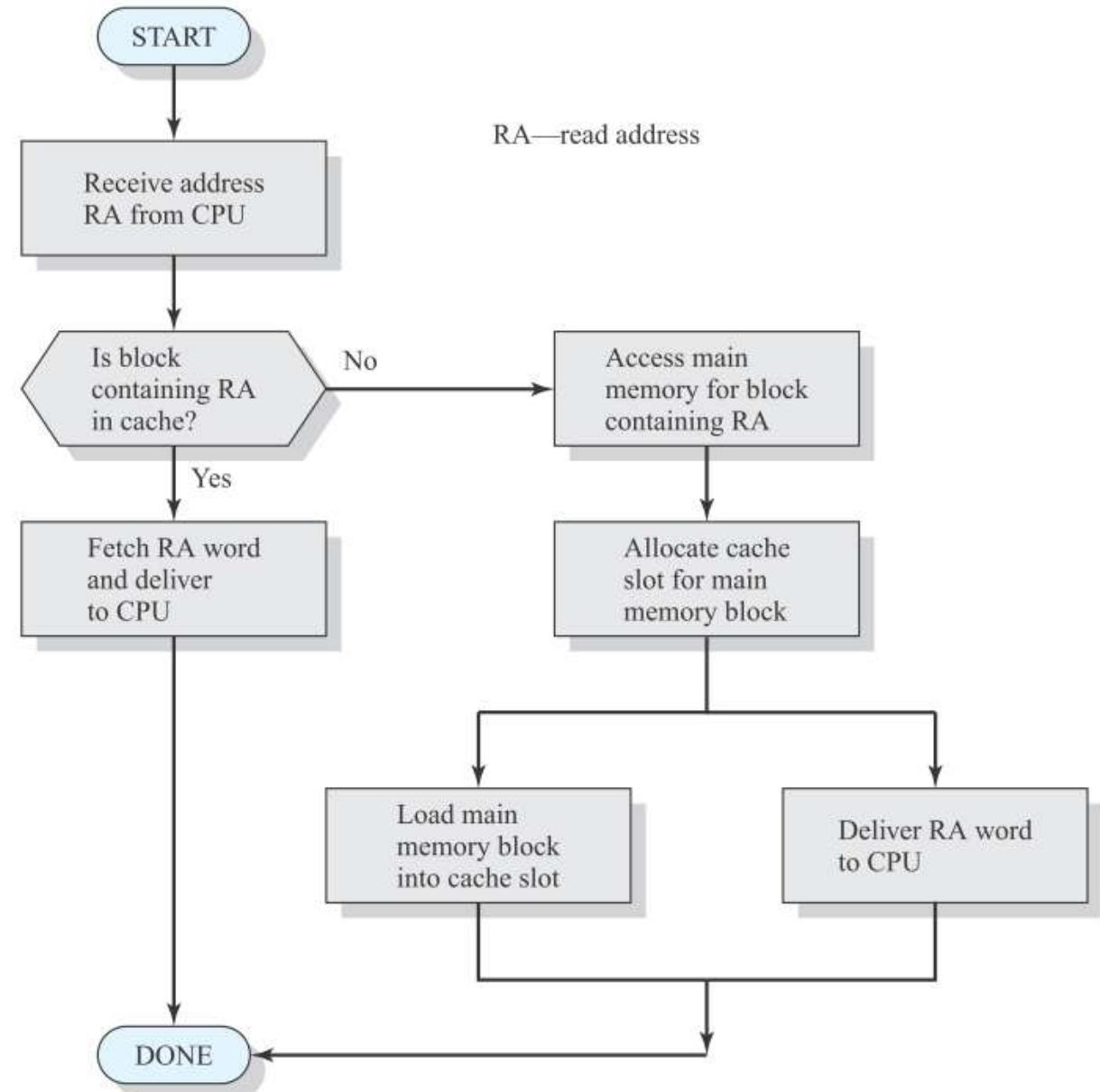
### Cache:

- Consists of  $C$  blocks (often called **lines**), where  $C$  is dramatically smaller than  $M$ .
- If a word is accessed, then the cache management unit checks, whether this word resides in the cache at the moment.



- To do so, it has to check the **tags** that is number of the block that is the corresponding line currently. If the block (number) is found in the cache (in the set of tags), then we have a **cache hit** and the contents of the memory word are sent to the processor.
- Otherwise, we have a “cache miss” and the cache management unit will move the **whole block** containing the desired word into the cache.

## Cache Read Operation



### Exercise

1. Build groups consisting of 2 members.
2. Imagine that you are cache designers now. What are the options for your cache? Think about both the HW related options and SW related options.

### Exercise

1. Build groups consisting of 2 members.
2. Imagine that you are cache designers now. What are the options for your cache? Think about both the HW related options and SW related options.

## Cache Design (I)

Designing a cache boils down to make decision regarding:

| Element    | Description                                                                                                                                                         |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Cache size | The experience tells us that even small cache have already a significant impact on the overall performance of a system. However: Larger cache are better . . .      |
| Block size | First the hit ratio increases with the block size due the locality of references. But it decreases when the number of clusters is higher than the number of blocks. |

## Cache Design (II)

Designing a cache boils down to make decision regarding:

| Element               | Description                                                                                                                                                                                                                                                                                                                          |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Mapping function      | This functions maps blocks to lines in cache. It can be simple (one admissible line per block) or complex (each line can contain a block). Trade-off between the complexity of the mapping function (HW!) and flexibility of the cache.                                                                                              |
| Replacement algorithm | Whenever a new block is copied into the cache another one has leave it. If a block has more than one admissible line in the cache, the question is: Which block has to be replaced? A standard solution: The block that is <b>least recently used (LRU)</b> .                                                                        |
| Write policy          | If the processor changes a word that is in the cache currently, the question is when this change has to be written back to the main memory. At one extreme, writing is always done. At the other extreme, writing done only when the block is replaced in the cache.<br><b>What can be a problem if the last strategy is chosen?</b> |

## Computer System Overview

Basic Elements

Processor Registers

Instruction Execution

Interrupts

Memory Hierarchy

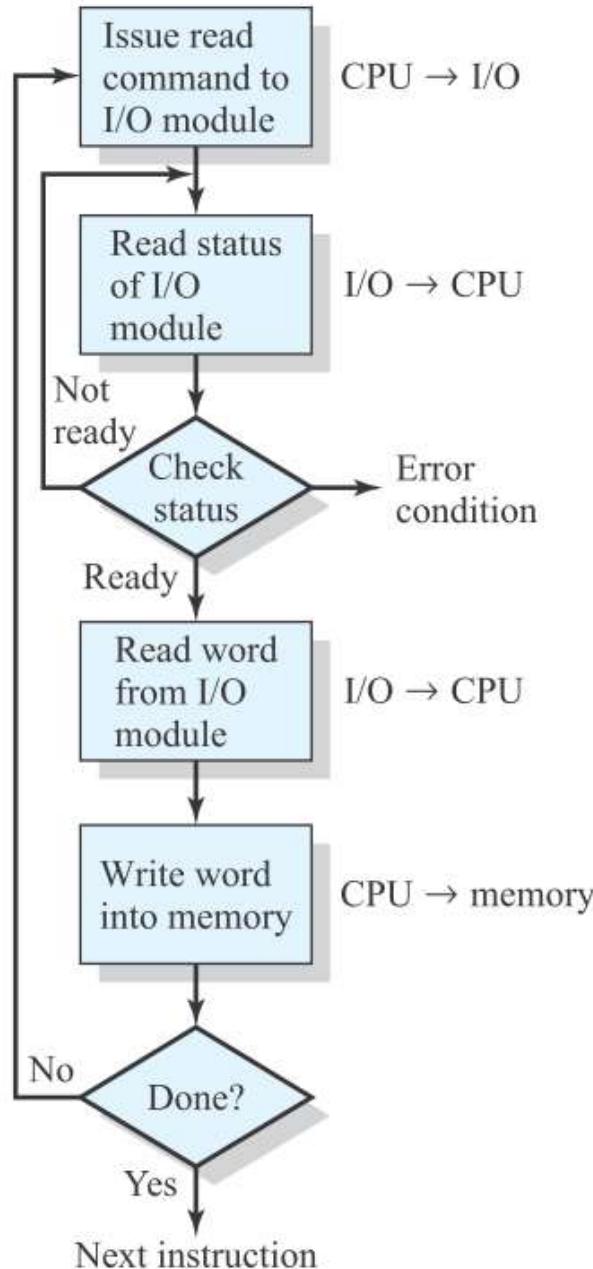
Cache Memory

I/O Communication Techniques

## I/O Communication Techniques

There are three kinds of I/O Communication:

| Element                    | Description                                                                                                       |
|----------------------------|-------------------------------------------------------------------------------------------------------------------|
| Programmed I/O             | The processor is responsible for the correct handling of the I/O operations, no interrupts are used to inform it. |
| Interrupt driven I/O       | Processor is triggered via interrupts to react to I/O operations in time.                                         |
| Direct memory access (DMA) | I/O operations are delegated to a dedicated Component                                                             |



## Programmed I/O

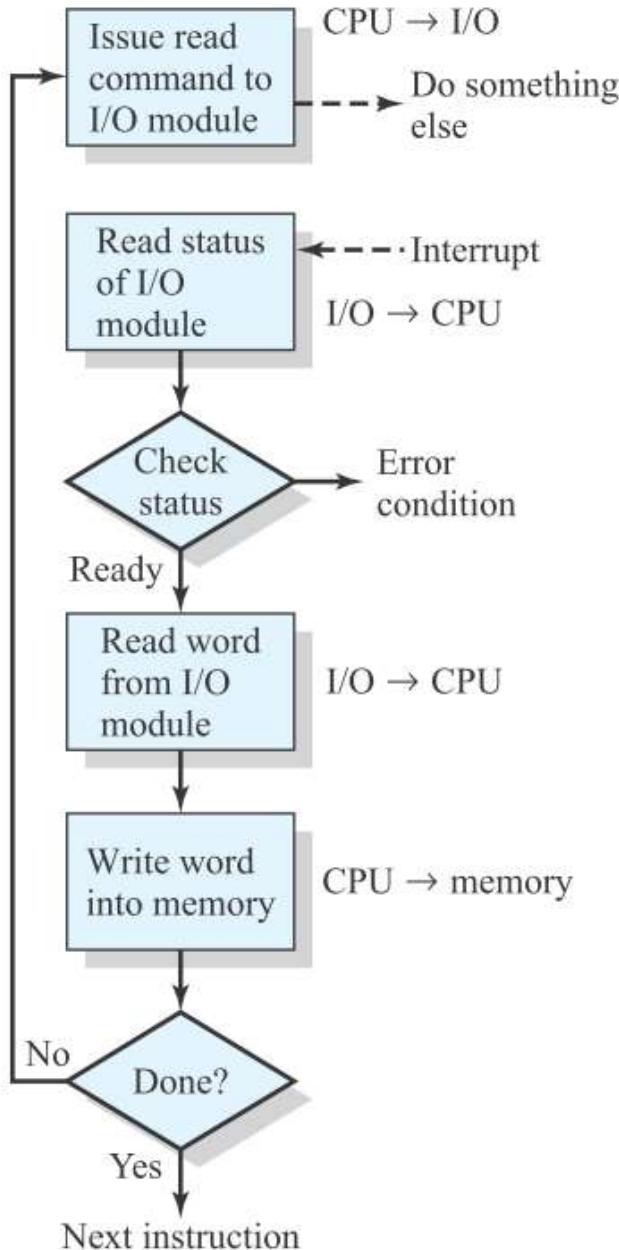
Programmed I/O requires that the instruction set of the processor includes dedicated commands utilising dedicated flags and registers in order to control the devices.

The commands are of the following kind:

| Element  | Description                                                                               |
|----------|-------------------------------------------------------------------------------------------|
| Control  | Used to activate an external device and tell it what to do.                               |
| Status   | Used to test various status conditions associated with an I/O module and its peripherals. |
| Transfer | Used to read and/or write data between processor registers and external devices.          |

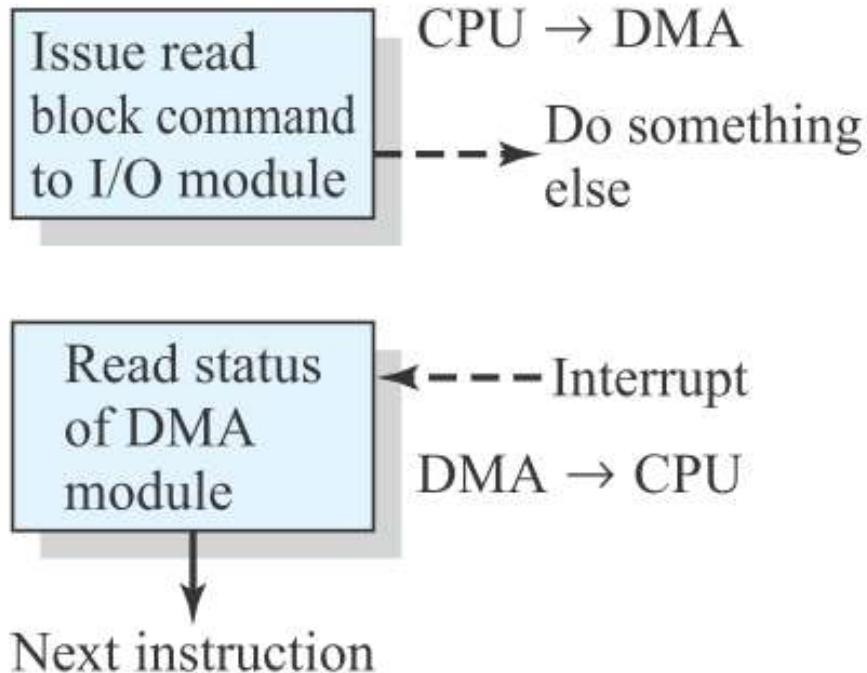
MINUS

- ▶ Processor is busy



## Interrupt-driven I/O

|       |                                                                                               |
|-------|-----------------------------------------------------------------------------------------------|
|       |                                                                                               |
| PLUS  | ➤ No busy waiting                                                                             |
| MINUS | ➤ Overhead to handle (multiple) interrupts<br>➤ Data has to pass processor from I/O to memory |



## Direct Memory Access (DMA)

|       |                                                                                                                                                                               |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|       |                                                                                                                                                                               |
| PLUS  | <ul style="list-style-type: none"> <li>➤ Processor is completely out of the loop</li> <li>➤ Not limited by the processor</li> <li>➤ Reduce usage of the system bus</li> </ul> |
| MINUS | <ul style="list-style-type: none"> <li>➤ DMA module needed</li> </ul>                                                                                                         |

In case of DMA the processor delegates the whole task to a dedicated component, the DMA module. It just sends the following information to it:

- Whether **to read or write**
- Which **I/O device** is involved
- **Starting position** in the main memory
- **Size of data** to move (how many words).

# Betriebssysteme / Operating Systems

## Meltdown attack using out-of-order-execution and cache

SS 2022

Prof. Dr.-Ing. Holger Gräßner

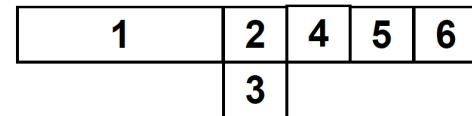
## Execution of commands: In order / out of order

- (1)  $r1 \leftarrow r4 / r7$  assume division takes a lot of cycles  
(2)  $r8 \leftarrow r1 + r2$   
(3)  $r5 \leftarrow r5 + 1$   
(4)  $r6 \leftarrow r6 - r3$   
(5)  $r4 \leftarrow r5 + r6$   
(6)  $r7 \leftarrow r8 * r4$

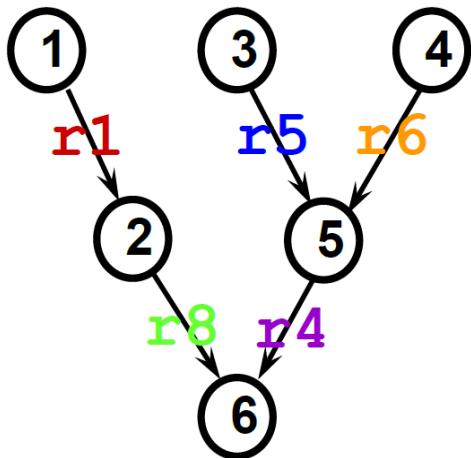
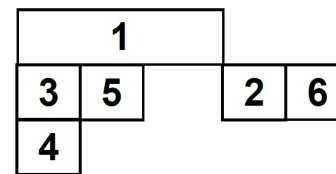
In-order execution



In-order (2-way superscalar)



Out-of-order execution



### Process 1

Symbolic machine code

Assumption: Cache block size is 0x1000.

```
01 Div D0 47.11          // divide register D0 by 47.11
02 test_rights            // test access rights: Zero if denied
03 jmp_zero LABEL1        // jump to label, if permission denied
04 load D1 SecretByte    // load SecretByte to register D1
05 mul D1 0x1000          // multiply Register D1 by 0x1000
06 transfer D1 A1         // data register D1 → adress register A1
07 load D2 (A1)           // load data from RAM adress given by A1
08 nop
09 nop
10 nop
LABEL1: 11 nop
```

### Process 1

Symbolic machine code

Assumption: Cache block size is 0x1000.

```
01 Div D0 47.11          // divide register D0 by 47.11
02 test_rights           // test access rights: Zero if denied
03 jmp_zero LABEL1       // jump to label, if permission denied
04 load D1 SecretByte // load SecretByte to register D1
05 mul D1 0x1000         // multiply Register D1 by 0x1000
06 transfer D1 A1        // data register D1 → address register A1
07 load D2 (A1)          // load data from RAM address given by A1
08 nop                  // side effect:
09 nop                  // the page containing(SecretByte * 0x1000)
10 nop                  // is now in the cache memory!
LABEL1: 11 nop
```

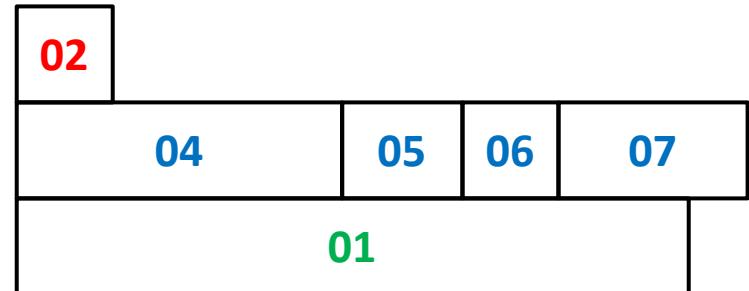
# Process 1

Attempt: Try to get the content of SecretByte.

Symbolic machine code

Assumption: Cache block size is 0x1000.

Speculative out of order execution



Processor runs speculative

```

00 flush_cache           // cache is now empty
01 Div D0 47.11          // divide register D0 by 47.11
02 test_rights            // test access rights: Zero if denied
03 jmp_zero LABEL1        // jump to label, if permission denied
04 load D1 SecretByte    // load SecretByte to register D1
05 mul D1 0x1000          // multiply Register D1 by 0x1000
06 transfer D1 A1         // data register D1 → address register A1
07 load D2 (A1)           // load data from RAM address given by A1
08 nop                   // side effect:
09 nop                   // the page containing(SecretByte * 0x1000)
10 nop                   // is now in the cache memory!
LABEL1: 11 nop           // even, if permission to access SecretByte has never been given!

```

# Process 2

Attempt: Use the cache as a side channel to get the content of `SecretByte`.

Start this process after Process 1.

C-Code

Assumption: Cache block size is 0x1000.

```
for (i := 0; i < 255; i++) {  
    start_timer();  
    dummy := *(i * 0x1000);      // access a specific RAM adress / RAM page  
    stop_timer();  
    AccessTime := timer.value;  
    if (AccessTime < c) {        // short access time to this RAM page?  
        break;                  // Yes: this RAM page was obviously in cache!  
                                // Therefore: i == SecretByte  
                                // Reason: Process 1 loaded this specific  
                                // page in the cache - using SecretByte...  
    }  
}
```

# **Betriebssysteme / Operating Systems**

## **Operating system overview**

SS 2018

Prof. Dr.-Ing. Holger Gräßner

04 OS-BS 2018 operating system overview.pptx



## Operating System Overview

- OS Objectives and Functions
- The evolution of OSs
- Major achievements

The purpose of this section is to provide an overview how OS have evolved over time and to describe two (major) contemporary operating systems.

Name OS that you know so far?

# Operating System Overview

## OS Objectives and Functions

The evolution of OSs

Major achievements

# OS Objectives and Functions

- **Convenience**

An OS makes a computer more convenient to use

- **Efficiency**

An OS allows the computer system resources to be used in an efficient manner

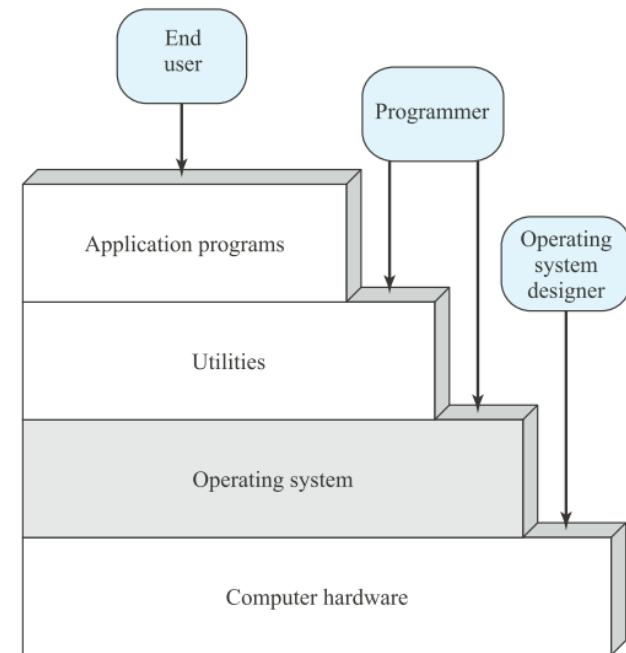
- **Ability to evolve**

An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service

## The OS as a User/Computer Interface

A computer system has several layers that provide different views for the different kind of users.

- As **end user** we just want to see our application programs and expect convenient and efficient use of them.
- As **programmer** we expect that to have access to the whole computer system but we do not want to be annoyed by the internal details of the underlying HW (which might change. . . ).



As **designer** of an OS we have access to everything of the computer and our job is to provide service to all of the other users.

## Services Provided by an OS (I)

| Service                    | Description                                                                                                                                                                               |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Program development        | Editors, debuggers (these are utility programs)                                                                                                                                           |
| Program execution          | OS handles initialisation, suspension, scheduling, and termination of (application) programs                                                                                              |
| Access to I/O devices      | OS provides a uniform interface that hides the details so that the programmer can access I/O devices using simple reads and writes.                                                       |
| Controlled access to files | The OS knows how files are stored on disks (depends on the nature of the device). This is hidden to the user. Moreover, it provides protection mechanisms to control access to the files. |
| System access              | It controls access to the system as a whole and to specific system resources. That provides protection of resources and data from unauthorised users and must resolve conflicts.          |

## Services Provided by an OS (II)

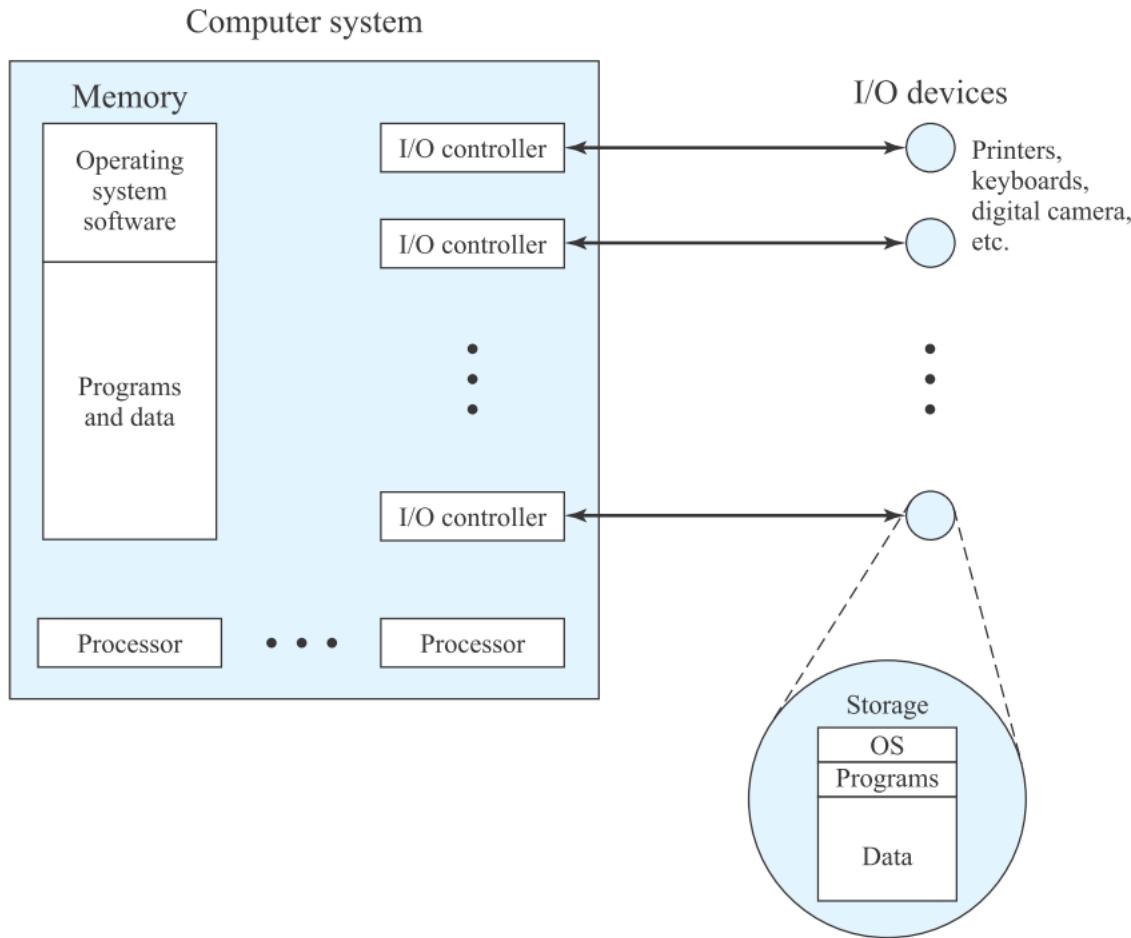
| Service                      | Description                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Error detection and response | <p>It has to handle various kinds of errors that might occur:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> internal and external HW errors like memory errors, device failures or malfunction.</li><li><input type="checkbox"/> SW failures like division by zero, attempt to access forbidden memory</li></ul> <p>These errors have to be handled in a reasonable way (depending on the nature of the error).</p> |
| Accounting                   | <p>It has to collect statistics of usage and to monitor performance parameters. Might be used for billing.</p>                                                                                                                                                                                                                                                                                                                           |

### The OS as Resource Manager

The OS manages the various resources of a computer systems. But it is in some sense an odd control system because:

- The OS functions in the same way as ordinary computer software; that is, it is a program or suite of programs executed by the processor.
- The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

# OS Objectives and Functions



## Ease of Evolution of an OS

A major OS will evolve over time for a number of reasons:



| Service                                     | Description                                                                                                                                                                                                                                        |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>HW upgrades,<br/>new types of<br/>HW</b> | If you buy a computer and use it over a period of 5+ years, it is very likely that you will extend it some day with peripheral devices that were not invented as you bought the computer. The OS has to cope with that and hence it must evolve... |
| <b>New services</b>                         | Once Microsoft extended its OS with an utility called "Windows Explorer"                                                                                                                                                                           |
| <b>Fixes</b>                                | <b>Any OS has faults!</b> If an OS would not be updated it will be vulnerable.                                                                                                                                                                     |

# Operating System Overview

OS Objectives and Functions

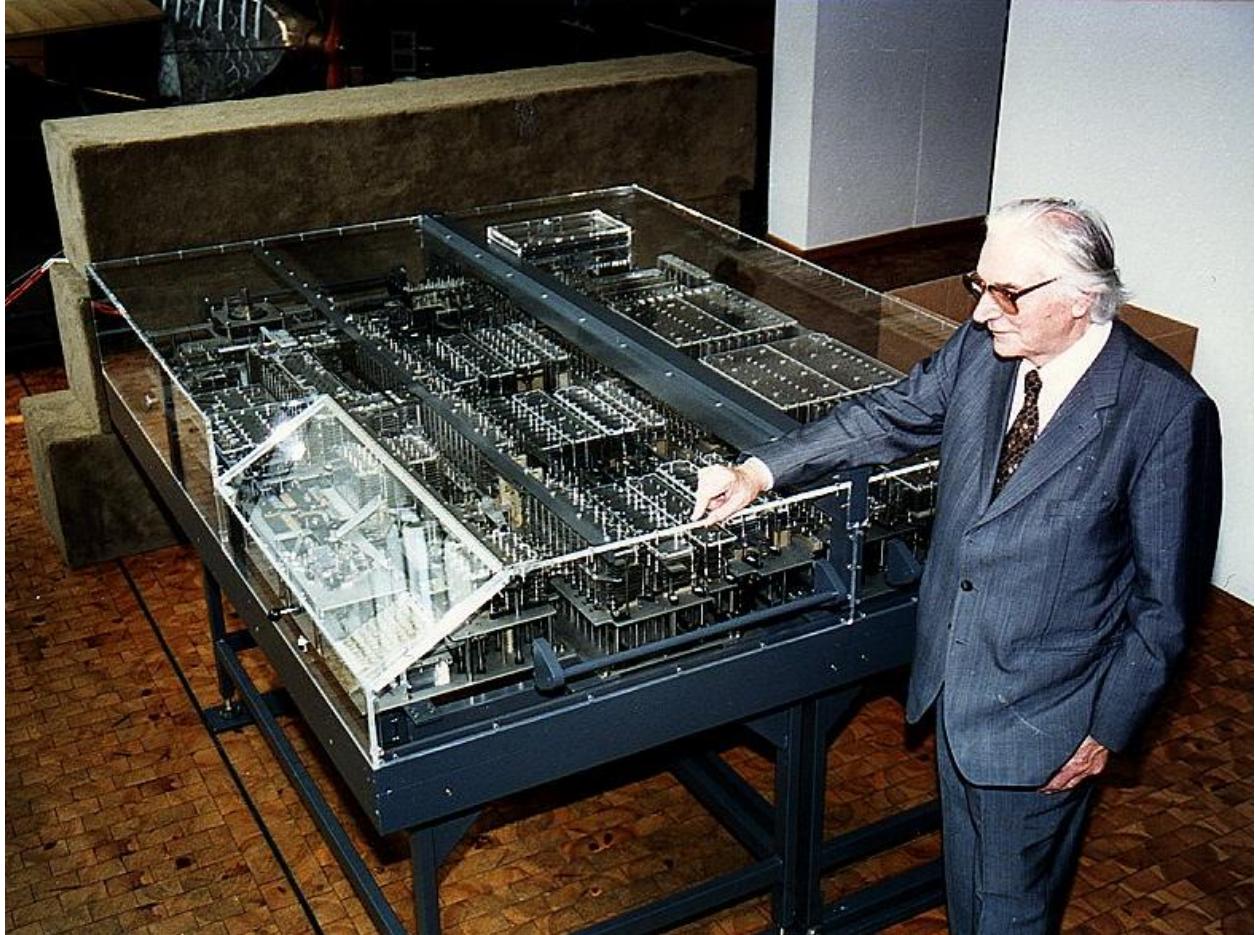
The evolution of OSs

Major achievements

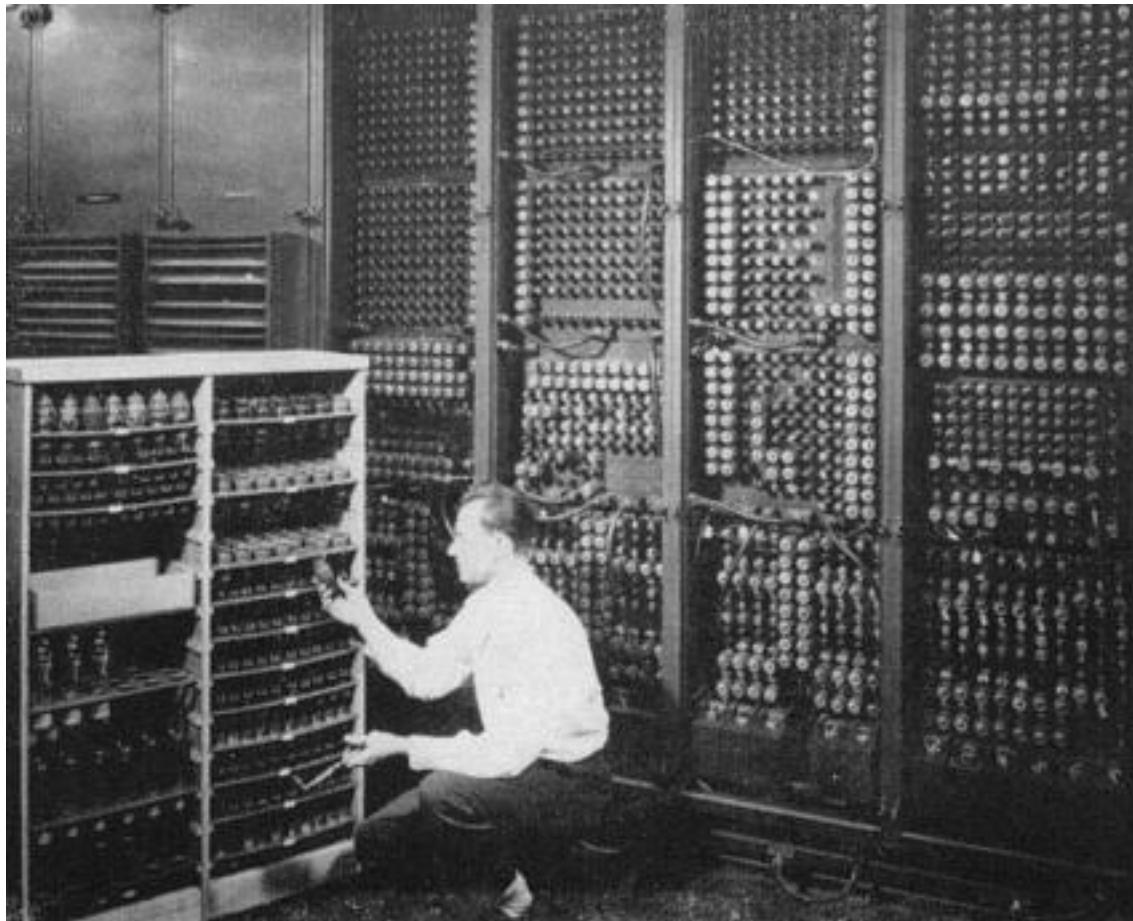
Let us review the history of  
operating systems a bit . . .

## Mechanical computer (1936)

Zuse Z1



## ENIAC (1936)



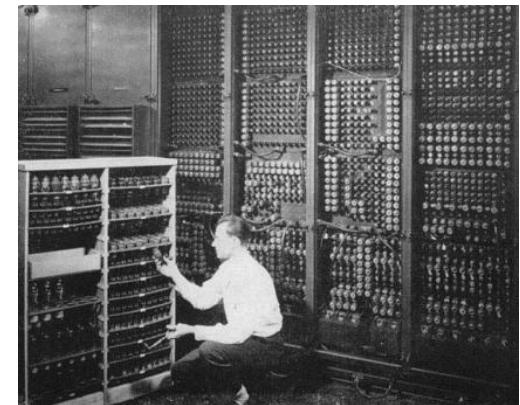
## Simple Batch Systems



## Serial Processing

The earliest computer – like the ENIAC – had nothing that could be called OS.

These computers had two major problems:



| Problem           | why                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Scheduling</b> | Users got time slots, e.g. 1 hours. In this time the program had to be loaded (on punch cards). However, programs terminated earlier or later. In both cases valuable time was wasted.                             |
| <b>Setup time</b> | Running a single program (“job”) required loading a compiler and program into memory, saving the compiled program, loading it and linking it. Each step required mounting and dismounting of tapes and card decks. |

## Simple Batch Systems

To overcome these problems, the first OS were introduced implementing simple batch systems, for instance for the IBM 7090 system in the early 1960s. The OS was called IBSYS.

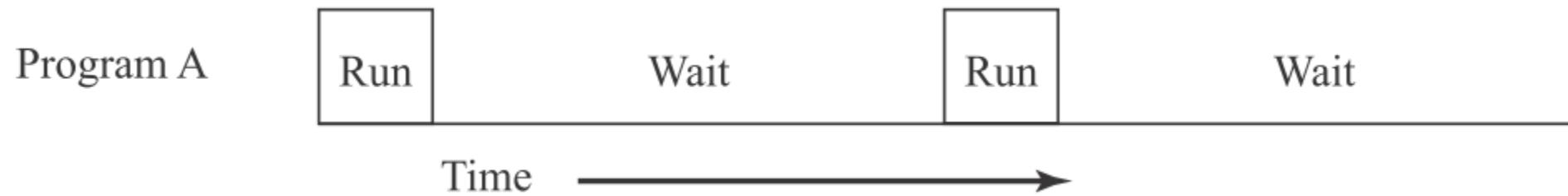


The core of the OS was a software called “**monitor**”. The task of this was to process a sequence of programs automatically. The programs were stored on cards or on tapes sequentially (“batch”). Programs were started by the monitor and when user program terminates it had to branch back to the monitor. The latter started the next program then.

→ **Lessons learned:** Memory protection needed, timeouts needed (requires timers), different modes useful.

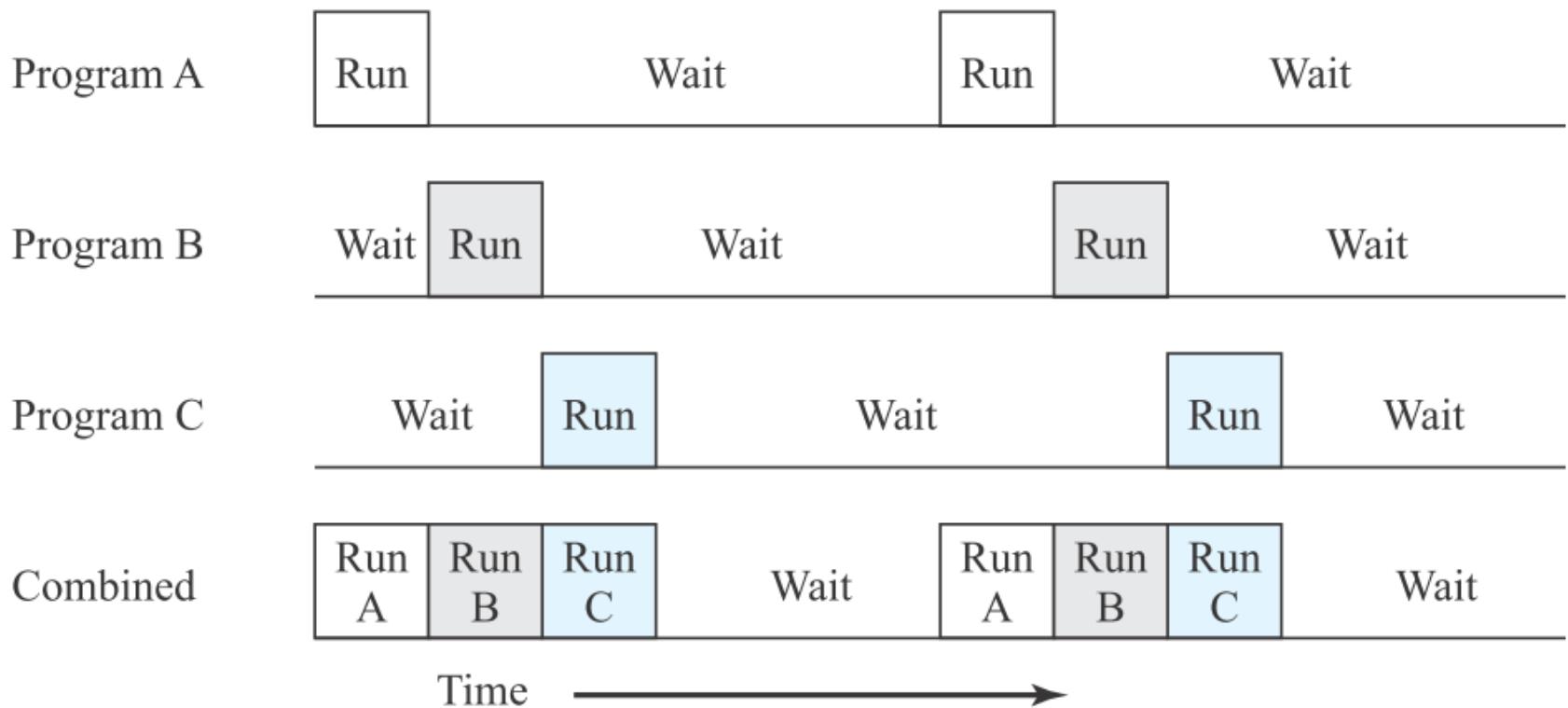
## Multiprogrammed Batch Systems (I)

Even in the early times of computing I/O devices were substantially slower than processors. In case of a simple batch systems, that were introduced to increase processor utilisation, the following behaviour was observed in the presence of I/O operations:

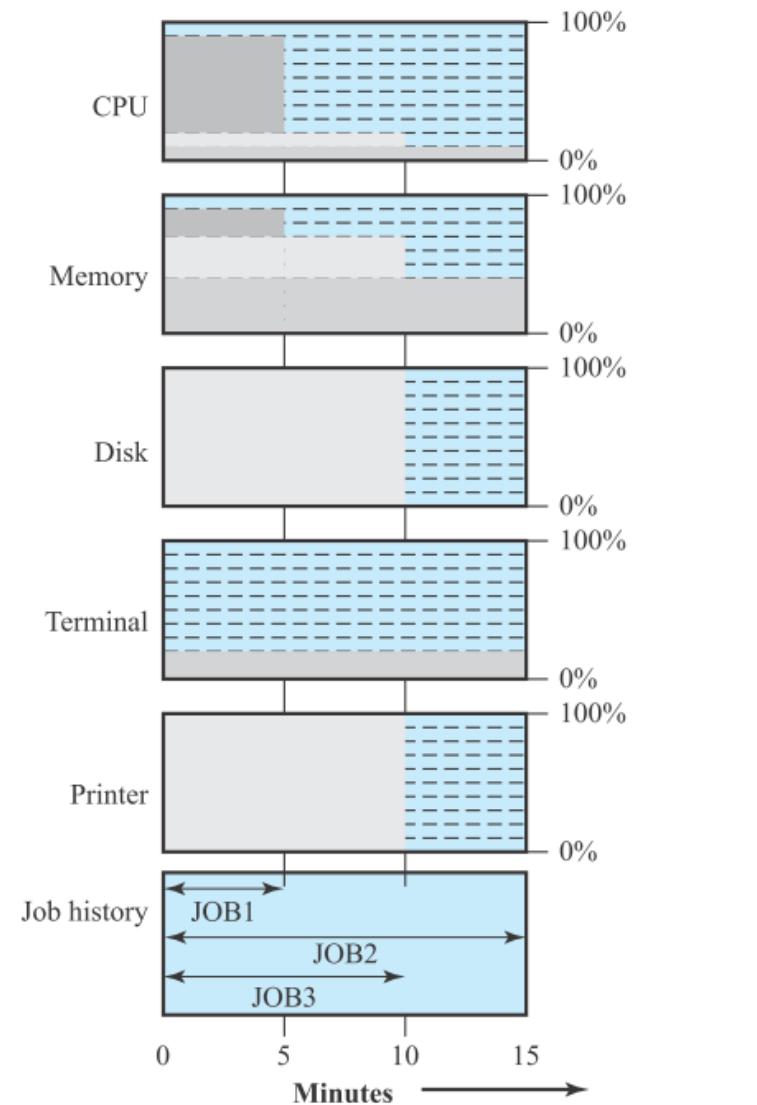
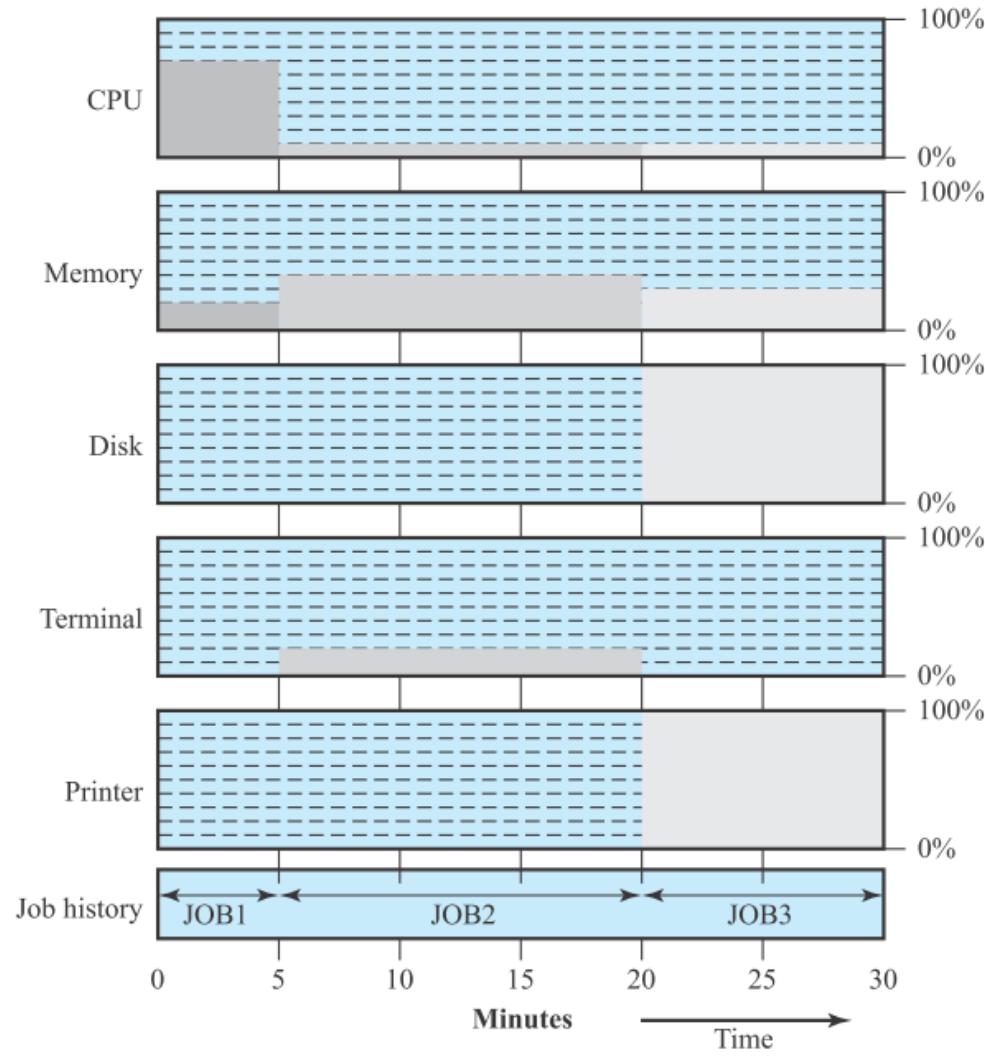


## Multiprogrammed Batch Systems (II)

**Multiprogramming** (aka. **Multitasking**) was invented to improve the situation provided that the main memory was sufficiently large to storage several programs:



## Multibprogrammed Batch Systems: Example



## Time-Sharing Systems

Multiprogrammed batch systems were built to maximize the processor utilisation.

However, in the good old times, many users shared one computer system at the same time via “**terminals**”.

In this case, processor utilisation is not the only issue.

Hence, **time sharing** systems have been developed.

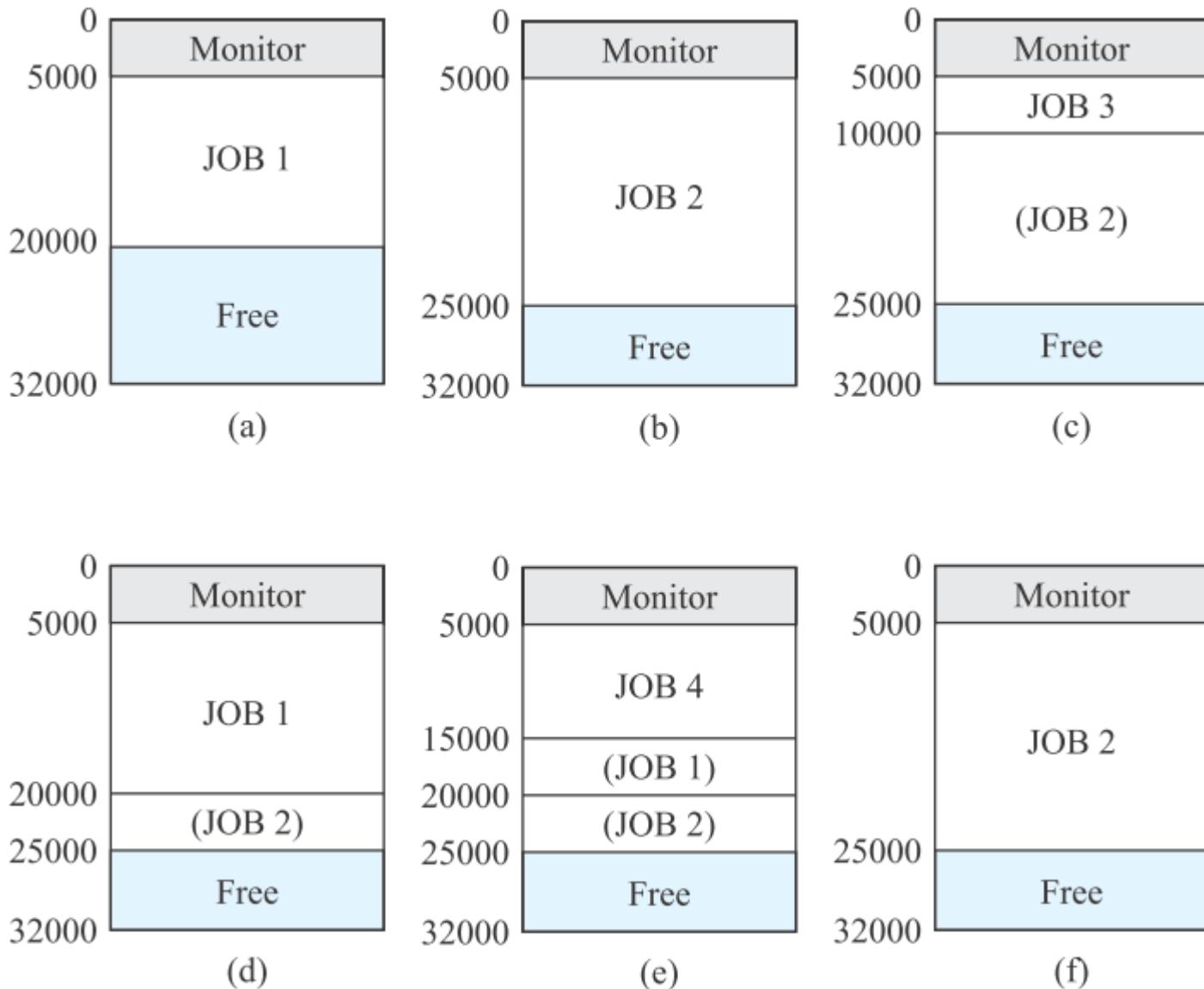
These systems allotted short **time slice** to each user.

Hence, from the user's perspective she got a system with a processor speed of  $1/n$  where  $n$  is the number of current users (ignoring OS overhead).

|                                          | Batch Multiprogramming                              | Time Sharing                     |
|------------------------------------------|-----------------------------------------------------|----------------------------------|
| Principal objective                      | Maximize processor use                              | Minimize response time           |
| Source of directives to operating system | Job control language commands provided with the job | Commands entered at the terminal |

## Time-Sharing Systems: CTSS

Earliest OS of that kind:  
CTSS  
(Compatible Time-Sharing System) in the early 1960s in IBM 7094 with up to 32 users.



## Time-Sharing Systems

### Lessons learned:

- Memory protection needed (now regarding users only)
- Resource allocation needed, mutual exclusive access to devices

# Operating System Overview

OS Objectives and Functions

The evolution of OSs

Major achievements

### Major Achievements

Building an OS that is state-of-the-art is not a trivial task

- Microsoft claimed that the project that developed VISTA was both more **complex and more expensive** than the APOLLO missions.

- Recently, a business institute estimated that LINUX is worth **1.000.000.000 Euro.**



## Major Achievements

The following topics are identified (by Stallings) to be the major achievements in the history of OSs:

1. Processes
2. Memory Management
3. Information Protection and Security
4. Scheduling and Resource Management
5. System Structure

# Betriebssysteme / Operating Systems

## Processes

SS 2020

Prof. Dr.-Ing. Holger Gräßner

05 OS-BS 2020 Processes.pptx

## Processes

**Fundamental** task of any modern OS is **process management**:

Resource allocation, information exchange, synchronization.

**Multiprogramming  
uniprocessor system :**

Execution of multiple processes  
**interleaved** in time.

**Multiprocessor system:**  
Simultaneous execution of  
multiple processes.

## Definition (Thread)

In a multithreaded system, the process retains the attributes of resource ownership, while the attribute of multiple, concurrent execution streams is a property of threads running within a process.

## Process Description and Control

What is a Process?

Process States

Process Description

Process Control

Execution of the Operating System

# Process Description and Control

What is a Process?

Process States

Process Description

Process Control

Execution of the Operating System

## Processes

There are various attempts to define the term “**process**” in the literature:

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

We have two essential elements: code and data associated with that code.

– *At any given point in time, while the program is executing, this process can be uniquely characterized by a number of elements, including the following:*

## Process and process block

|                        |                               |                                                                                                           |
|------------------------|-------------------------------|-----------------------------------------------------------------------------------------------------------|
| Identifier             | <b>Identifier</b>             | (aka ‘Handle’) unique identifier associated with this process, to distinguish it from all other processes |
| State                  | <b>State:</b>                 | A status information (running, suspended, . . . )                                                         |
| Priority               | <b>Priority</b>               | relative to other processes                                                                               |
| Program Counter        | <b>Program counter</b>        | Address of the next instruction                                                                           |
| Memory Pointers        | <b>Memory pointers</b>        | pointer to program code, data, memory blocks allocated, . . .                                             |
| Context Data           | <b>Context data</b>           | data contained in the registers of the processor while the process is executed.                           |
| I/O Status Information | <b>I/O status information</b> | For example outstanding I/O requests, devices assigned, files in use, . . .                               |
| Accounting Information | <b>Accounting information</b> | processor time used, time limits, . . .                                                                   |
| •<br>•<br>•            |                               |                                                                                                           |

# Process Description and Control

What is a Process?

Process States

Process Description

Process Control

Execution of the Operating System

### Process States: Example (I)

Let us consider a simple example first.

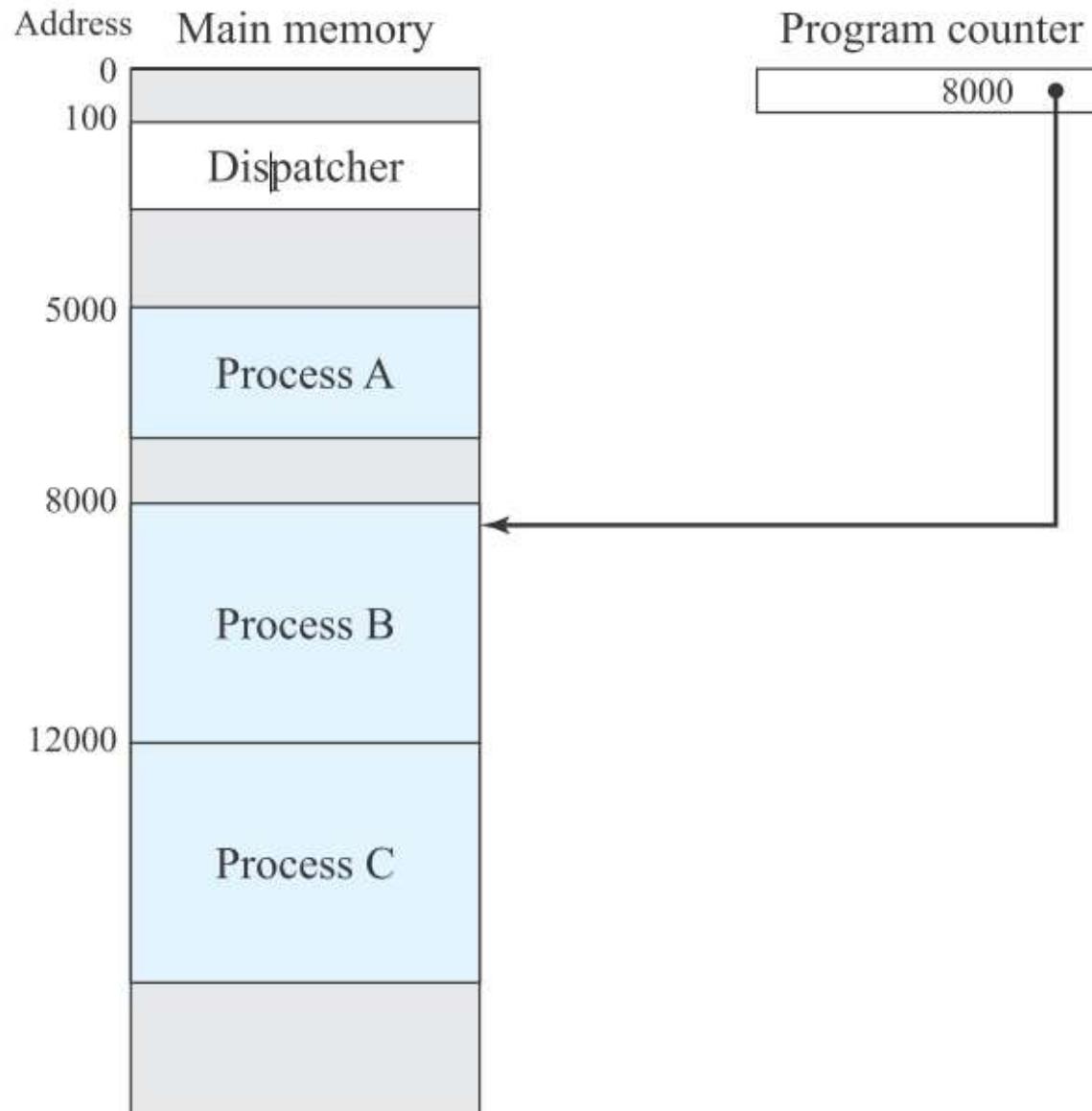
#### Example

We have three processes and all of them are fully loaded in main memory.

A **dispatcher** switches from one process to another.

The policy of the dispatcher is to allow a process to execute at most 6 instruction cycles before it interrupts and hands over to the next process.

## Process States: Example (II)



## Process States: Example (III)

Execution of instructions from the processes' point of view:

| Trace Proc A | Trace Proc B | Trace Proc C | Starting address |
|--------------|--------------|--------------|------------------|
| 5000         | 8000         | 12000        |                  |
| 5001         | 8001         | 12001        |                  |
| 5002         | 8002         | 12002        |                  |
| 5003         | 8003         | 12003        |                  |
| 5004         |              | 12004        |                  |
| 5005         |              | 12005        |                  |
| 5006         |              | 12006        |                  |
| 5007         |              | 12007        |                  |
| 5008         |              | 12008        |                  |
| 5009         |              | 12009        |                  |
| 5010         |              | 12010        |                  |
| 5011         |              | 12011        |                  |

## Process States: Example (IV)

Execution of instructions from the **processor's point of view**.

The processes are stopped and started by the dispatcher. The instructions of the dispatching routine are stored at memory location 100.

In this example, it seems to implement a **round robin** strategy.

Note that the 2<sup>nd</sup> process was stopped before the maximum number of instructions was reached because the process B started an I/O request.

|    |       |  |    |               |
|----|-------|--|----|---------------|
| 1  | 5000  |  | 27 | 12004         |
| 2  | 5001  |  | 28 | 12005         |
| 3  | 5002  |  |    | ----- Timeout |
| 4  | 5003  |  | 29 | 100           |
| 5  | 5004  |  | 30 | 101           |
| 6  | 5005  |  | 31 | 102           |
| 7  | 100   |  | 32 | 103           |
| 8  | 101   |  | 33 | 104           |
| 9  | 102   |  | 34 | 105           |
| 10 | 103   |  | 35 | 5006          |
| 11 | 104   |  | 36 | 5007          |
| 12 | 105   |  | 37 | 5008          |
| 13 | 8000  |  | 38 | 5009          |
| 14 | 8001  |  | 39 | 5010          |
| 15 | 8002  |  | 40 | 5011          |
| 16 | 8003  |  |    | ----- Timeout |
| 17 | 100   |  | 41 | 100           |
| 18 | 101   |  | 42 | 101           |
| 19 | 102   |  | 43 | 102           |
| 20 | 103   |  | 44 | 103           |
| 21 | 104   |  | 45 | 104           |
| 22 | 105   |  | 46 | 105           |
| 23 | 12000 |  | 47 | 12006         |
| 24 | 12001 |  | 48 | 12007         |
| 25 | 12002 |  | 49 | 12008         |
| 26 | 12003 |  | 50 | 12009         |
|    |       |  | 51 | 12010         |
|    |       |  | 52 | 12011         |
|    |       |  |    | ----- Timeout |

## Two-State Process Model

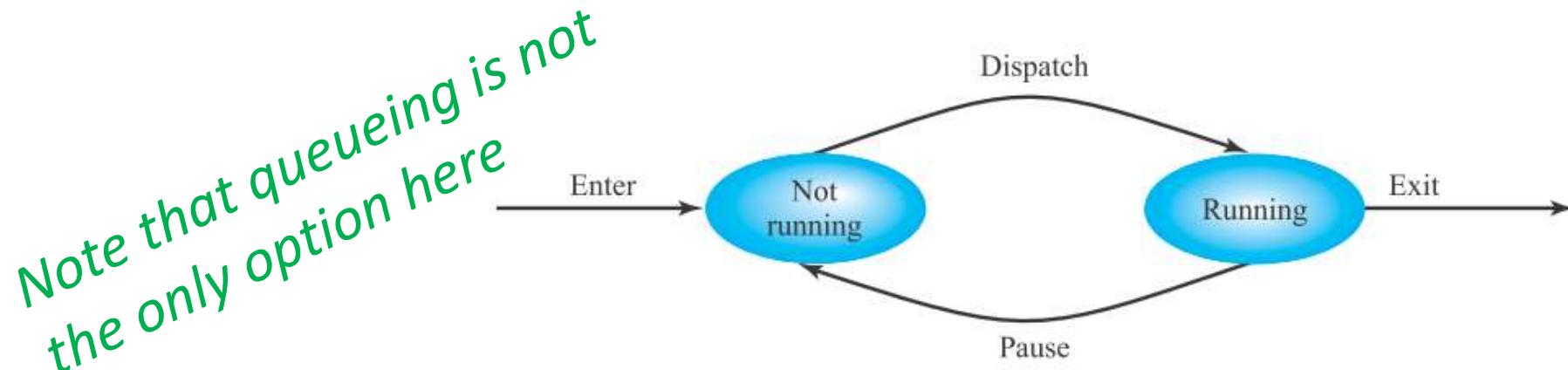
The simplest model of a process consists of two states:  
Either a process is **running** or **not running**.

The creation of a process by the OS is a task in which a process block for this process is created and initially this process is in state “not running”.

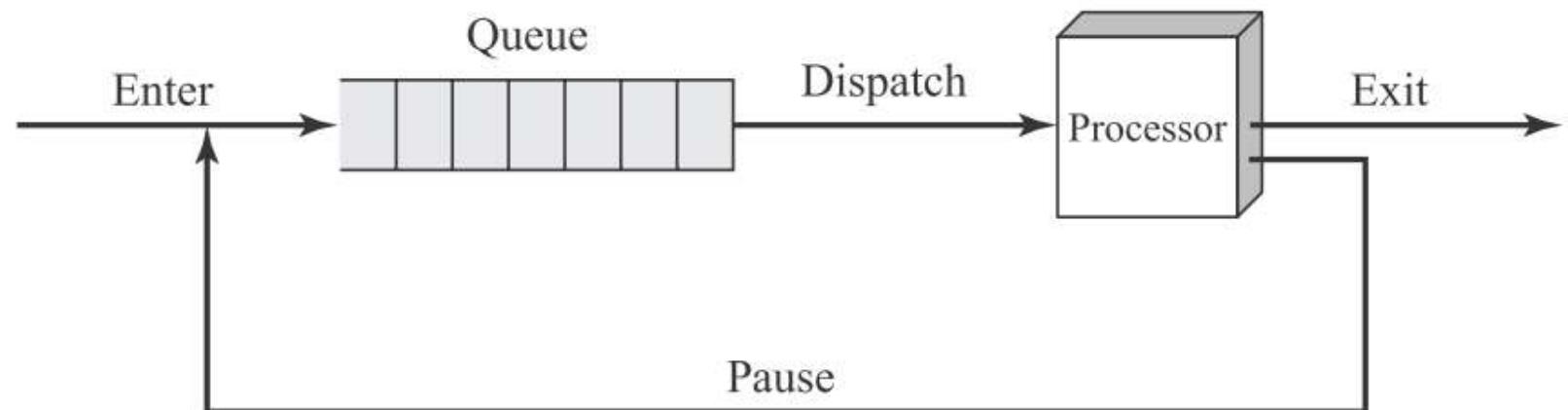
A dispatching routine will then select this process (from time to time) to execute instructions until it is interrupted again.

## Two-State Process Model: Overview

The state transition diagram:



OS: Queueing the processes



## Creation of Processes (I)

| Trigger                            | Description                                                                                                                                                                       |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| New batch job                      | The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands. |
| Interactive logon                  | A user at a terminal logs on to the system.                                                                                                                                       |
| Created by OS to provide a service | The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).                             |
| Spawned by existing process        | For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.                                                           |

## Termination of Processes (I)

| Cause               | Description                                                                                                                                                                                                                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Normal completion   | The process executes an OS service call to indicate that it has completed running.                                                                                                                                                                                                                                                      |
| Time limit exceeded | The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input. |
| Memory unavailable  | The process requires more memory than the system can provide.                                                                                                                                                                                                                                                                           |
| Bounds violation    | The process tries to access a memory location that it is not allowed to access.                                                                                                                                                                                                                                                         |

## Termination of Processes (II)

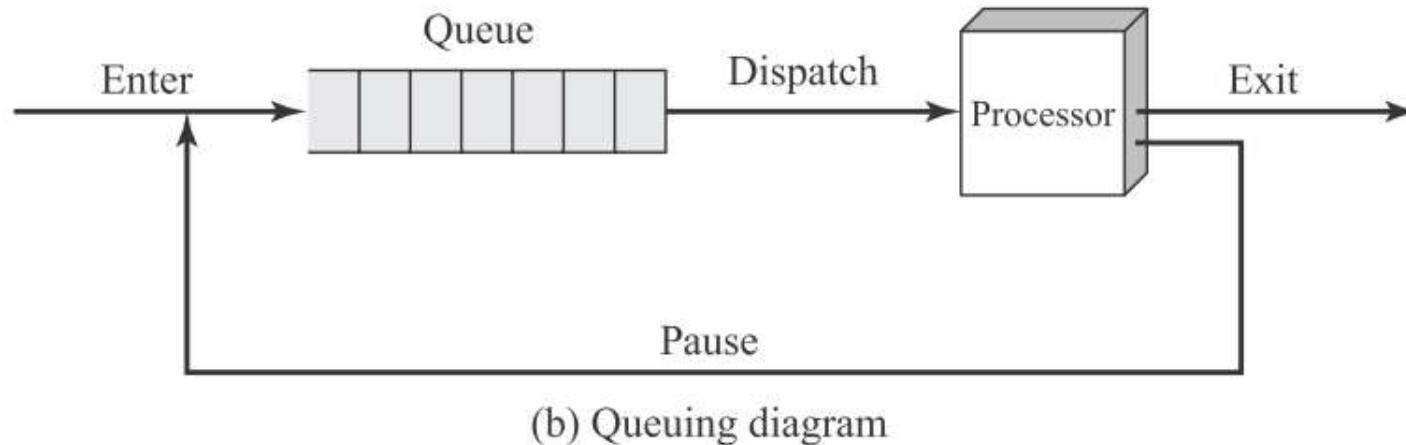
| Cause            | Description                                                                                                                                                                                                                                                                |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Protection error | The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.                                                                                                    |
| Arithmetic error | The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.                                                                                                                                  |
| Time overrun     | The process has waited longer than a specified maximum for a certain event to occur.                                                                                                                                                                                       |
| I/O failure      | An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer). |

## Termination of Processes (III)

| Cause                       | Description                                                                                                                                  |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Invalid instruction         | The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data). |
| Privileged instruction      | The process attempts to use an instruction reserved for the operating system.                                                                |
| Data misuse                 | A piece of data is of the wrong type or is not initialized.                                                                                  |
| Operator or OS intervention | For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).                        |
| Parent termination          | When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.                              |
| Parent request              | A parent process typically has the authority to terminate any of its offspring.                                                              |

## A Five-State Model

Consider the process queue of the previous two-state model again.



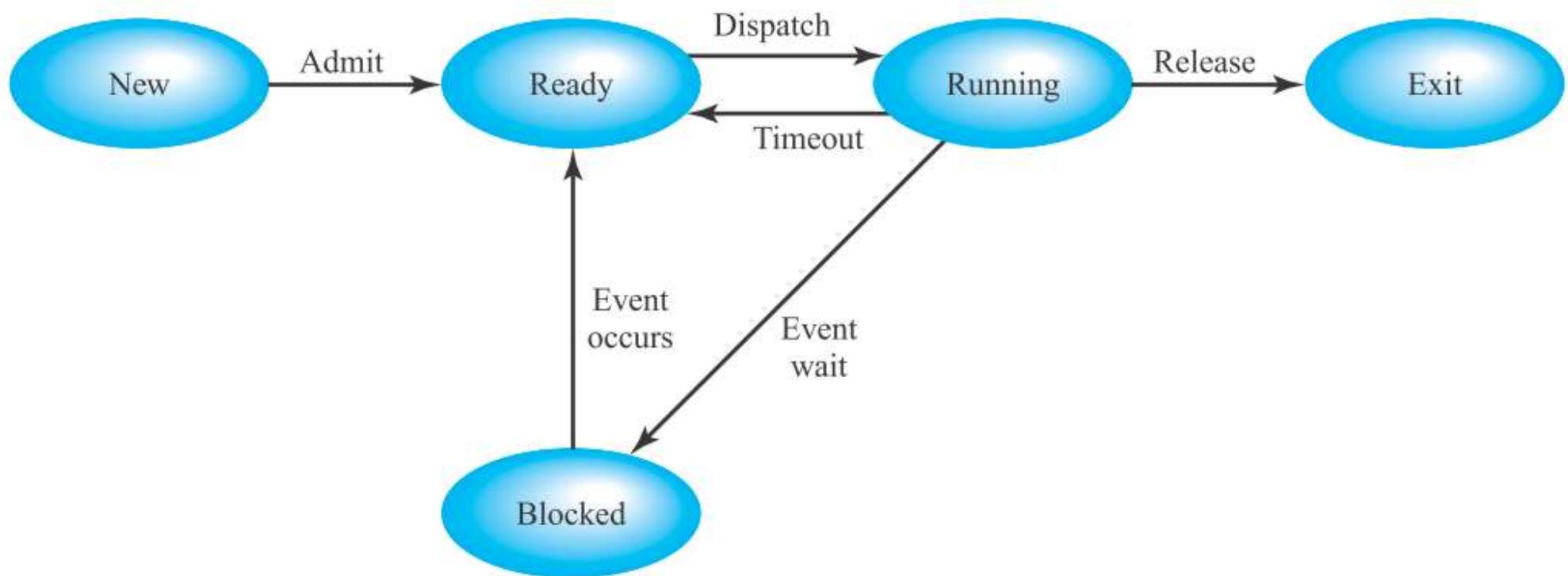
What can happen if the next process is selected, i.e. the oldest one in the queue?

The dispatcher might find a process that is just waiting for an I/O operation (and it still has to wait for that).

Hence, we should distinguish at least the “not running” processes into those which are ready to go and those which are blocked.

## A Five-State Model (I)

Here is a more sophisticated process model:

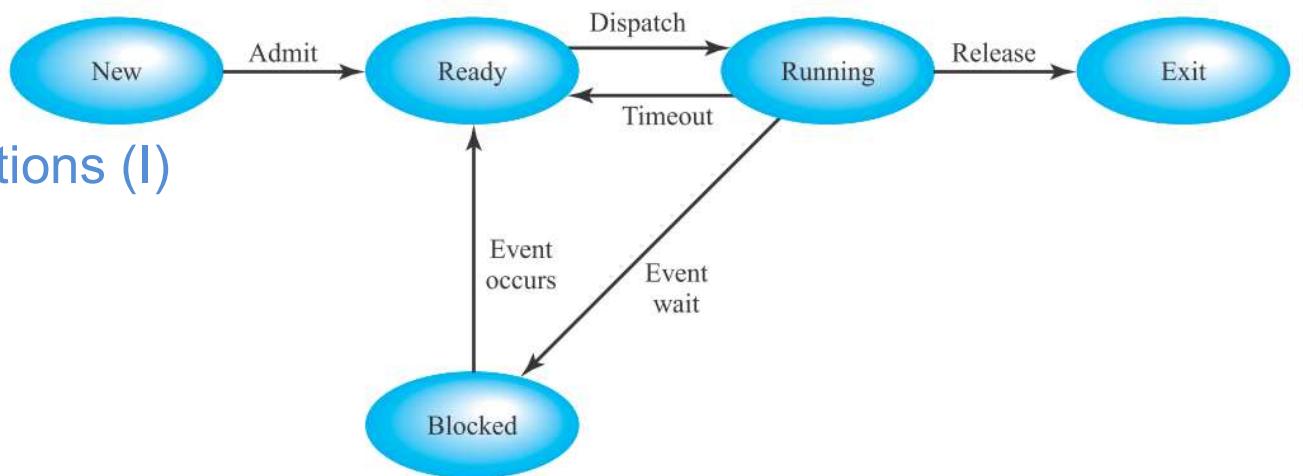


Here, the “not running” state is divided into the “ready” and “blocked” (aka. “waiting”). The states “New” and “Exit” are useful, too.

## A Five-State Model (II)

| State   | Description                                                                                                                                                                                                                 |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Running | The process that is currently running. For the moment, let us assume that we have a single processor system. Hence, at most one process at a time can be in this state.                                                     |
| Ready   | A process that is prepared to execute when given the opportunity.                                                                                                                                                           |
| Blocked | A process that cannot execute until some event occurs, such as a completion of an I/O operation.                                                                                                                            |
| New     | A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.<br>(Typically, the process block has been created but the code has not been load to the main memory yet). |
| Exit    | A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.                                                                           |

## Process states



### A Five-State Model: Transitions (I)

#### →New:

A new process is created to execute a program.

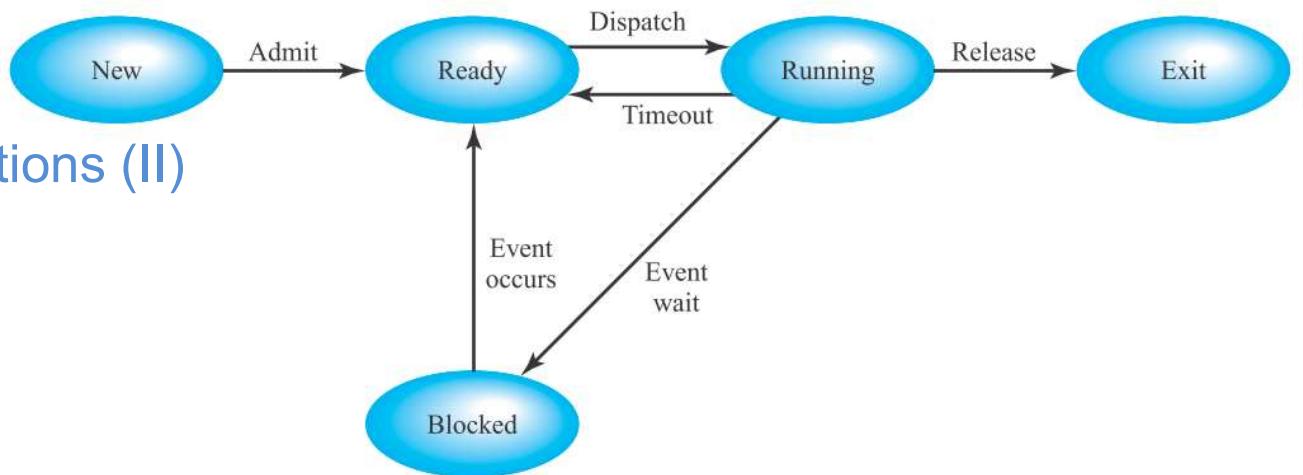
#### New→Ready:

If the OS can handle another process it moves one process from New to Ready. (No. of processes might be limited, memory might be limited).

**Ready→Running:** When it is time to select a process to run, the OS picks one process that can execute instructions. The question which process is selected is the topic of the chapter on Scheduling.

**Running→Exit:** A process has to be terminated or it indicates that it is finished. Refer to the table with reasons for process termination.

## Process states



### Running→Ready:

For instance, the OS detects that a process has reached its maximum of allowable time for uninterrupted execution.

Alternative reason: A process decides to be nice, ie. it voluntarily releases the control of the processor.

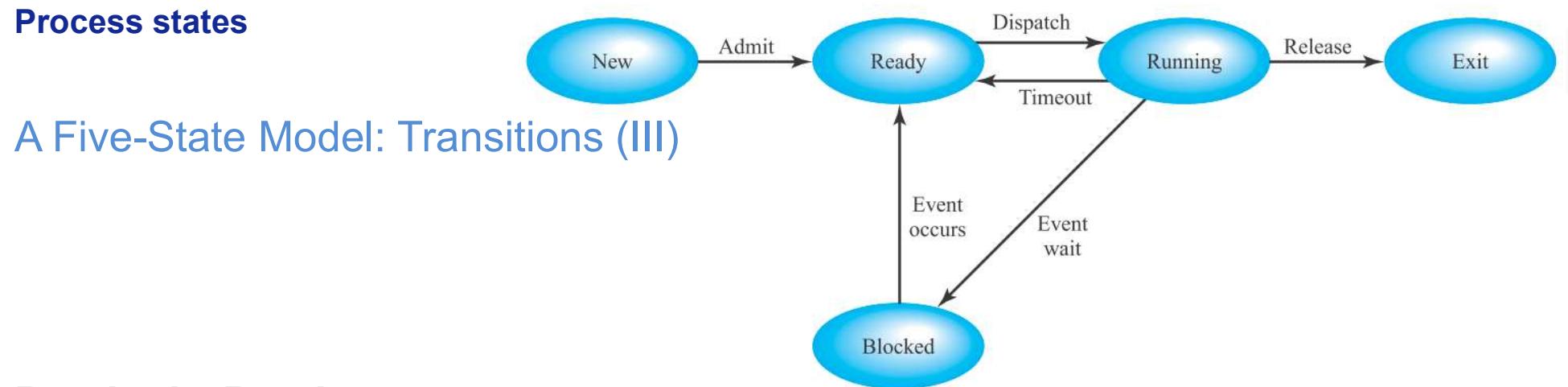
In OSs which process priority it might also happen that a process (B) of higher priority of the currently executed one (A) becomes ready. Then A is preempted.

### Running→Blocked:

The process requires an action for which it has to wait.

(I/O operation, memory allocation, system procedure call, . . . )

## Process states



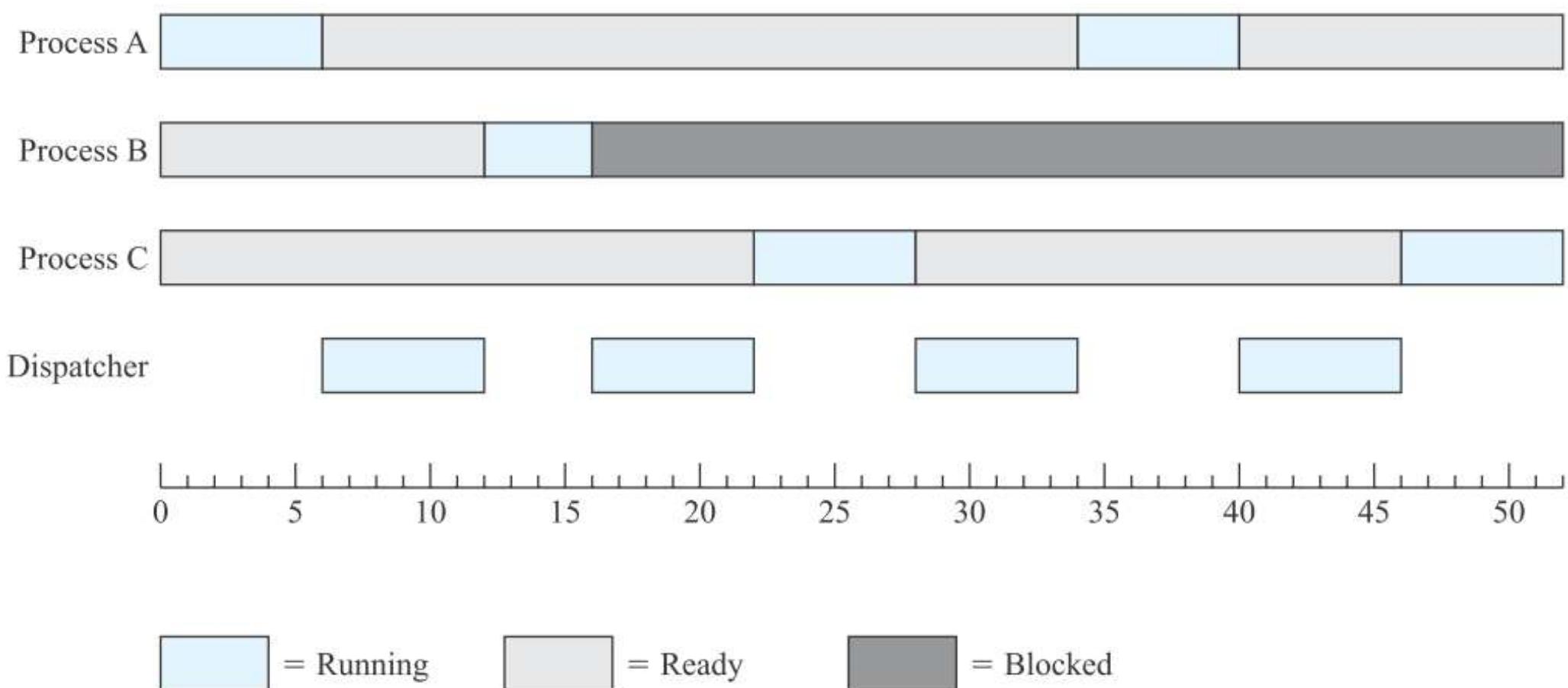
### Blocked→Ready:

A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.

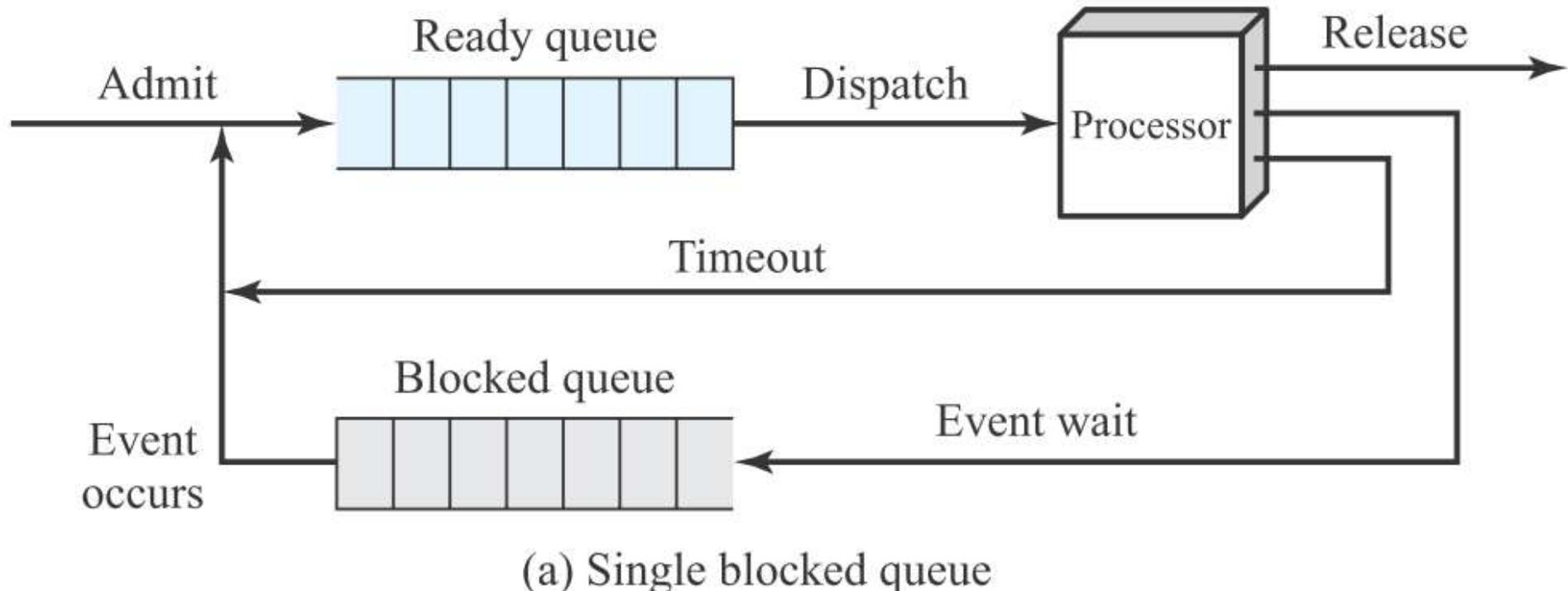
### Ready→Exit and Blocked→Exit: (not shown in the diagram!)

In some systems, a parent may terminate a child process at any time. Also, if a parent terminates, all its child processes may be terminated.

## Example revisited

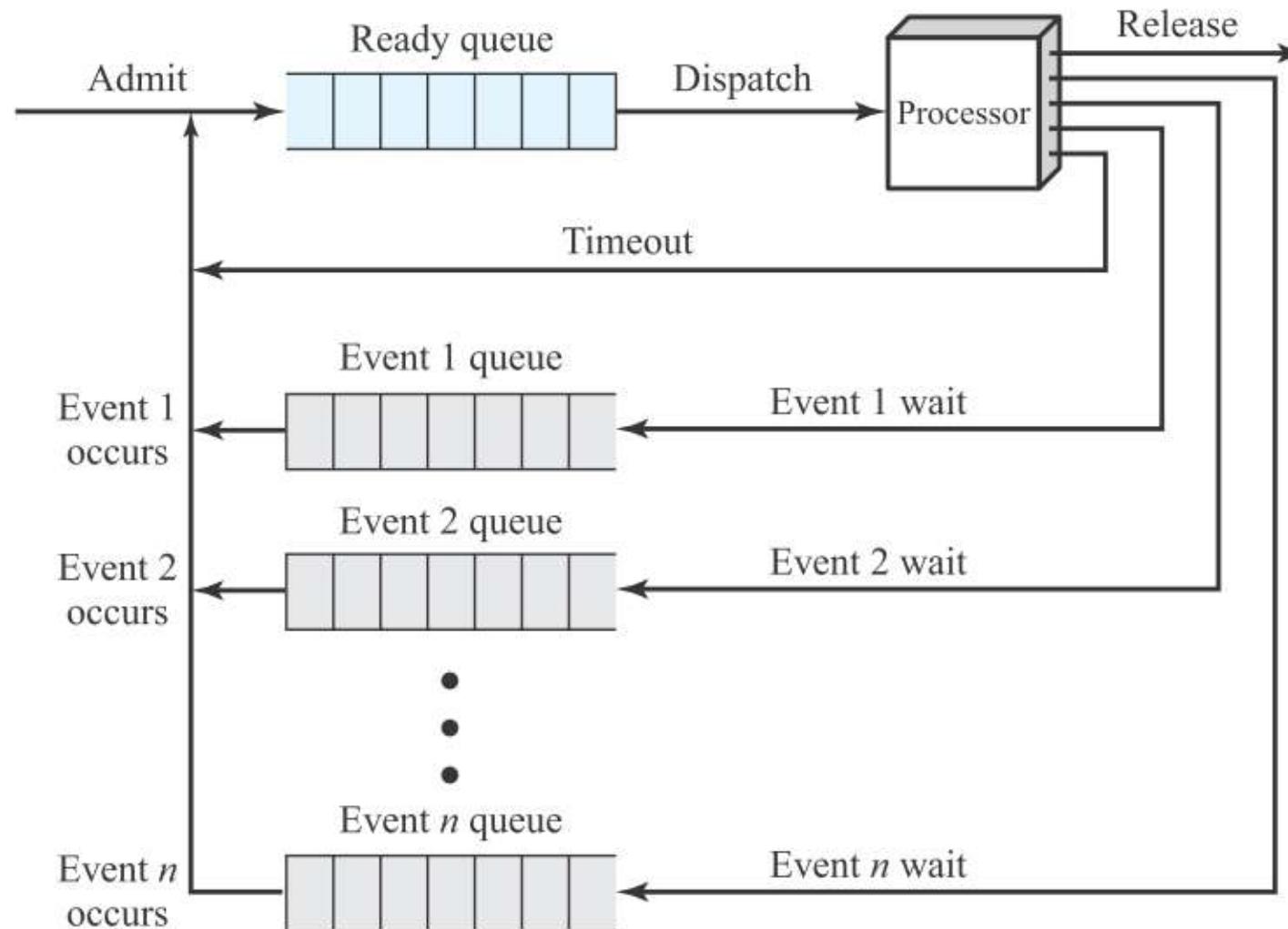


## Five-state Model: Possible Implementation (I)



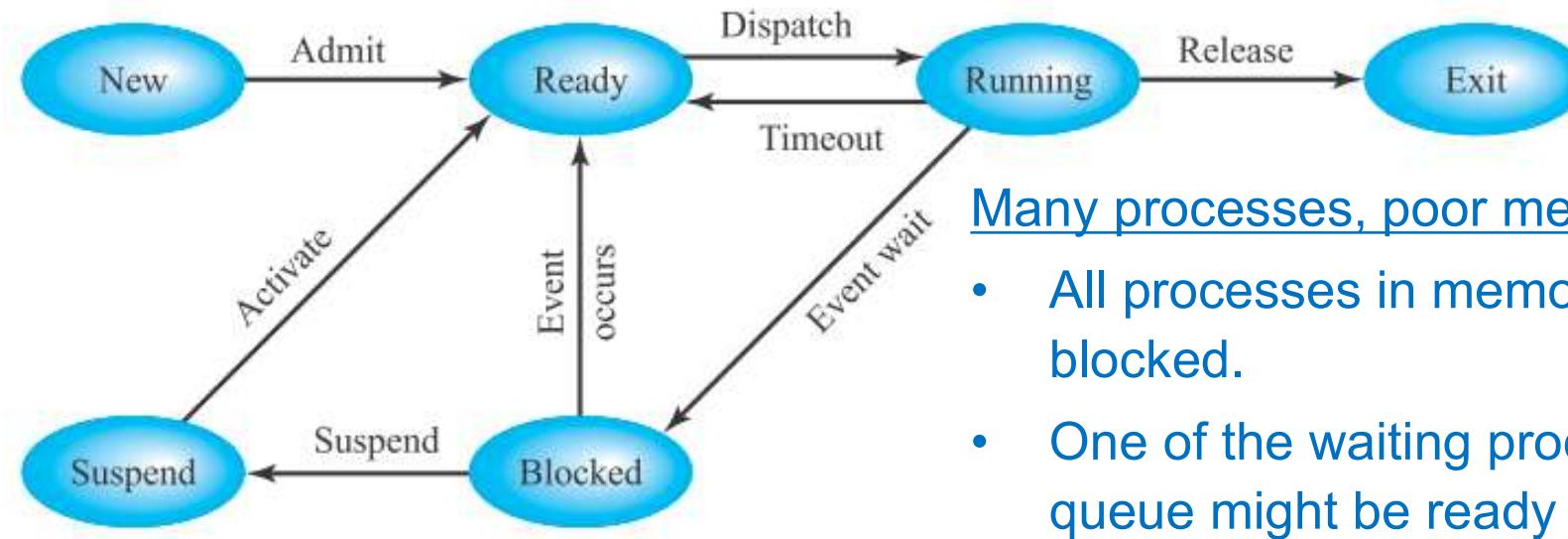
Might be inefficient if there are hundreds of blocked processes.

## Five-state Model: Possible Implementation (I)



Even more sophisticated: Queue for each priority level.

## Process State Model with Suspended Processes (I)



Many processes, poor memory:

- All processes in memory might be blocked.
- One of the waiting processes in the queue might be ready in the meantime...

A process is **suspended** (new state) if no process is ready or running.

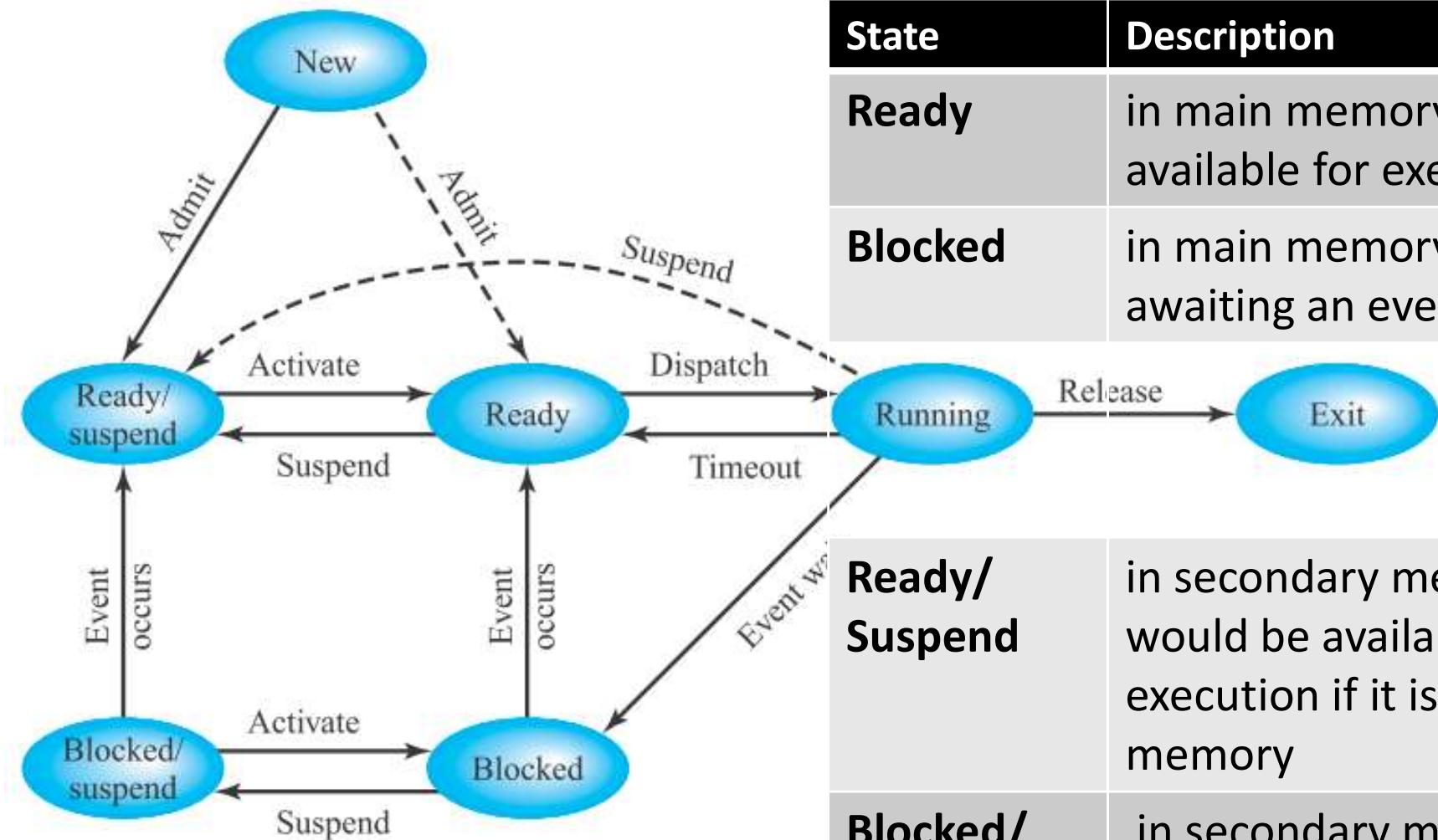
Then one (or more) blocked process is kicked out of memory to make room for a suspended process or a new process.

What are the advantages  
of the options?

Picking a suspended  
process from the disk:

| PLUS  | ➤ Does not increase the workload    |
|-------|-------------------------------------|
| MINUS | ➤ The suspended process was blocked |

## Process State Model with Suspended Processes (II)

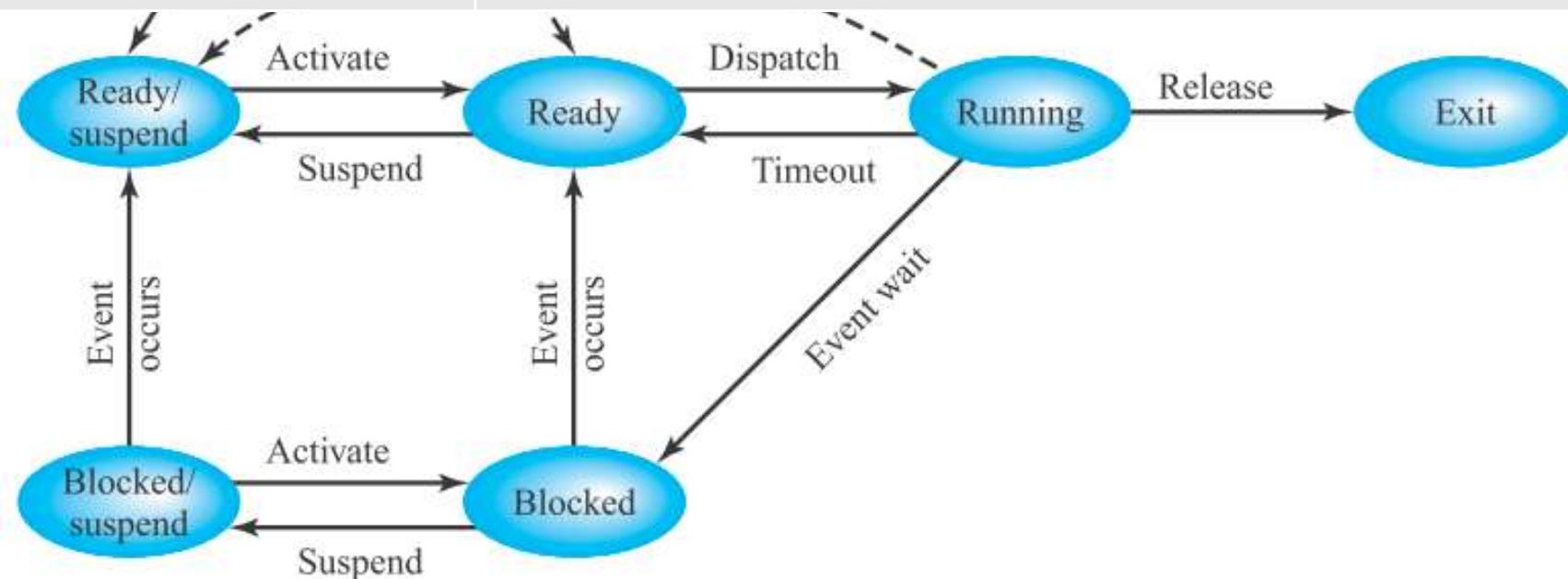


| State                       | Description                                                                      |
|-----------------------------|----------------------------------------------------------------------------------|
| <b>Ready</b>                | in main memory and available for execution                                       |
| <b>Blocked</b>              | in main memory and awaiting an event                                             |
| <b>Ready/<br/>Suspend</b>   | in secondary memory and would be available for execution if it is in main memory |
| <b>Blocked/<br/>Suspend</b> | in secondary memory and awaiting an event                                        |

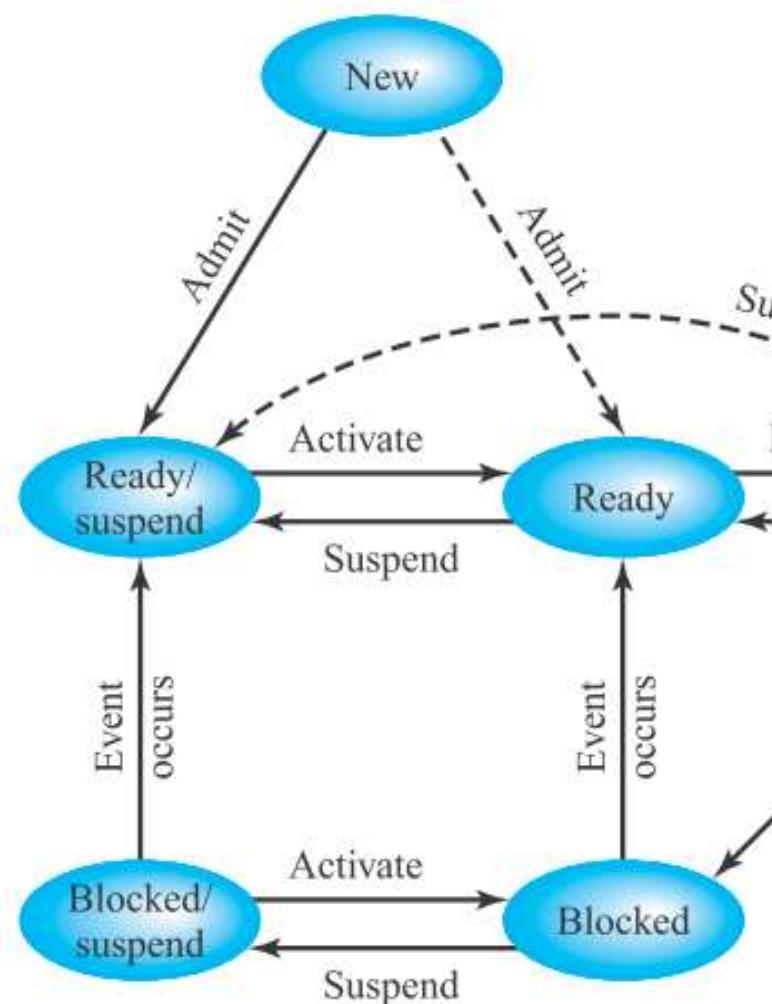
Note that we have a two-dimensional state of a process here

## Process State Transitions with Suspended Processes (I)

| State                           | Description                                                                                                                                            |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| Blocked → Blocked/Suspend:      | If room in memory is needed (for several reason), then (at least) one process is swapped onto disk.                                                    |
| Blocked/Suspend → Ready/Suspend | An event occurs a suspended process is waiting for. The OS must have access to this information, i.e. the process block must contain this information. |

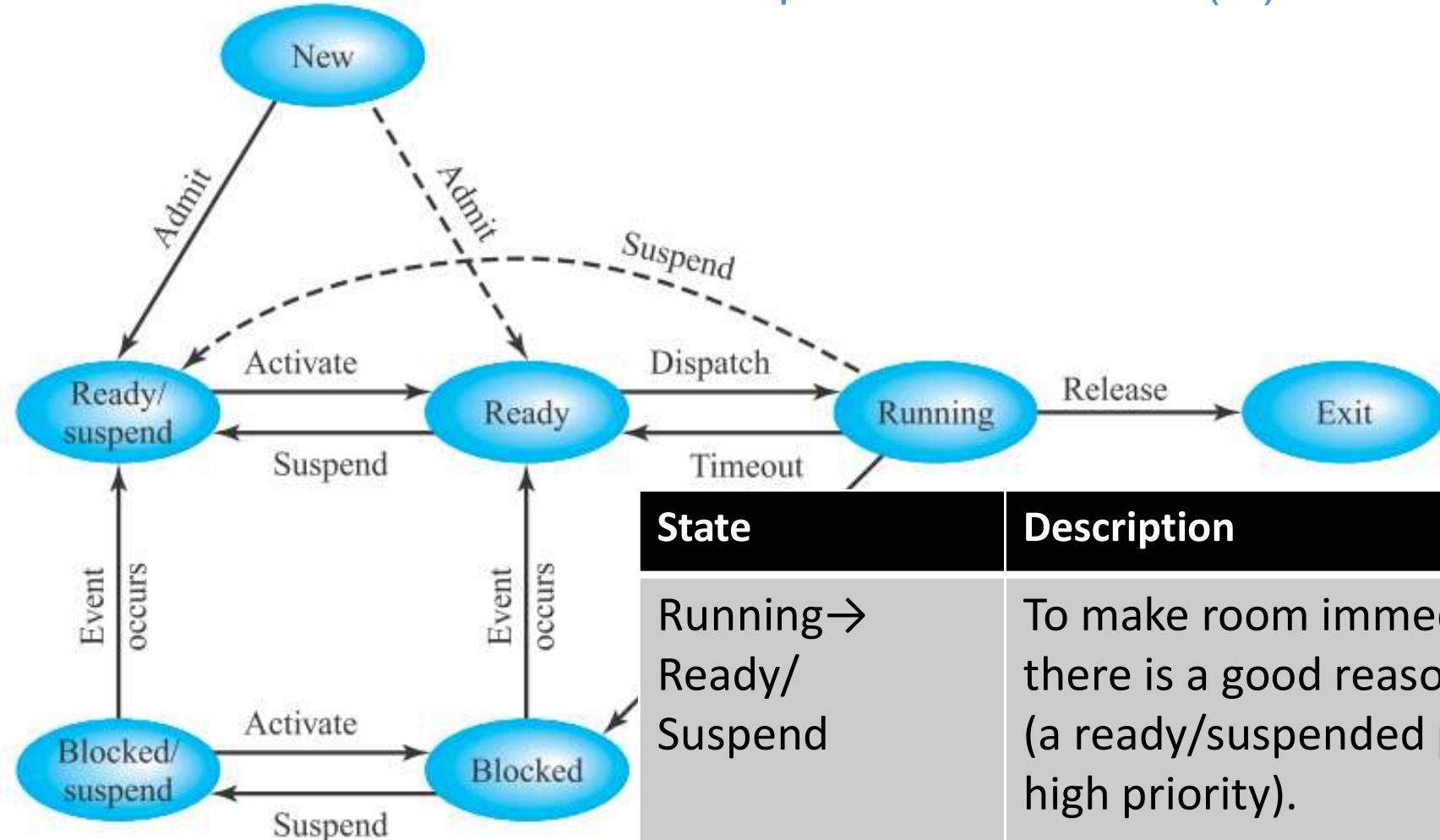


## Process State Transitions with Suspended Processes (II)



| State                       | Description                                                                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| New → Ready / Suspend       | Trade-off between having many ready process and memory consumption.                                                                  |
| Blocked / Suspend → Blocked | Poor design? However, it might be reasonable if the OS has reason to believe that this process (of high priority) become ready soon. |

## Process State Transitions with Suspended Processes (III)



| State                       | Description                                                                                      |
|-----------------------------|--------------------------------------------------------------------------------------------------|
| Running → Ready/<br>Suspend | To make room immediately if there is a good reason (a ready/suspended process of high priority). |
| Any state → Exit            | Possible from any state!                                                                         |

## Suspension (I)

The definition of a suspended process is more general:

### Definition (Suspended Process)

A process is called **suspended** if it satisfies the following characteristics:

1. The process is not immediately available for execution.
2. The process may or may not be waiting on an event. If so, the occurrence of the blocking event does not enable immediate execution.
3. The process was placed in a suspended state by an agent (itself, a parent process, or the OS).
4. The process may not be removed from this state until the agent explicitly orders the removal.

## Suspension (II)

| Reason                          | Description                                                                                                                                                      |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Swapping</b>                 | The OS needs to release sufficient main memory to bring in a process that is ready to execute.                                                                   |
| <b>Other OS reason</b>          | The OS may suspend a background or utility process or a process that is suspected of causing a problem.                                                          |
| <b>Interactive user request</b> | A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.                                         |
| <b>Timing</b>                   | A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.         |
| <b>Parent process request</b>   | A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants. |

# Process Description and Control

What is a Process?

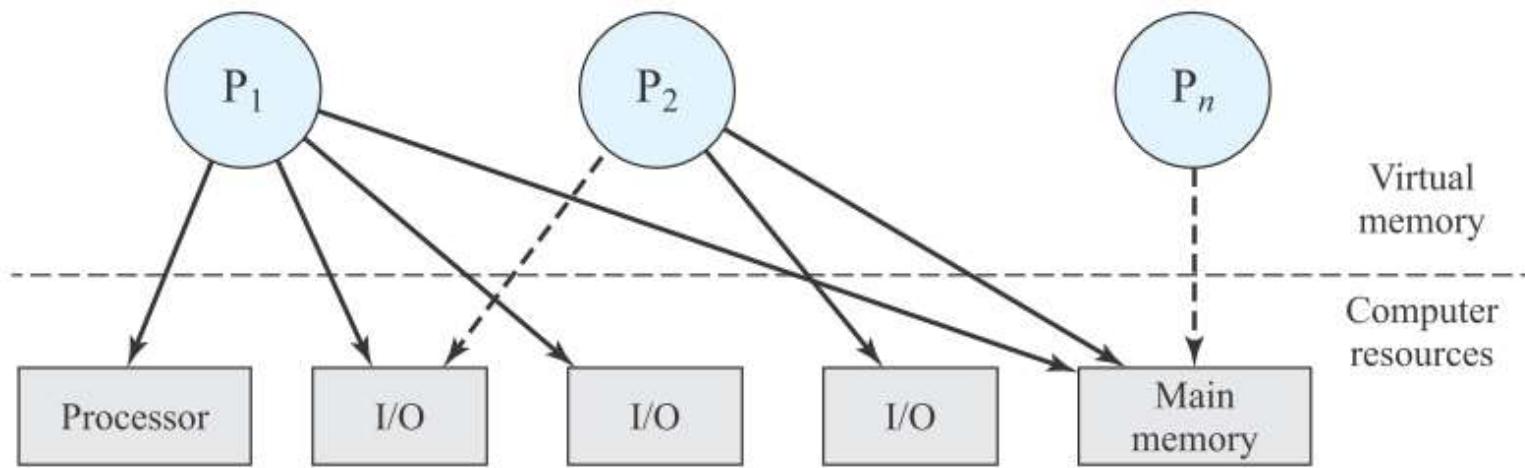
Process States

**Process Description**

Process Control

Execution of the Operating System

## Process Description



The picture above shows processes and resources. Arrows stand for allocation. It is a major task of the OS to manage

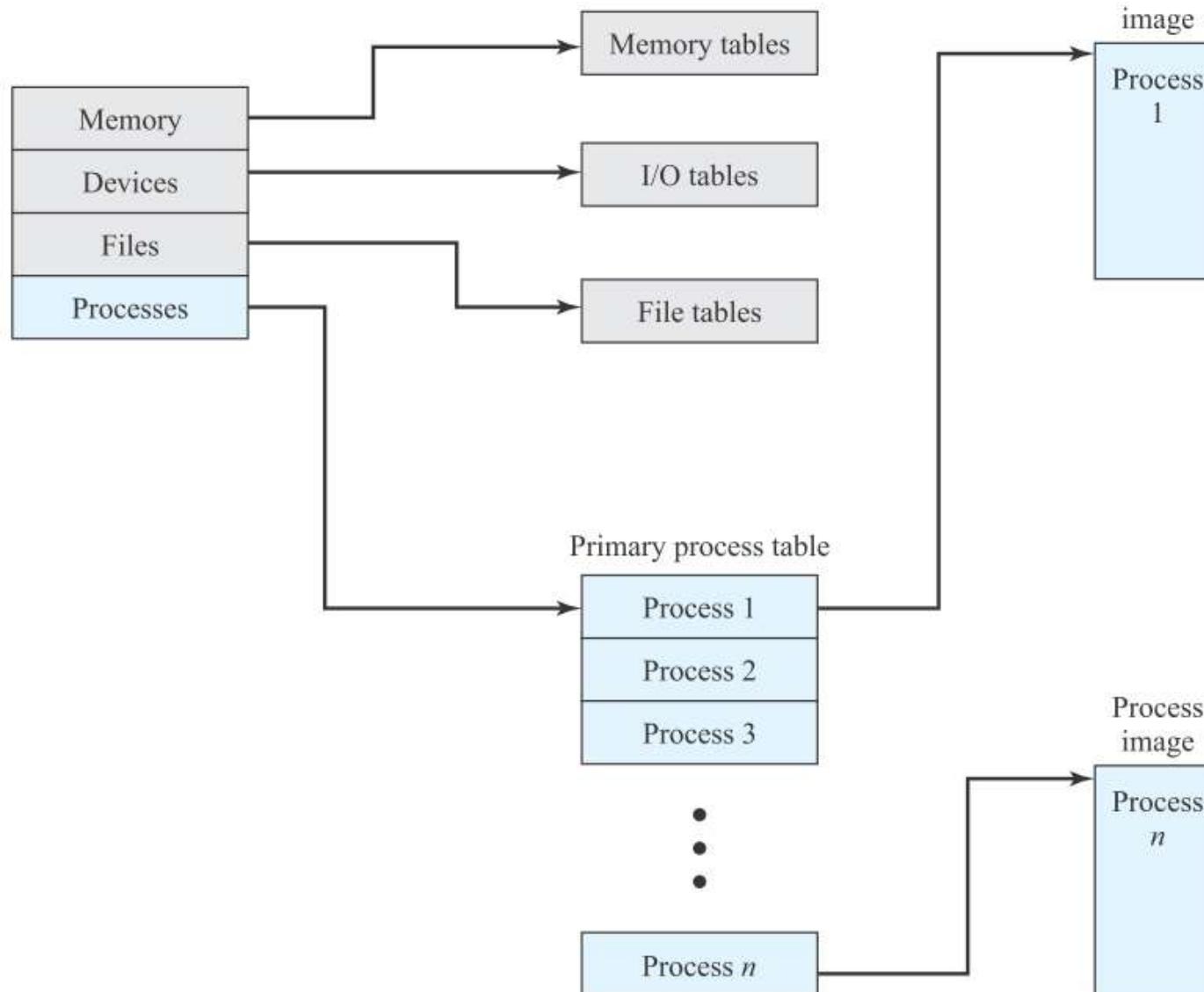
- the processes,
- the resources, and
- the allocations and access grants.

In this section we are concerned with the question: **What information does the OS need to control processes and manage resources for them?**

## OS Control Structures

The OS manages processes and resources, the straightforward approach to store information about those is to construct and maintain **tables** about each entity that it is managing.

## OS Control Structures



## OS Control Structures

The way the information is organised is an individual decision of each OS. However, the general structure is fundamentally the same.

| Table   | Description                                                                                        |
|---------|----------------------------------------------------------------------------------------------------|
| Memory  | keep track of both real and virtual memory.                                                        |
| I/O     | are used to manage I/O devices and channels of the system.                                         |
| File    | provide information about existing files, their location on secondary memory, current status, etc. |
| Process | needed to keep track of the processes in the system.                                               |

### Remarks

These tables contain cross-references. The OS requires initial information. It gets that somehow from the system.

## Process Control Structures

What must be stored for a process?

First of all, the OS must know where things are stored.

The process image:

### User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

### User Program

The program to be executed.

### Stack

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

### Process Control Block

Data needed by the OS to control the process

## Process Control Structures

What is contained in the process control block?

Generally speaking, we have three kinds of information:

| data                        | Description                                                                         |
|-----------------------------|-------------------------------------------------------------------------------------|
| Process identification      | Identifiers for the process itself, its parent, and its user, etc.                  |
| Process state information   | Registers (user-visible reg., control and status reg. (PC, . . . ), stack pointers) |
| Process control information | This is a generic phrase for various information needed to control a process.       |

### Process Control Information

#### Interprocess Communication

Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

#### Process Privileges

Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

#### Memory Management

This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

#### Resource Ownership and Utilization

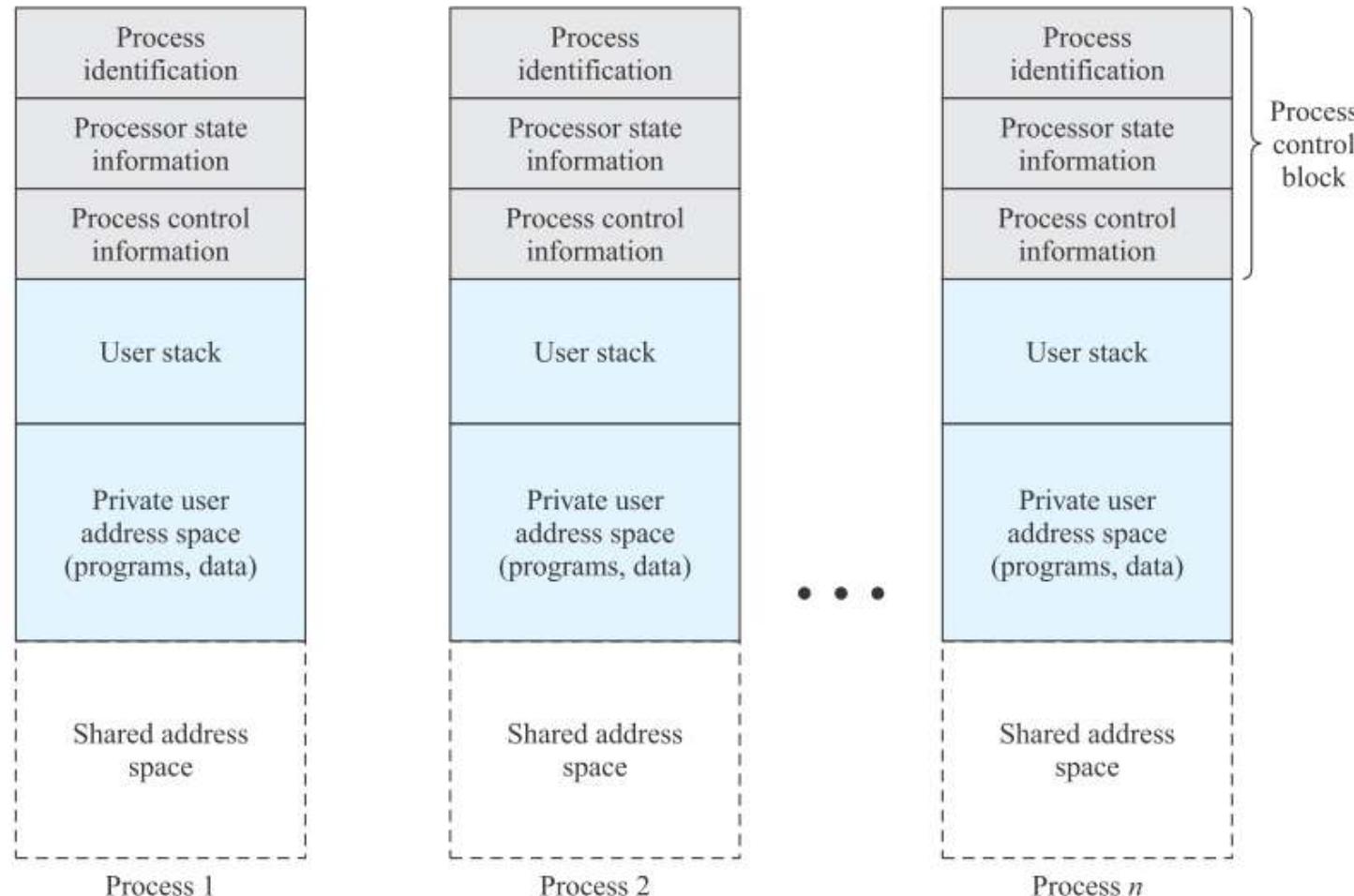
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

## Process Control Structures (I)

How can the storage of processes be organised?

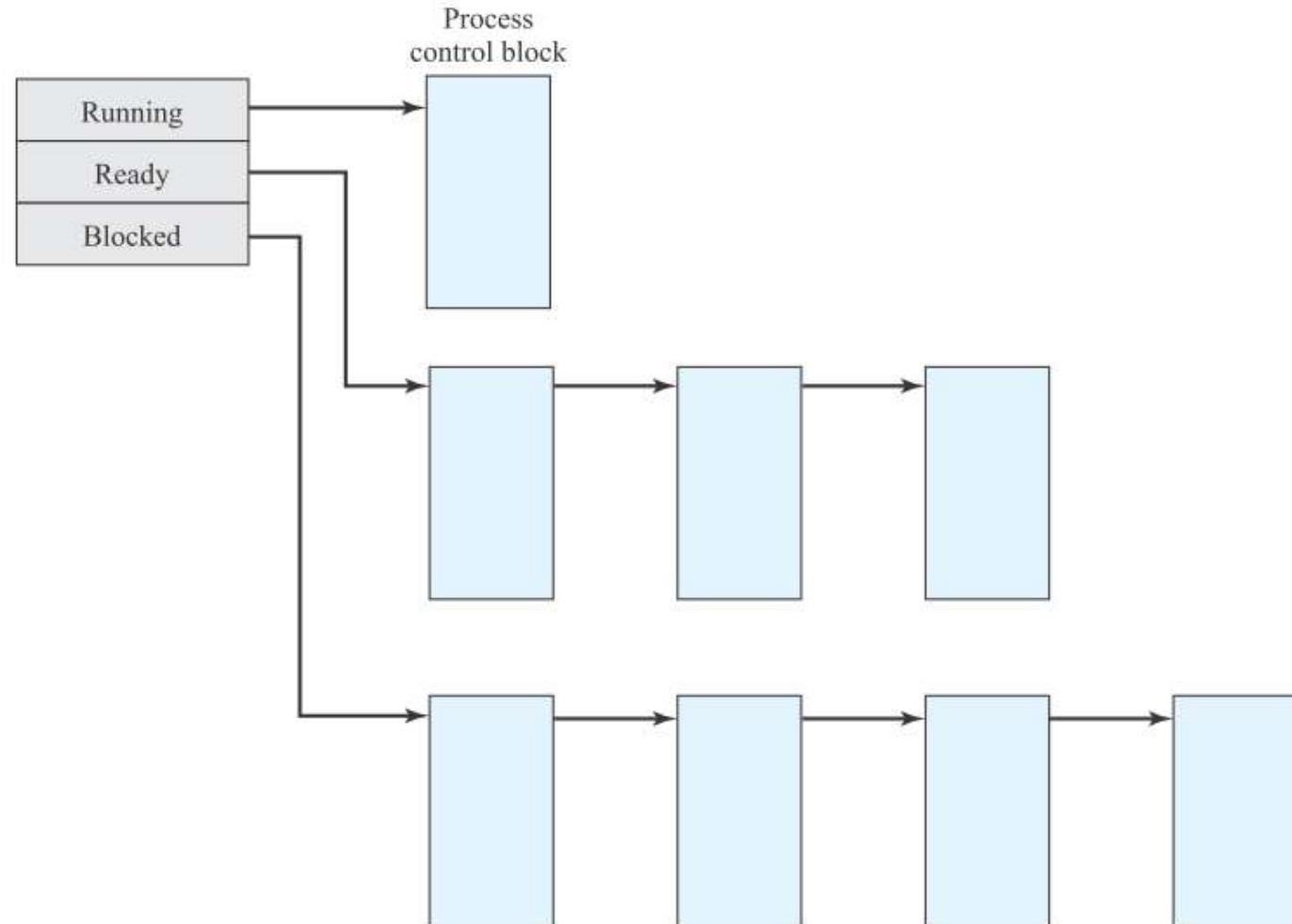
## Process Control Structures (II)

In virtual memory, the following structure is reasonable:



## Process Control Structures (III)

The list of processes can be refined (in the light of the previous discussion of process states)



## Process Description and Control

What is a Process?

Process States

Process Description

**Process Control**

Execution of the Operating System

## Process Control: Modes of Execution

User programs tend to make nonsense!

→ Hence, it is recommendable to protect a system from this danger.

A standard way to do this is to give a processor two modi of execution:

- **User mode**: Less privileged mode, for user applications and processes.
- **Kernel mode**: Non-restricted mode, for the OS (or at least those parts of the OS doing things that are important for system integrity). Aka **system mode** and **control mode**.

**Switching** between modes: Degradation to user mode is done by the OS, when it wants to hand over to ordinary process. Jumping into kernel mode is triggered by interrupts.

## Kernel Functions

### Process Management

- Process creation and termination
- Process scheduling and dispatching
- Process switching
- Process synchronization and support for interprocess communication
- Management of process control blocks

### Memory Management

- Allocation of address space to processes
- Swapping
- Page and segment management

### I/O Management

- Buffer management
- Allocation of I/O channels and devices to processes

### Support Functions: Interrupt handling

## Process Creation

How to create a new process:

1. **Assign a new PID** to the new process. Add this to the primary process table.
2. **Allocate space** for the whole process image. The size of that space is determined by default values, by the user, or by the parent process.
3. **Initialise process control block.** The initial value of the attributes are straightforward and are set in this step.
4. **Set appropriate linkage.** The OS maintains various tables. They have to be updated.
5. **Create/expand other data structures.** Some data structures used by the OS are process individual. Those have to be created.

## Process Switching

**Switching** the processes has several issues:

- **When** to switch?
- **What is to do** regarding the various data structures of the OS in case of a process switch?

## When to Switch a Process

Reasons for a process switch:

| Mechanism       | Cause                                                    | Use                                            |
|-----------------|----------------------------------------------------------|------------------------------------------------|
| Interrupt       | External to the execution of the current instruction     | Reaction to an asynchronous external event     |
| Trap            | Associated with the execution of the current instruction | Handling of an error or an exception condition |
| Supervisor call | Explicit request                                         | Call to an operating system function           |

**Traps** are interrupts which occur due to some error or exception condition. In the table above “interrupt” stands for an ordinary interrupt, for instance due to some clock event.

## When to Switch a Process: Interrupts

If an ordinary interrupt occurs, then OS branches to a dedicated interrupt handling routine.

| Interrupt       | description                                                                                                                                                                                                                                                                   |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Clock interrupt | In this case the OS checks whether the running process has exceeded its time slice. If so, the OS moves that to the “Ready” list and dispatches another process.                                                                                                              |
| I/O interrupt   | The OS figures out which processes are waiting for this event and updates the “Blocked” and “Ready” lists appropriately. Then it decides how to proceed (go on with current process or suspend it due to this new situation).                                                 |
| Memory fault    | A requested virtual memory address is not in the main memory.<br>Hence, the OS must move blocks (a page or segment) of memory from secondary memory to main memory, i.e. it issues an I/O request. The current process is blocked now and the OS switches to another process. |

## When to Switch a Process: Traps and Supervisor Calls

In case of a **trap** the OS has to determine whether this trap is fatal. If so, the process is moved to “Exit”.

Otherwise, it does some recovery operations (highly depends on the OS and the given trap).

A **supervisor call** is done if a process calls the OS for assistance.

For instance, the process could request an I/O operation explicitly. Then it calls a dedicated routine of the OS to apply the desired operation. The OS routine will then execute appropriate actions (like starting the I/O operation) and move the caller to the “Blocked” list.

## Change of the Process State

1. Save the context of the processor, including PC and other registers.
2. Update the process control block of the process that is currently in the “Running” state. Other fields have to be updated, for instance accounting information.
3. Move the process control block of this process to the appropriate queue.
4. Select another process for execution. **This is a huge topic!**
5. Update the process control block of the process selected. This includes changing the state of the process to “Running”.
6. Update memory management data structures. **This will be discuss in the part on memory management.**
7. Restore the context of the processor to that which existed at the time the
8. selected process was last switched out of the “Running” state, by loading the previous values of the PC and other registers.

## Process Description and Control

What is a Process?

Process States

Process Description

Process Control

Execution of the Operating System

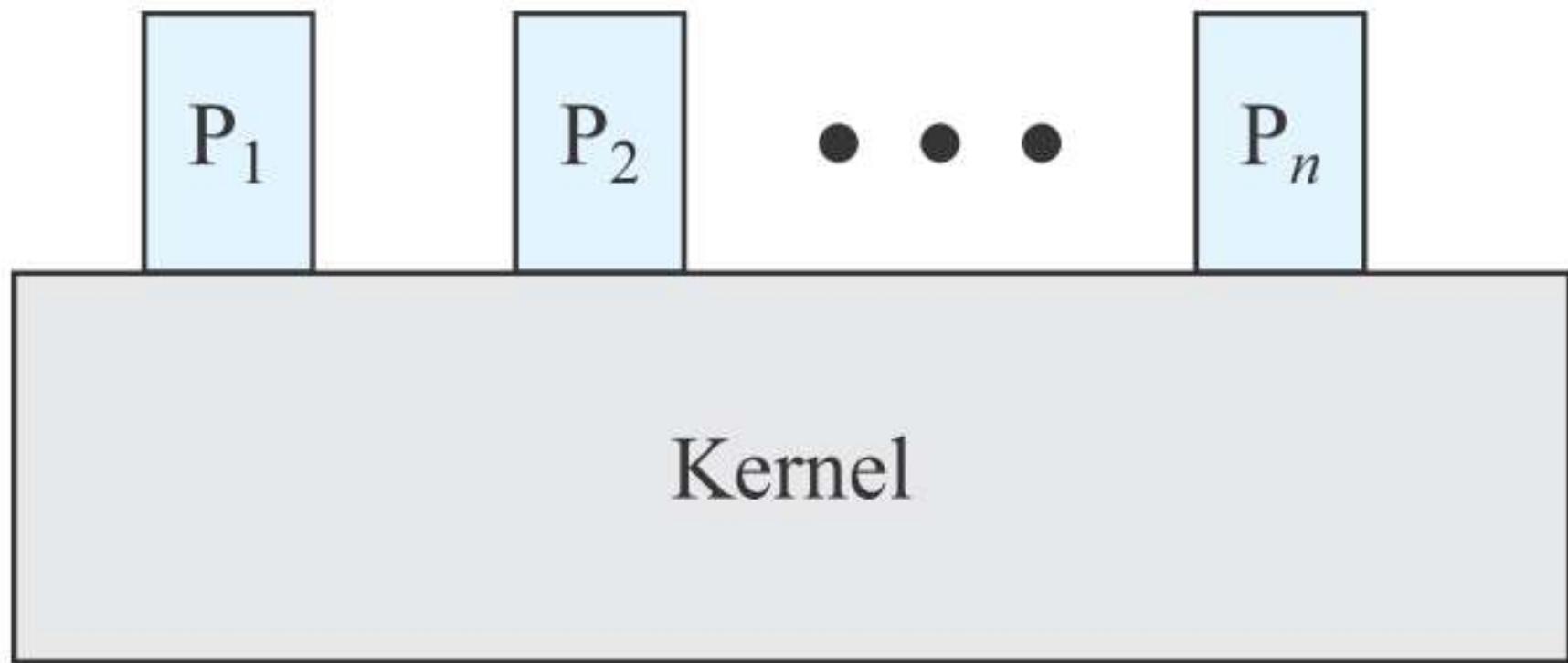
### Execution of the OS

- An OS is “just” a (set of) program(s), ie. piece(s) of software executed by the processor.
- The OS relinquishes control frequently and regains control after a while.

There several options to organize the relationship between ordinary processes and the OS.

### Option 1: Nonprocess Kernel

In this case the OS is not consider to be a process. It has its own stack and memory. Execution of OS functions is done in kernel mode.

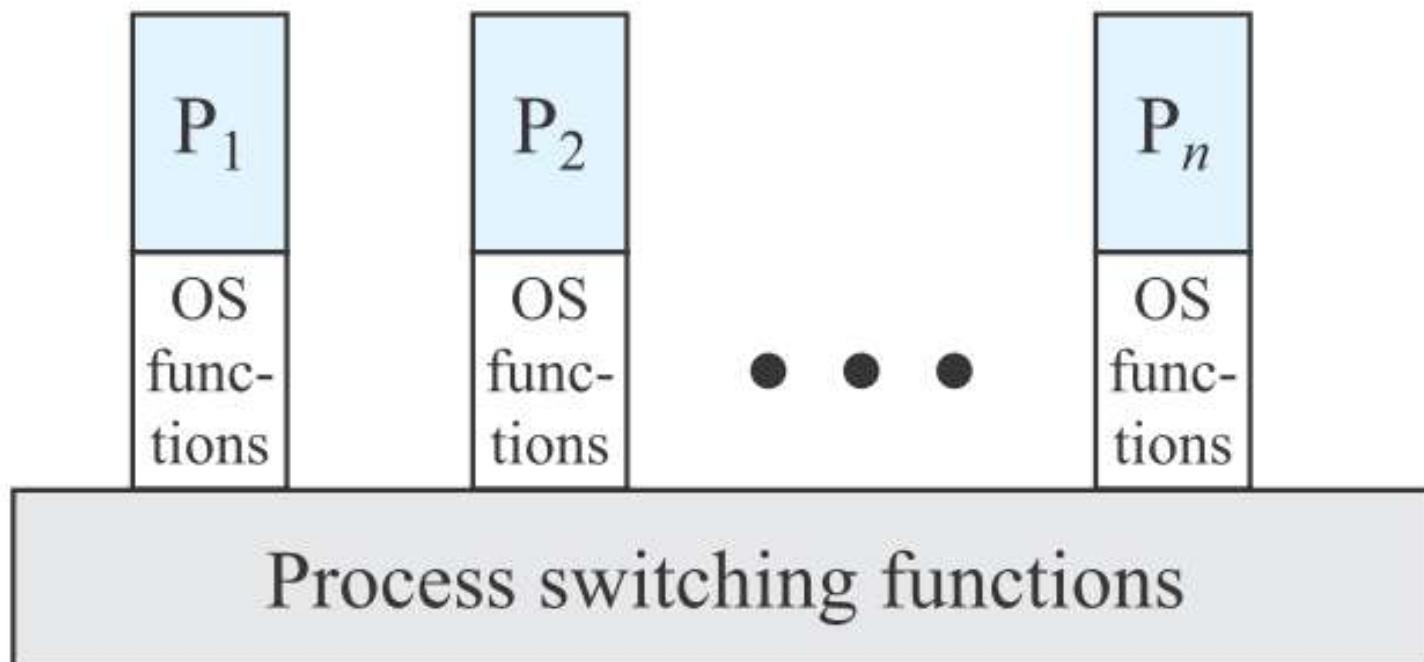


### Option 2: Execution within User Processes (I)

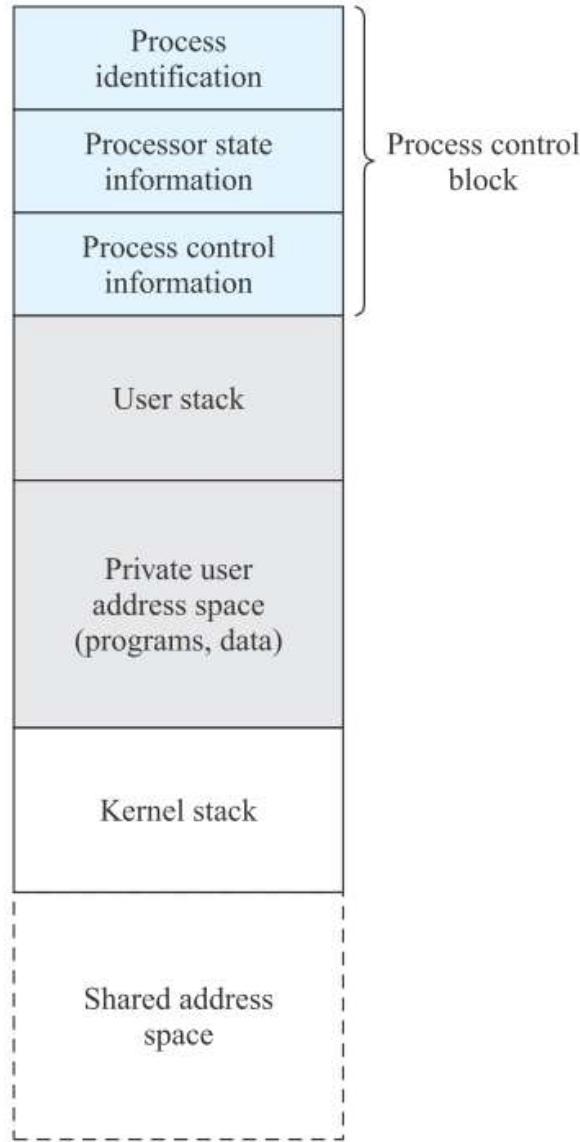
Here, an OS is seen as a collection of routines. These routines are executed within the environment of the **user's process**.

To do so, we require the following:

- The process image must be extended by a kernel stack.
- The user process can switch into the kernel when OS routines are executed.



## Option 2: Execution within User Processes (II)



### PLUS

- ▶ Saves many process switches  $\Rightarrow$  less overhead

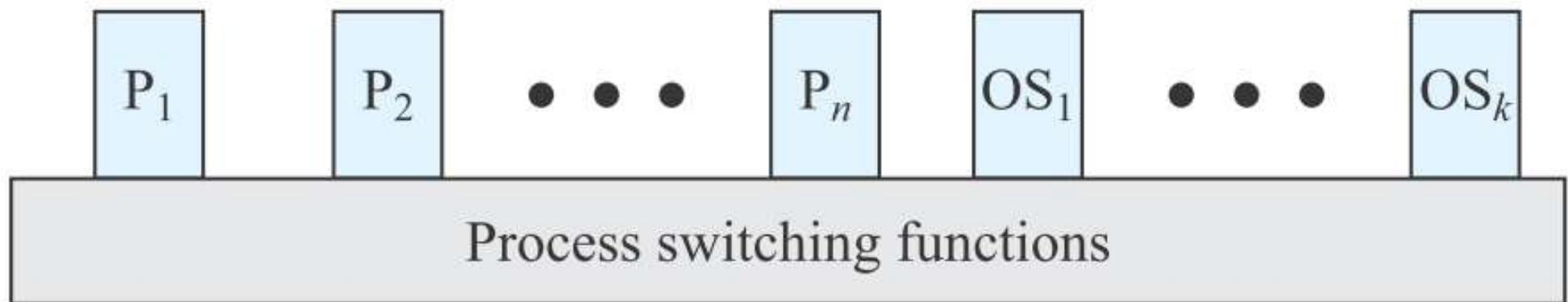
### MINUS

- ▶ Increased memory consumption

### Remark

Note the difference here between **process** and **program**.

## Option 3: Process-Based OS



### PLUS

- ▶ Clear, modular structure of the OS.
- ▶ Clean interfaces required.
- ▶ OS functions may have various priorities.

# Betriebssysteme / Operating Systems

## Multitasking

SS 2022

Prof. Dr.-Ing. Holger Gräßner

[07 OS-BS 2022 Multitasking.pptx]



### A simple multitask programm

```
while (1) {  
    /* read keyboard */  
    /* calculate content of screen */  
    /* update screen content */  
}
```

## User's influence

```
/* read keyboard */

if (kbhit() )          /* any key hit? */
    key = getch() ;     /* yes: read character */

else
    key = '\0' ;        /* no: report '\0' */
```

## A simple multitask programm

```
while (1) {  
    /* read keyboard */  
    /* read mouse position */  
    /* calculate content of screen */  
    /* update screen content */  
    /* output sound */  
    ...  
    ...  
}
```

## States of processes and threads

### READY

Occupies the CPU (only one process in a single core system) or is ready to run (multiple processes)

### BLOCKED

Can't run due to a blocked resource

### WAIT

Waiting for the end of one ore multiple subprocesses

### DEAD

Has been terminated, but did not free the used memory (processes in this state are called 'zombies')

### HELD

Has been stopped by another process

## Function pthread create()

```
pthread_create(tid,attr,nthread,arglist);
```

**tid:** Pointer to thread ID

**attr:** Pointer to attributes

**nthread:** Pointer to thread function

**arglist:** Pointer to argument list

## Thread generation: pthread create() (I)

```
#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <errno.h>

void *thread2(void *);

int main(void)
{
    pthread_t tid;
    if (pthread_create(&tid, NULL, thread2, NULL) !=0) {
        fprintf(stderr, „Error in pthread_create() : %d\n",
                strerror(errno));
        return EXIT_FAILURE;
    }
}
```

→ [example\\_pthread\\_create.c](#)

## Thread generation: pthread create() (II)

```
printf („Output Thread 1\n”);  
sleep(1) ;  
return EXIT_SUCCESS;  
}  
  
void *thread2(void *data)  
{  
    printf („Output Thread 2\n”);  
}
```

→ [example\\_pthread\\_create.c](#)

## Function fork()

```
rval = fork();
```

```
rval = PID(child):    parent process
```

```
rval = 0:              child process
```

```
rval < 0:              error
```

## Process generation: fork() (I)

```
#include <unistd.h> #include <sys/types.h> #include <stdio.h> #include <errno.h>
/* Compiler call: gcc -o example_fork example_fork.c */

int main(void)
{
    pid_t npid;
    int i;

    npid = fork();
    if (npid > 0) {
        printf("Process %d: I am your father.\n",
               getpid());
    }
}
```

## Process generation: fork() (II)

```
else if (npid ==0) {  
    printf("Prozess %d: Output child process.\n",  
           getpid());  
  
}  
  
else {  
    fprintf(stderr, "Error No. %d in fork() !\n", errno);  
    return 1;  
}  
  
}
```

## Process generation: fork() (III)

```
#include <unistd.h> #include <sys/types.h> #include <stdio.h> #include <errno.h>
/* Compiler call: gcc -o example_fork example_fork.c */
```

```
int main(void)
```

```
{
```

```
    pid_t npid;
```

```
    int i;
```

Now in a loop!

```
for (i = 0; i<2; i++) {
```

```
    npid = fork();
```

```
    if (npid >0) {
```

```
        printf("Process %d: I am your father.\n",
               getpid());
```

```
}
```

→ [example\\_fork.c](#)

## Process generation: fork() (IV)

```
else if (npid ==0) {  
    printf("Prozess %d: Output child process.\n",  
           getpid());  
  
}  
  
else {  
    fprintf(stderr, "Error No. %d in fork() !\n", errno);  
    return 1;  
}  
  
}  
} → output??
```

→ [example\\_fork.c](#)

## Process generation: fork() (V)

## Skywalker family tree

| i     |  | PID |  | who's that?    |  | output                      |
|-------|--|-----|--|----------------|--|-----------------------------|
| ----- |  |     |  |                |  |                             |
| 0     |  | 101 |  | Darth Vader    |  | 101: I am your father.      |
| 0     |  | 102 |  | L Luke         |  | 102: Newborn child process! |
| 1     |  | 101 |  | Darth Vader    |  | 101: I am your father.      |
| 1     |  | 103 |  | L Leia         |  | 103: Newborn child process! |
| 1     |  | 102 |  | Luke           |  | 102: I am your father.      |
| 1     |  | 104 |  | L Luke's child |  | 104: Newborn child process! |

→ [example\\_fork.c](#)

### Process generation: fork() (VI)

**Homework 1 – create the following output:**

i=0, PID=101: I am your father.

i=0, PID=102: Newborn! Line of ancestors: 101→102.

i=1, PID=101: I am your father.

i=1, PID=103: Newborn! Line of ancestors: 101→103.

i=1, PID=102: I am your father.

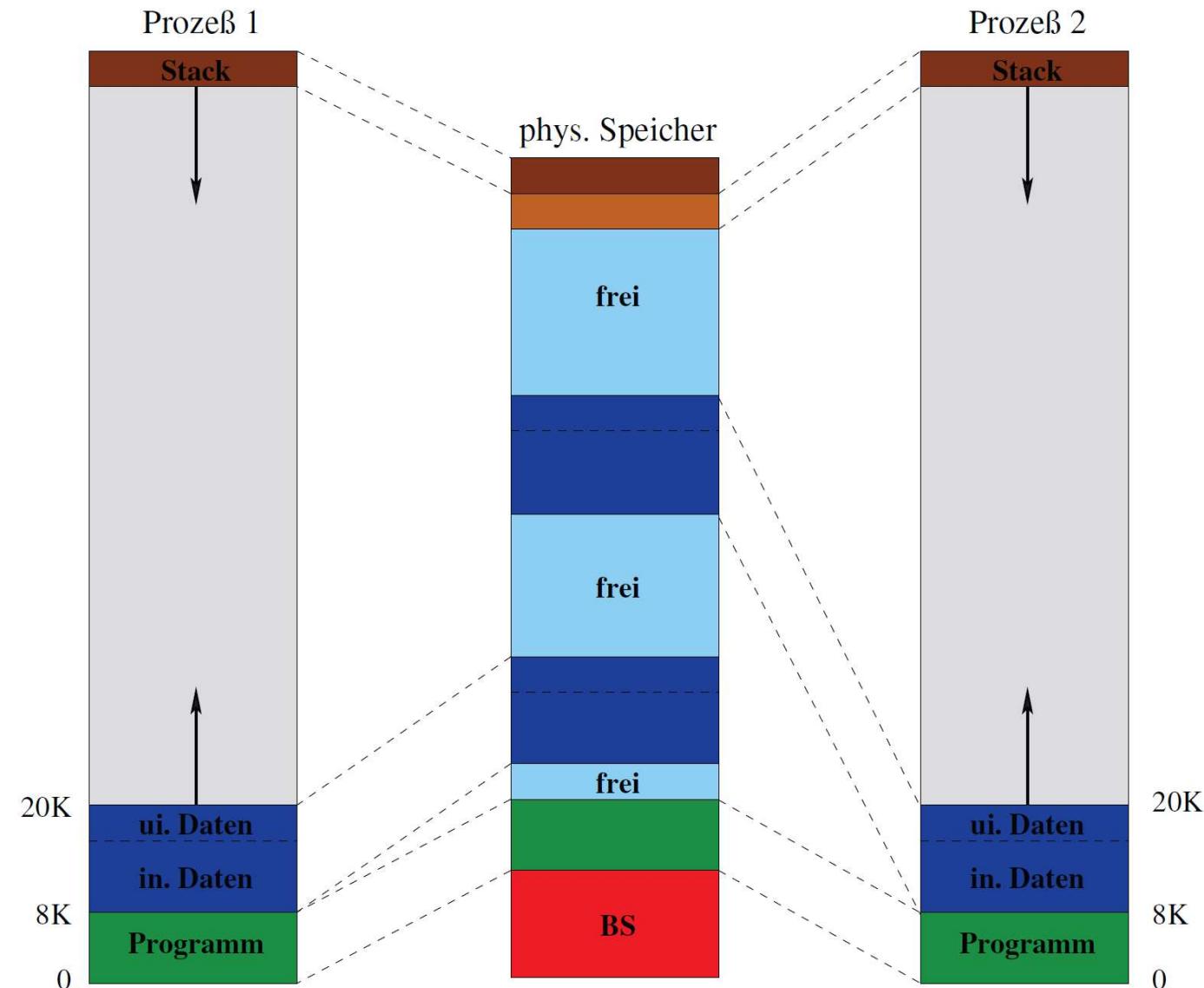
i=1, PID=104: Newborn! Line of ancestors: 101→102→104.

→ `example_forkloop2.c`

**Homework 2:**

Write a simple program to run 16 identical processes!

## Process layout in memory



## exec()

```
int rval = execl(char *file, char *arg, ...);  
int rval = execlp(char *path, char *arg, ...);  
int rval = execle(char *file, char *arg, ...,  
                  NULL, char *envp[]);  
  
int rval = execv(char *file, char *argv[]);  
int rval = execvp(char *path, char *argv[]);  
int rval = execve(char *file, char *argv[],  
                  char *envp[]);
```

**rval**: -1 in case of error. No return otherwise.

**file**: file to be called

**path**: file to be called in search path

## Process generation: fork() and exec() (I)

```
/* Program example_fork_exec1 (parent process) */

#define PROGNAM "example_fork_exec2"

#include <unistd.h> #include <sys/wait.h> #include <stdio.h> #include <stdlib.h>
#include <string.h> #include <errno.h>

int main(int argc, char *argv[]) {
    pid_t npid;
    int rval;
    char *argvk[] = {PROGNAM, NULL};
    char *envmt[] = {"PROG=exec", "TEST=text", NULL};

    npid = fork() ;
```

→ [example\\_fork\\_exec1.c](#)

## Process generation: fork() and exec() (II)

```
if (npid > 0) {  
    printf ("Output parent process \"%s\" (PID: %d) !\n",  
           argv[0], getpid());  
    wait(&rval);  
    return EXIT_SUCCESS;  
} else if (npid == 0) {  
    execve(PROGNAM, argvk, envmt);  
    return EXIT_FAILURE;  
} else {  
    fprintf (stderr , "Error in fork() : %s\n",  
            strerror(errno)) ;  
    return EXIT_FAILURE;  
}  
}
```

→ [example\\_fork\\_exec1.c](#)

## Process generation: fork() and exec() (III)

```
/* Programm example_fork_exec2 (child process) */
#include <unistd.h> #include <stdio.h> #include <stdlib.h>

int main(int argc, char *argv[], char *envp[])
{
    int i=0;
    printf ("Output child process \\"%s\\" (PID: %d) !\n",
            argv[0], getpid());
    printf ("Environment:\n");
    while (envp[i] != NULL) {
        printf ("%d. %s\n", i+1, envp[i]);
        i++;
    }
    return EXIT_SUCCESS;
}
```

→ [example\\_fork\\_exec2.c](#)

### wait() functions

```
pid_t wait(int *status)  
pid_t waitpid(pid_t pid, int *status, int options)
```

**status:** -1 return value child process

**options:** 0 or WNOHANG (don't wait)

## POSIX spawn() functions

```
int rval = posix_spawn(pid_t *pid, char *file, NULL,  
                      NULL, char *argv[],  
                      char *envp[]);
```

```
int rval = posix_spawnp(pid_t *pid, char *path, NULL,  
                       NULL, char *argv[],  
                       char *envp[]);
```

**rval:** 0 if successfull

**file:** name of file to be called

**path:** name of file in search path

## Process generation: POSIX spawn() (I)

```
/* Programm spawn1 (parent process) */
#define PROGNAM "example_spawn2"
#include <spawn.h> #include <sys/wait.h> #include <stdio.h> #include <stdlib.h>

int main(int argc, char *argv[])
{
    pid_t pid;
    int rval ;
    char *argvk[] = {PROGNAM, NULL};

    if (posix_spawn(&pid, PROGNAM, NULL, NULL, argvk, NULL)
        != 0) {
        printf ("error in 'posix_spawn()' '\n");
        return EXIT_FAILURE;
    }
    → example_spawn1.c
```

## Process generation: POSIX spawn() (II)

```
printf ("Output parent process \">%s\": Child's PID is  
        %d\n", argv[0], pid);  
  
wait(&rval);  
  
return EXIT_SUCCESS;  
}
```

→ [example\\_spawn1.c](#)

## Process generation: POSIX spawn() (III)

```
/* Programm spawn2 (child process) */

#include <unistd.h> #include <stdio.h> #include <stdlib.h>

int main(int argc, char *argv[])
{
    printf ("Output child process \">%s\": My PID is %d\n",
            argv[0], getpid());
    return EXIT_SUCCESS;
}
```

→ [example\\_spawn2.c](#)

## Priorities

List all processes with current ID, priority and path:

```
ps ax -o pid,ni,cmd
```

Start a new program **prog** with priority of **niceness**:

```
nice -n niceness prog
```

Set the priority of a running process **PID** to the new value of **niceness**:

```
renice niceness -p PID
```

Note:

- Users can change their own processes in the range of 0...19 for **niceness**.
- root can change all processes in the range of -20...19 for **niceness**.

# **Betriebssysteme / Operating Systems**

## **Exercise „Bidding during a Skat Game“**

SS 2020

Prof. Dr.-Ing. Holger Gräßner

[09 OS-BS 2020 Skat-Demo.pptx]

## Introduction

We will program the bidding of a skat game in C:

- The program will receive the maximum bids of the 3 players via command line.
- The program code for the 3 players will be executed concurrently.
- We will use different operating system calls to do the synchronisation of concurrent processes/threads and the communication between them.
- At the end, the winner of the bidding will be determined.

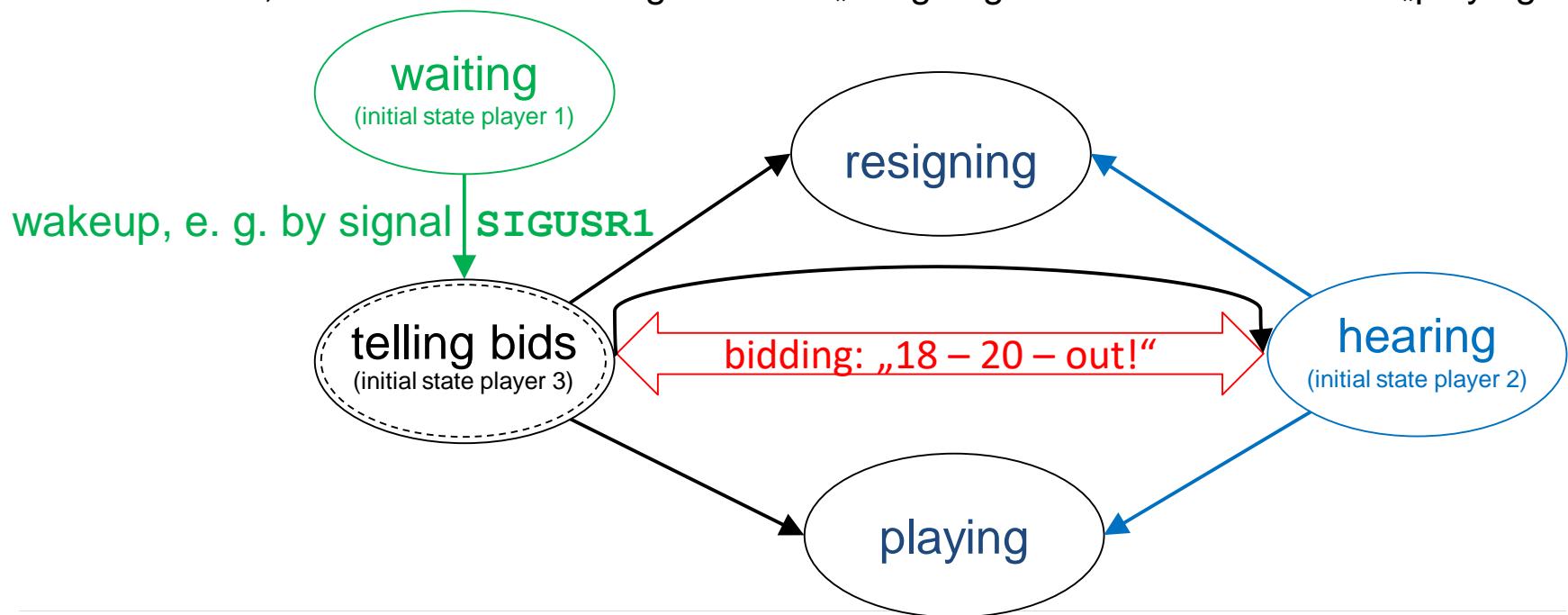
## Learning outcome

- Capability to develop concurrent programs.
- Knowledge of typical methods for interprocess communication and process synchronisation.

## State machine for the bidding

Player 1: Starts waiting, player 2 starts hearing, player 3 starts telling bids.

1. Players 2 and 3 are bidding.
2. As a result, the looser will change to state „resigning“ and the winner (stays in or switches to) state „hearing“.
3. Player 1 awakes and bids against the winner.
4. As a result, one of them will change to state „resigning“ and the other to state „playing“.



## What we are going to develop

We will do the skat task multiple times.

Every time, another OS method will be used:

| No. | Player  | How to place bids                | How to wakeup player 1 | Program module to link |
|-----|---------|----------------------------------|------------------------|------------------------|
| 1   | thread  | global variable                  | global variable        | <b>skat-threads.c</b>  |
| 2   | process | global variable in shared memory | signal <b>SIGUSR1</b>  | <b>skat-shm.c</b>      |
| 3   | process | pipe                             | signal <b>SIGUSR1</b>  | <b>skat-pipes.c</b>    |
| 4   | process | message queue                    | signal <b>SIGUSR1</b>  | <b>skat-queues.c</b>   |

## Execution of the program

Our C-Program SKAT has to implement the bidding of a skat game.

- We will pass the maximum bids of process 1, process 2 and process 3 by using the command line.

Example: SKAT (18, 46, 23)

Player 1 (dealer) is able to bid up to 18, player 2 (hearer) up to 46 and player 3 (teller) up to 23.

- The programm will start a concurrent thread or process for each of the 3 players.
- Players will communicate using of the methods listed in the table.
- In the first step, player 3 (hearer) and player 2 (teller) will bid, while player 1 (dealer) sleeps. The latter will be wakened by one of the methods listed in the table.

## Output

The first process has to init all necessary variables.

The player, who is in the state `telling`, has to

- place the next bid in the communication channel,
- print the next bid to the console.

The player, who is in the state `hearing`, must answer by placing a number in the communication channel and to the console:

- 1 means „OK“,
- 0 means „I resign“.

It might be necessary to implement an appropriate handshake mechanism for the communication.

Each player ends with one of the following final statements to the console:

- „Player 1234 (teller): Resigning at 23“
- „Player 1235 (hearer): I‘m going to play!“

## Setup of the C-Projekt (I)

Existing modules and modules to be created:

- **glob-defs.h, skat-comm.h, skat-defs.h**: Header files.
- **skat-main.c**:
  - Skat specific functions and data structures (implementation of bidding etc.).  
Deeper knowledge of this module is irrelevant for the lecture.
- **skat-threads.c**:
  - Implementation of players as threads. Communication between threads by global variables.  
No skat knowledge required.
- **skat-shm.c**:
  - Implementation of players as processes. Communication between processes by global variables, located in a shared memory.  
No skat knowledge required.

## Setup of the C-Projekt (II)

Existing modules and modules to be created:

- **skat-pipes.c**:
  - Implementation of players as processes. Communication between processes by pipes and signals.  
No skat knowledge required.
- **skat-queues.c**:
  - Implementation of players as processes. Communication between processes by message queues.  
No skat knowledge required.

## Predefined types and variables

```
// Every player will be in one of the following states:  
enum stateOfGame {waiting, hearing, tellingBids, resigning,  
                    playing};  
  
#define NUMSPW 14;  
  
int sequenceOfBids[NUMSPW] = {18, 20, 22, 23, 27, 33, 35, 36,  
                                40, 44, 45, 46, 48, 50};  
  
struct tPlayer {  
    const char * pcTask; // Task of the player (Dealer,  
                        Listener, Teller)  
    enum stateOfGame nState; // 1. process (Dealer) will start  
                            waiting, 2.: hearing, 3.: tellingBids  
    int nMaxBid; // upper bound for this processe's bids  
};  
  
struct tPlayer skatPlayers[3];
```

## makefile (I)

- The whole build process of the executable **skat** programm will be defined by a **makefile**.
- Switching between the solutions (threads, shared memory, pipes oder MessageQueues) can be done by placing some command characters <#>.

# Important notes:

```
# All "recipe lines" MUST start with a <tab> character!!!
# This file has to be named "makefile".
# To build the skat progam, just type "make skat".
# "make clean" will remove object files + executable files.
```

# solution based on threads and global variables:

```
skat : skat-main.o skat-threads.o
        gcc -o skat skat-main.o skat-threads.o -lrt
```

## makefile (II)

```
# solution based on processes and global variables in a
# shared memory:
#skat : skat-main.o skat-shm.o
# gcc -o skat skat-main.o skat-shm.o -lrt

# solution based on processes and pipes:
#skat : skat-main.o skat-pipes.o
# gcc -o skat skat-main.o skat-pipes.o

# solution based on processes and message queues:
#skat : skat-main.o skat-queues.o
# gcc -o skat skat-main.o skat-queues.o -lrt
```

## makefile (III)

```
skat-threads.o : skat-threads.c skat-defs.h glob-defs.h  
gcc -c skat-threads.c
```

```
skat-shm.o : skat-shm.c skat-defs.h glob-defs.h  
gcc -c skat-shm.c
```

```
skat-main.o : skat-main.c skat-comm.h glob-defs.h  
gcc -c skat-main.c
```

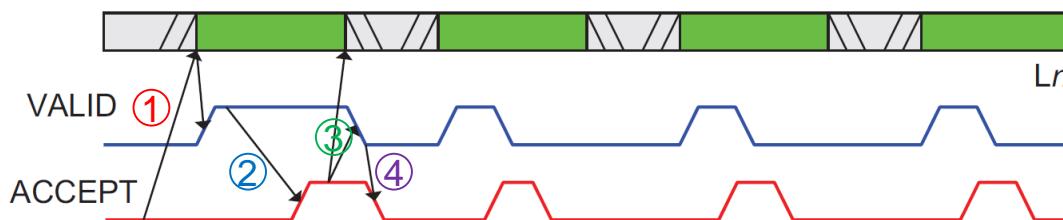
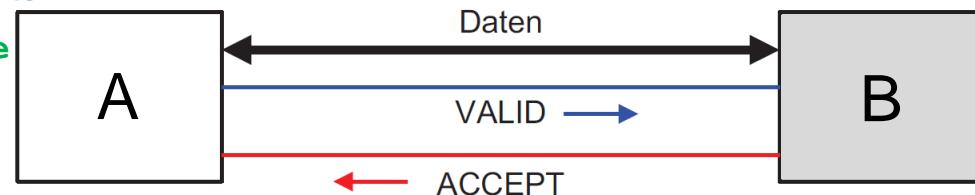
```
skat-pipes.o : skat-pipes.c skat-defs.h glob-defs.h  
gcc -c skat-pipes.c
```

## makefile (IV)

```
skat-queues.o : skat-queues.c skat-defs.h glob-defs.h  
    gcc -c skat-queues.c  
  
clean :  
    rm skat skat-main.o skat-pipes.o skat-queues.o skat-  
    threads.o skat-shm.o
```

Handshake for data transfer `g_Bid` from thread A to B:

```
int g_Bid = 0;      // data to exchange
int g_accept = 0;   // handshake flag
int g_valid = 0;    // handshake flag
```



```
// Thread A:                                     // Thread B:
while (g_accept != 0) usleep(100); ① int newData;
g_Bid = 18; // transfer 18 to B           // wait for NEW data:
g_valid = 1;                                ② while (g_valid == 0) usleep(100);
  newData = g_Bid; // I got new data!
  g_Bid = 123;    // my answer
// wait for B's confirmation:                ③ g_accept = 1;    // indicate: I got it!
while (g_accept == 0) usleep(100); ④          // wait for A's confirmation:
printf("answer is %d\n", g_Bid);           ⑤ while (g_valid != 0) usleep(100);
g_valid = 0; // ready for more            ① g_accept = 0; // indicate: Ready for more
```

# **Betriebssysteme / Operating Systems**

## **Interprocess Communication**

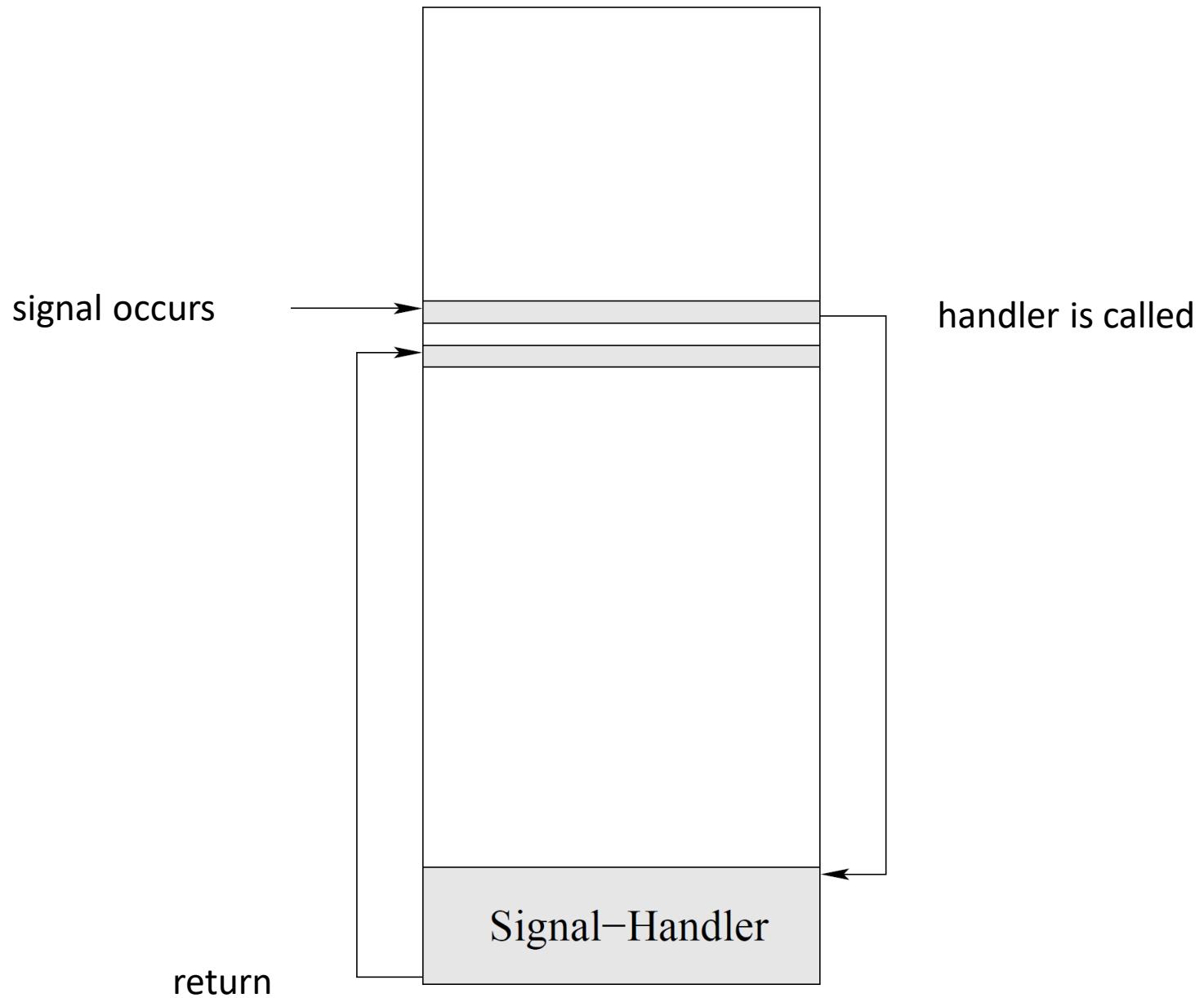
SS 2020

Prof. Dr.-Ing. Holger Gräßner

[10 OS-BS 2020 Interprocess communication.pptx]



## Signals



## Catching a signal with a signal handler (I)

```
#include <stdio.h> #include <signal.h>

#define FALSE 0
#define TRUE (!FALSE)

void sigh(int); /* handler declaration */

volatile sig atomic t flag = FALSE;

int main()
{
    signal(SIGINT,sigh); /* assign handler fuction */
    printf ("Press ^C to call the signal handler\n");
}
```

→ signal\_handler.c

## Catching a signal with a signal handler (II)

```
while (!flag)
;
printf (" Program will be terminated! \n") ;
return 0;

}

void sigh(int signum)
{
    flag = TRUE;
}
```

→ [signal\\_handler.c](#)

## Process synchronisation with a signal (I)

```
#include <stdio.h> #include <signal.h> #include <unistd.h>
#include <sys/wait.h> #include <sys/types.h>

#define FALSE 0
#define TRUE (!FALSE)

void sigh(int); /* handler declaration */

int main(void)
{
    pid_t npid;
    int status;

    npid = fork();
```

→ proc\_sync\_signal.c

## Process synchronisation with a signal (II)

```
if (npid) {  
    printf ("Parent process: Press CR to send SIGUSR1 to  
            child process!\n");  
    getchar();  
    kill (npid, SIGUSR1); /* send SIGUSR1 to child npid */  
    printf ("Parent process: SIGUSR1 has been send.\n");  
    wait(&status);  
    printf ("Parent process: Child process terminated,  
            exit state = %i\n", WEXITSTATUS(status));  
    return 0;  
}
```

→ proc\_sync\_signal.c

## Process synchronisation with a signal (III)

```
else {
    printf ("Child process: Waiting for signal...\n");
    signal(SIGUSR1, sigh); /* assign SIGUSR1 to handler */
    pause();                /* block until signal */
    printf ("Child process: SIGUSR1 received! End in 1s\n");
    sleep(1);
    return 55;
}

void sigh(int signum) /* Signal handler for SIGUSR1 */
{ /* attention: only reentrant resistant functions here... */
}
```

→ proc\_sync\_signal.c

## Basic signal handling

```
sighandler_t signal (int signum, sighandler_t action);
```

Parameter:

**signum**: Signal to specify it's behaviour.

**action**: New action:

- **SIG\_DFL**: Default action for this signal.
- **SIG\_IGN**: Ignore this signal (not possible for **SIGKILL** or **SIGSTOP**).
- Adress of a signal handler function.

## Advanced signal handling (I)

### Function `sigaction`:

```
int sigaction (int signum, const struct sigaction *restrict action,  
               struct sigaction *restrict old-action);
```

Parameter:

- **signum:** Signal to specify it's behaviour.
- **action:** New action. **NULL**: No change of behaviour.
- **old-action:** Get information about the current behaviour.  
**NULL**: No information required.

## Advanced signal handling(II)

Structure `sigaction` with some elements:

`sighandler_t sa_handler`

New action:

- `SIG_DFL`: Default action for this signal.
- `SIG_IGN`: Ignore this signal (not possible for `SIGKILL` or `SIGSTOP`).
- `Handler`: Address of a signal handler function.

`sigset_t sa_mask`:

Specifies a set of signals to block, while the handler is running.

Should be defined by usage of the functions `sigemptyset()` and `sigaddset()`, or `sigfillset()` and `sigdelset()`.

`int sa_flags`:

Flags to define the signal's behaviour:

- e. g. `SA_RESTART`: Library functions like `open()`, `read()`, `write()` will be resumed after execution of the signal handler.
- `NULL`: Library functions like `open()`, `read()`, `write()` will be terminated with errors after execution of the signal handler.

## Usage of signal sets (I)

```
#include <stdio.h> #include <signal.h>  
#define FALSE 0  
#define TRUE (!FALSE)  
  
sig_atomic_t flag = FALSE;  
  
void sigh(int);  
  
int main()  
{  
    sigset_t set;  
    struct sigaction act;
```

→ [signal\\_set.c](#)

## Usage of signal sets (II)

```
sigemptyset(&set);  
sigaddset(&set, SIGINT);  
act.sa_flags = 0;  
act.sa_mask = set;  
act.sa_handler = &sigh;  
sigaction(SIGINT, &act, NULL);  
printf ("Press ^C to call the signal handler!\n");  
while (!flag)  
{  
    ;  
    printf ("Programm terminates now!\n");  
    return 0;  
}
```

→ [signal\\_set.c](#)

## Usage of signal sets (III)

```
void sigh(int signum)
{
    flag = TRUE;
}
```

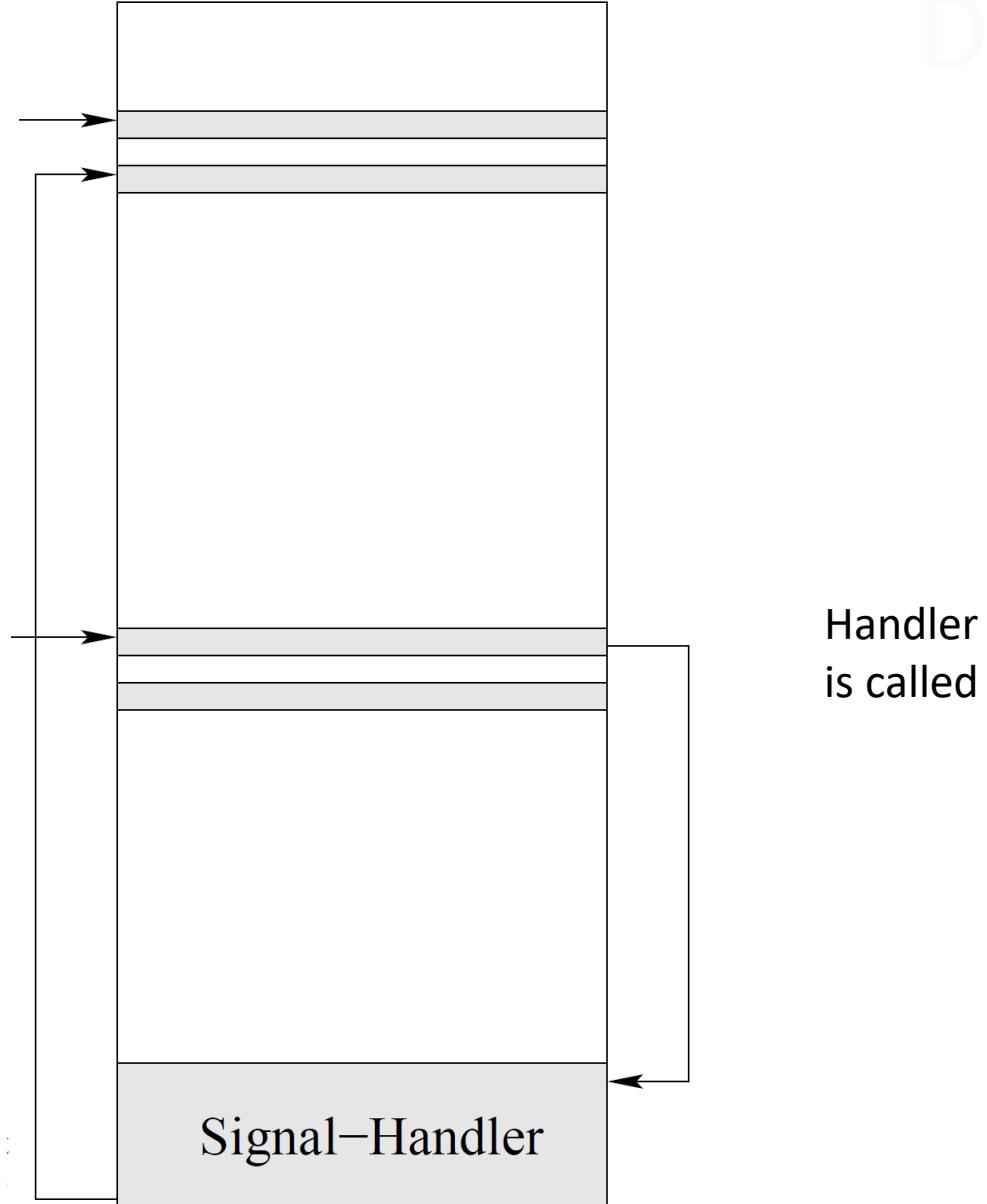
→ [signal\\_set.c](#)

`sigsetjmp()`  
`siglongjmp()`

Define point of  
reentry with  
`sigsetjmp()`

signal occurs

jump to point of  
reentry with  
`siglongjmp()`



## sigsetjmp() and siglongjmp()

```
sigsetjmp(env, smask) ;
```

**env:** address of environment buffer

**smask:** ≠ 0: include signal mask

**Return value:** = 0: first call (definition of jump label)

≠ 0: following calls (jump to label)

```
siglongjmp(env, ret) ;
```

**env:** address of environment buffer

**ret:** ≠ 0: second call of **sigsetjmp()**

## Usage of sigsetjmp() and siglongjmp() (I)

```
#include <stdio.h> #include <signal.h> #include <setjmp.h>

#define FALSE 0
#define TRUE (!FALSE)

void sigh(int);

Sigjmp_buf env;

int main()
{
    int retval;

    signal(SIGINT,sigh);
```

→ [siglongjmp.c](#)

## Usage of sigsetjmp() and siglongjmp() (II)

```
if (( ret_val = sigsetjmp(env, 0) ) == 0) { // first call
    printf ("sigsetjmp() has been initialised. Return
            value was %d.\n -> endless loop\n", retval);
    while (1)
        ;
} else // following calls
printf ("Return value of sigsetjmp() was now %d.\n
        -> EXIT!\n", retval);

return 0;
}

void sigh(int signum)
{
    siglongjmp(env, TRUE);
}
```

### Properties of signal handling:

- no order defined (multiple signals),
- no priorities defined,
- no queues,
- no data transmission,
- bad programming style (→GOTO).

→ [siglongjmp.c](#)

## Usage of pipes and FIFOs

```
int fds[2], rval;  
  
rval = pipe(fds);           // create pipe  
  
write(fds [1], ...);       // write access  
  
read(fds [0], ...);        // read access
```

```
int fds, rval ;  
  
rval = mkfifo(name,rights); // create FIFO  
  
fds = open(name,mode)       // open FIFO  
  
write(fds, ...);           // write access  
  
read(fds, ...);            // read access
```

## Message transmission with pipe() (I)

```
#include <stdio.h> #include <stdlib.h> #include <sys/types.h>
#include <sys/stat.h> #include <errno.h>

int main(void)
{
    pid_t npid;
    size_t anz;
    int fds[2];
    char msgbuf[100] = "\0";

    if (pipe(fds) < 0) {
        perror("Pipe");
        return EXIT_FAILURE;
    }
```

→ pipe.c

## Message transmission with pipe() (II)

```
npid = fork();  
if (npid) {  
    printf ("Parent process: please type a  
            message:\n");  
    fflush (stdin);  
    scanf ("%[^\\n]",msgbuf);  
    anz = strlen (msgbuf)+1;  
    write(fds[1], msgbuf, anz);  
    printf ("Parent process: EXIT\n");  
}  
}
```

→ pipe.c

## Message transmission with pipe() (III)

```
else {  
    printf ("Child process: waiting for message...\n");  
    if ((anz=read(fds[0], msgbuf, sizeof(msgbuf))) != -1) {  
        printf ("Child process: I received  
                this message: \n %s\n", msgbuf);  
        printf ("Child process: EXIT\n");  
    } else  
        printf ("Child process: No message for me!  
                (error %s) !\n", strerror(errno));  
}  
}
```

### Properties of pipes:

- no names,
- definition at compile time,
- easy and fast.

→ Skat exercise using pipes!

→ pipe.c

## Message transmission with FIFO (I)

```
#include <stdio.h>      #include <stdlib.h>  #include <sys/types.h>
#include <sys/stat.h>    #include <fcntl.h>    #include <errno.h>

// Note: FIFOs will not run on a Windows NTFS file system!

#define TFIFO "tfifo"
#define BUflen 100
#define MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)

int main(void)
{
    pid_t npid;
    size_t anz;
    int fds;
    char *fifo_nam = TFIFO;
    char msgbuf[BUflen] = "\0";
```

→ fifo.c

## Message transmission with FIFO (II)

```
if (mkfifo(fifo_nam, MODE) < 0) {  
    printf ("Error creating FIFO (%s) !\n", strerror(errno));  
    return EXIT_FAILURE;  
}  
  
npid = fork();  
  
if (npid) {  
    if ((fds=open(fifo_nam, O_WRONLY)) == -1) {  
        printf ("Parent process: Could't open FIFO for  
                writing (%s) !\n", strerror(errno));  
        return EXIT_FAILURE;  
    }  
  
    printf ("Parent process: Enter a message:\n");  
    fflush (stdin);  
    scanf ("%[^\\n]", msgbuf);  
}
```

→ fifo.c

## Message transmission with FIFO (III)

```
anz = strlen(msgbuf) + 1;  
write(fds, msgbuf, anz);  
printf ("Parent process: EXIT\n");  
} else {  
    if ((fds=open(fifo_nam, O_RDONLY)) == -1) {  
        printf ("Child process: Could't open FIFO for  
                reading (%s) !\n", strerror(errno));  
        return EXIT_FAILURE;  
    }  
    printf ("Child process: Waiting for a message...\\n");
```

→ fifo.c

## Message transmission with FIFO (IV)

```
if ((anz=read(fds, msgbuf, sizeof(msgbuf))) != -1) {  
    printf ("Child process: I received this  
            message:\n %s\n", msgbuf);  
    remove(fifo_nam);  
    printf ("Child process: EXIT\n");  
} else  
    printf ("Child process: No message for me (%s) !\n",  
           strerror(errno));  
}  
}
```

### Properties of FIFOs:

- for multiple processes
- access via names,
- definition at runtime,
- still existing after program termination
- slower.

→ Skat exercise using FIFOs!

→ fifo.c

### mq\_open()

```
mqptr = mq_open(mq_name, oflag, rights, attrib);
```

or

```
mqptr = mq_open(mq_name, oflag);
```

**mqptr:** Queue pointer

**mq\_name:** Name of the queue

**oflag:** Access mode

**rights:** Read- or write rights

**attrib:** attributes of the queue

### mq\_send()

```
mq_send(mqptr, msg, msg_len, prio);
```

**mqptr:** Queue pointer

**msg:** Pointer to date to send

**msg\_len:** number of bytes to send

**Prio:** Priority of the message

### mq\_receive ()

```
size = mq_receive(mqptr, msg, msg_len, prio_ptr);
```

**size:** size of the message in bytes

**mqptr:** Queue pointer

**msg:** Pointer to receive buffer

**msg\_len:** Size of receive buffer in bytes

**prio\_ptr:** Pointer to priority variable

## Message transmission with queues (I)

```
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h>
#include <sys/stat.h> #include <mqueue.h> #include <errno.h>

// Note (20200330): message queues are not implemented for
// Ubuntu 18.04 in a Windows 10 subsystem!

#define ZMAX 80
#define PRIO 0
#define MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH |S_IWOTH)

int main(void)
{
    char msgbuf[ZMAX], tmq_name[]="/tmq";
    unsigned int prio;
    pid_t npid;
    mqd_t tmq;
    size_t anz;
    struct mq_attr mqattr;
```

→ queues.c

## Message transmission with queues (II)

```
npid = fork();  
if (npid) {  
    sleep(1);  
    if ((tmq=mq_open(tmq_name, O_WRONLY)) == -1) {  
        printf ("Parent process: Can't open %s\n", tmq_name);  
        return EXIT_FAILURE;  
    }  
    printf ("Parent process: Please type a message:\n");  
    fflush (stdin) ;  
    scanf ("%[^\\n]", msgbuf);
```

→ queues.c

## Message transmission with queues (III)

```
if (mq_send(tmq, msgbuf, sizeof(msgbuf), PRIO) == -1) {
    printf ("Parent process: %s is not accessible\n",
            tmq_name);
    return EXIT_FAILURE;
}

if (mq_close(tmq) == -1) {
    printf ("Parent process: Can't close %s\n", tmq_name);
    return EXIT_FAILURE;
}

printf ("Parent process: EXIT\n");
}
```

## Message transmission with queues (IV)

```
else {
    mqattr.mq_maxmsg = 10;
    mqattr.mq_msgsize = ZMAX;
    mqattr.mq_flags = 0;
    if ((tmq=mq_open(tmq_name, O_CREAT|O_RDWR, MODE,
                      &mqattr)) == -1) {
        printf ("Child process: Can't create Message Queue
                %s\n", tmq_name);
        return EXIT_FAILURE;
    }
    printf ("Child process: Waiting for a message...\\n");
}
```

→ queues.c

## Message transmission with queues (V)

```
if ((anz= mq_receive(tmq, msgbuf, sizeof(msgbuf), &prio))>0) {  
    printf ("Child process: I received this message:\n  
            %s\n", msgbuf);  
    printf ("Child process: EXIT\n");  
}  
  
else printf ("Child process: No message for me!\n");  
  
if (mq_unlink(tmq_name) != 0) {  
    printf ("Child process: Can't remove Message Queue  
            %s (%s)\n", tmq_name, strerror(errno));  
    return EXIT_FAILURE;  
}  
return EXIT_SUCCESS;  
}  
  
→ Skat exercise using queues!
```

**Properties of message queues:**

- available in all OS,
- priorities
- well defined order of delivery,
- handling is similar to files but done by drivers (not file system).

[→ queues.c](#)

# Betriebssysteme / Operating Systems

## Processes and Threads

SS 2018

Prof. Dr.-Ing. Holger Gräßner

[11 OS-BS 2018 Processes and Threads.pptx]



# Threads, SMP, and Microkernels

## Processes and Threads

# Processes and Threads

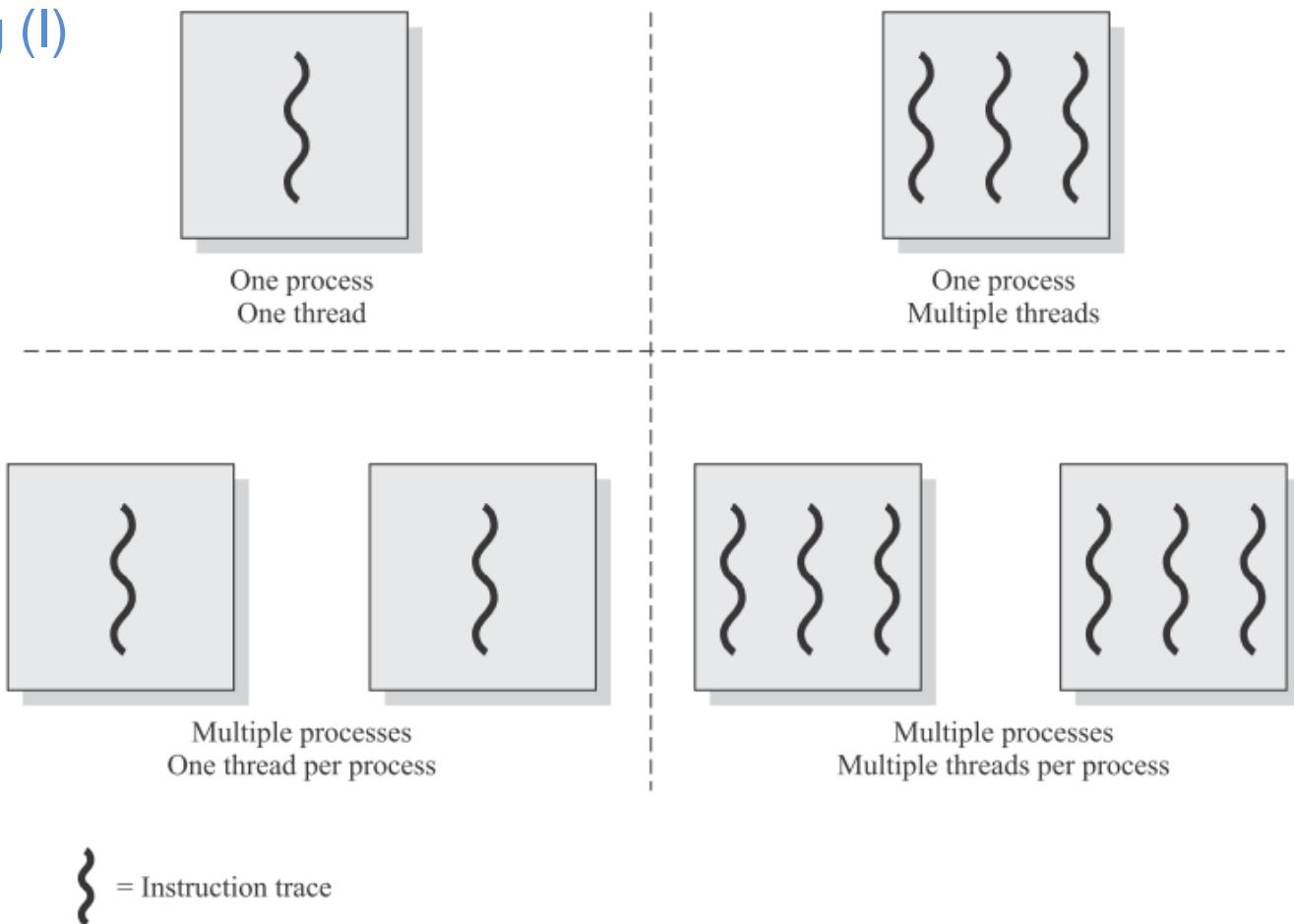
So far, processes have been presented as concept to handle

- **resource ownership** and
  - **scheduling**
- together.**

However, it makes sense to separate these issues.

To this end, contemporary OS distinguish between a concept called “**thread**” (for scheduling) and process (for resources).

## Multithreading (I)



**Multithreading** refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

### Multithreading (II)

In a **multithreaded** environment the following are associated with a process:

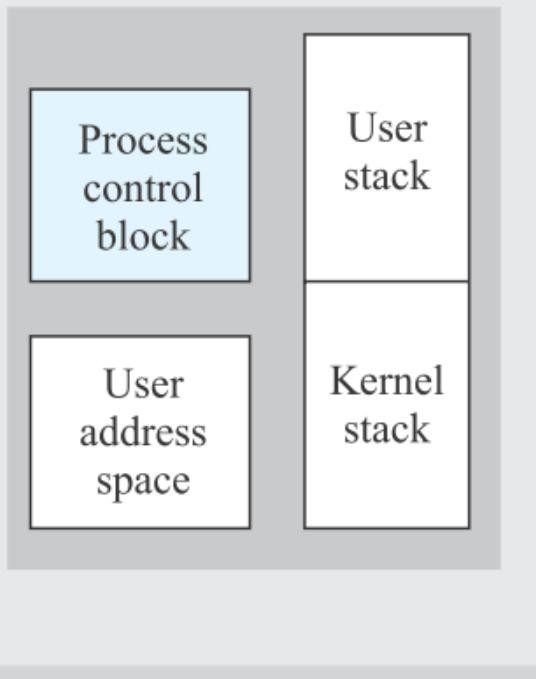
- A virtual address space that holds the process image.
- Protected access to processors, other processes (for communication), files, and I/O resources.

Within a process, there may be one or more threads, each with the following:

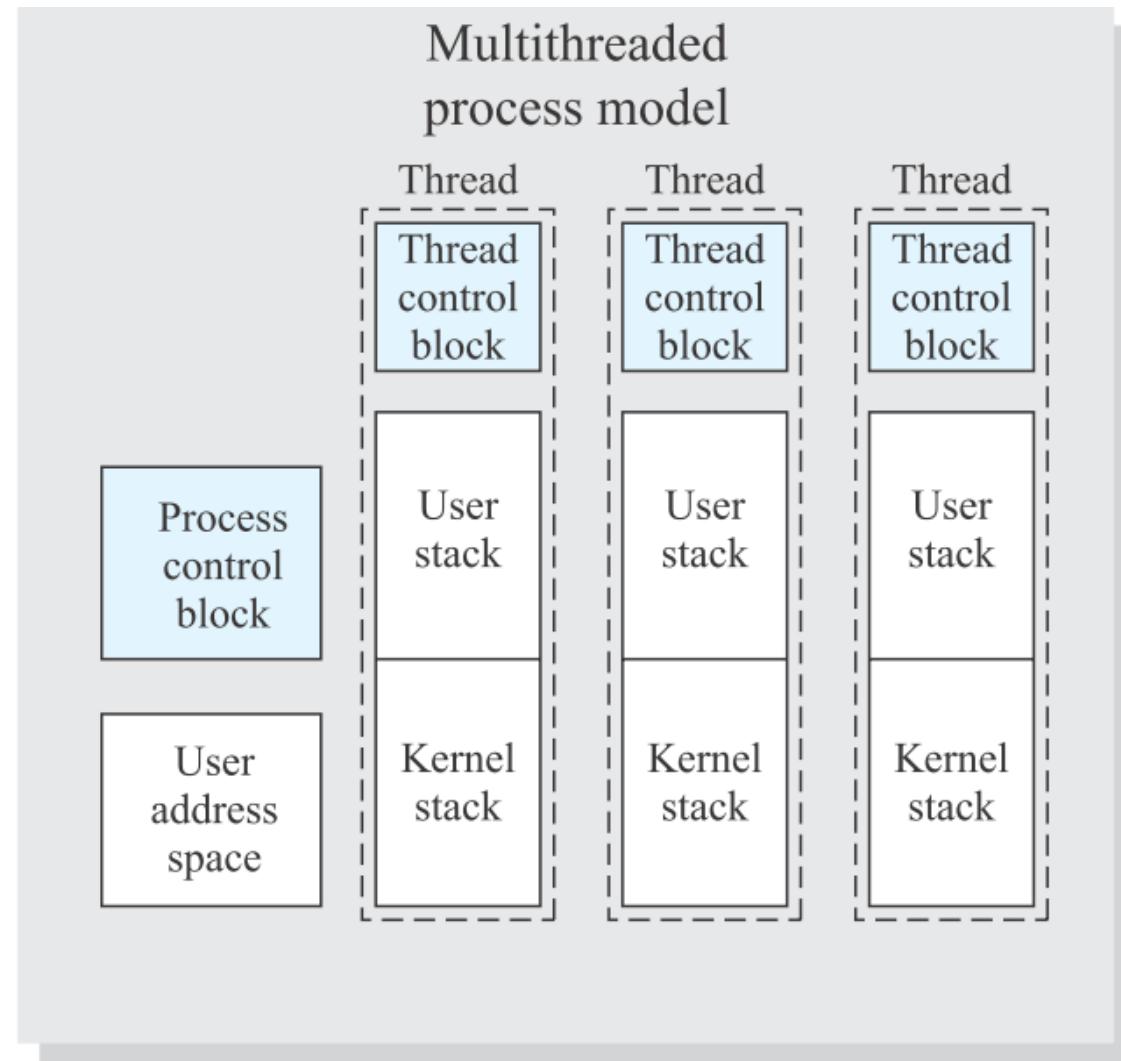
- A thread execution state (Running, Ready, . . . )
- A saved thread context when not running; one way to view a thread is as an independent PC operating within a process.
- An execution stack.
- Some per-thread static storage for local variables.
- Access to the memory and resources of its process, shared with all other threads in that process.

## Single Threaded and Multithreaded Process Models

Single-threaded  
process model



Multithreaded  
process model



## Why using Threads?

### PLUS

- Less creation overhead (a study: 10 times faster)
- Less termination time
- Less switching time
- Simplified communication. Threads can communicate directly without OS overhead, processes can't.

### Examples (Where threads are useful)

- Separation of fore- and background work
- Asynchronous processing
- Independent data processing
- Modular program structure

### Thread Functionality: Thread States (I)

In contrast to processes, it does not make much sense to suspend threads because the threads of a process share the same memory. Hence, they are Running, Blocked or Ready.

## Why?

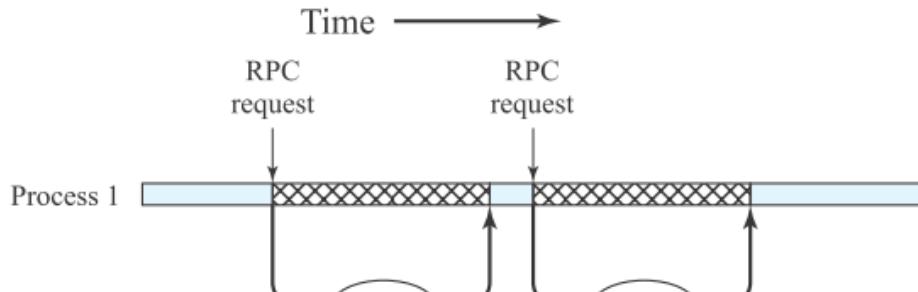
## Thread Functionality: Thread States (II)

In contrast to processes, it does not make much sense to suspend threads because the threads of a process share the same memory. Hence, they are Running, Blocked or Ready

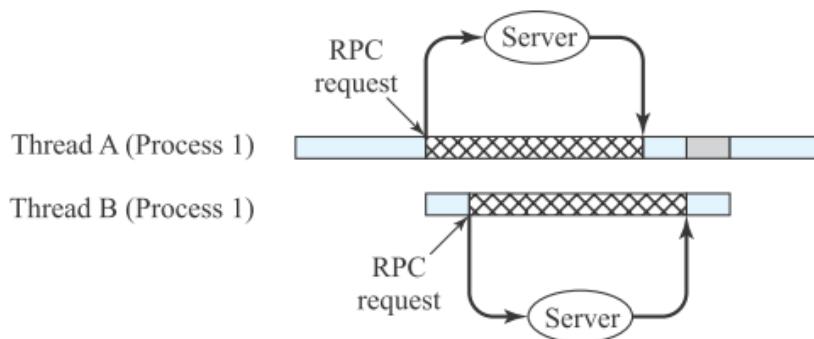
We have four operations:

- Spawn: Whenever a process is generated, a thread is generated, too. This thread (and all its ancestor threads) can spawn, i.e. to generate a new context, new stack and PC. New thread start in state Ready.
- Block: When a thread needs to wait for an event, it will block (saving its user context). The processor may now turn to the execution of another ready thread in the same or an different process.
- Unblock: When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
- Finish: When a thread completes, its register context and stacks are deallocated.

## Thread Functionality: Thread States (III)



(a) RPC using single thread



(b) RPC using one thread per server (on a uniprocessor)

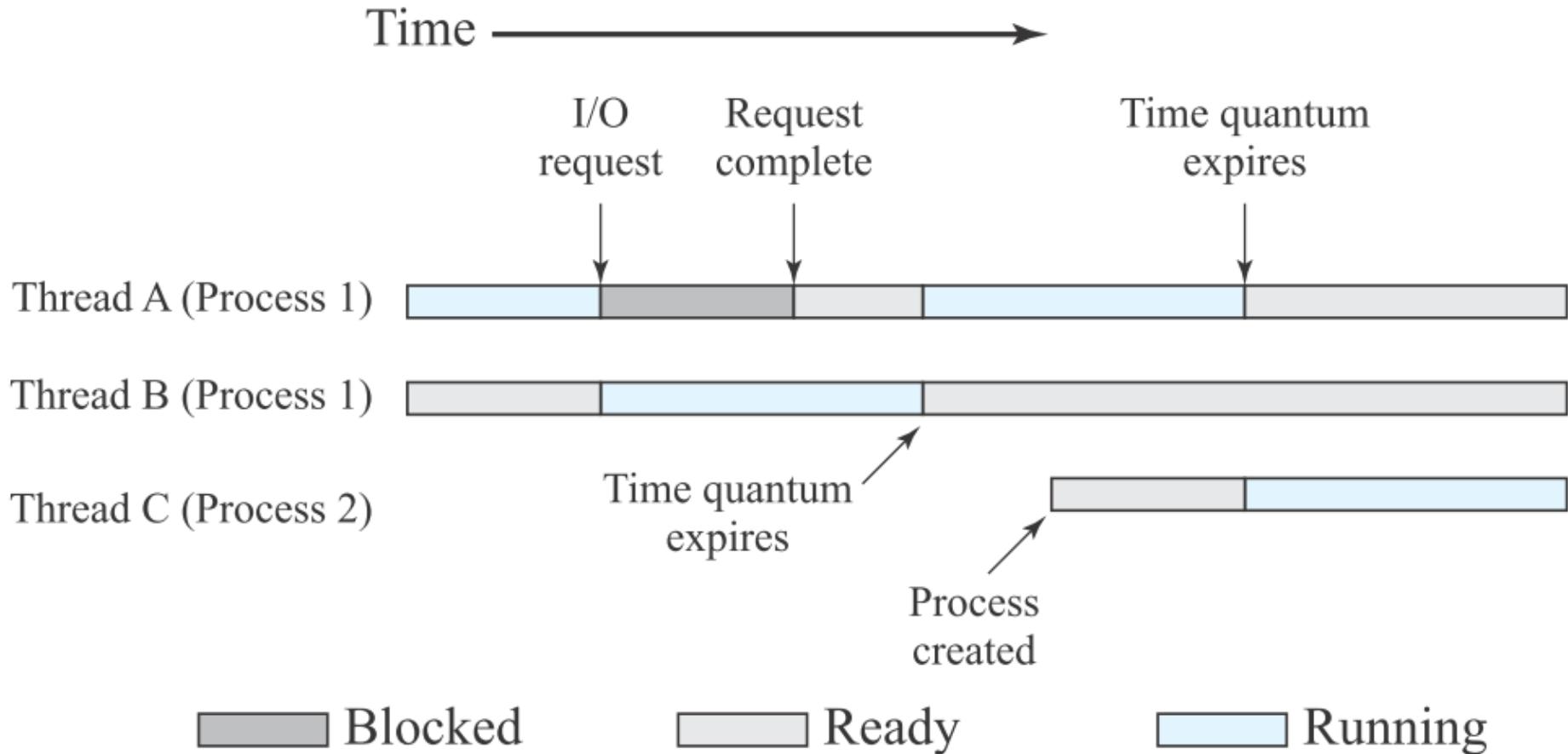
██████ Blocked, waiting for response to RPC

█████ Blocked, waiting for processor, which is in use by Thread B

████ Running

An example where multiple threads are useful to execute a remote procedure call (RPC) on a single processor system.

## Thread Functionality: Thread States (IV)



# Implementation of Threads (I)

To **implement threads** the OS designer has basically two options:

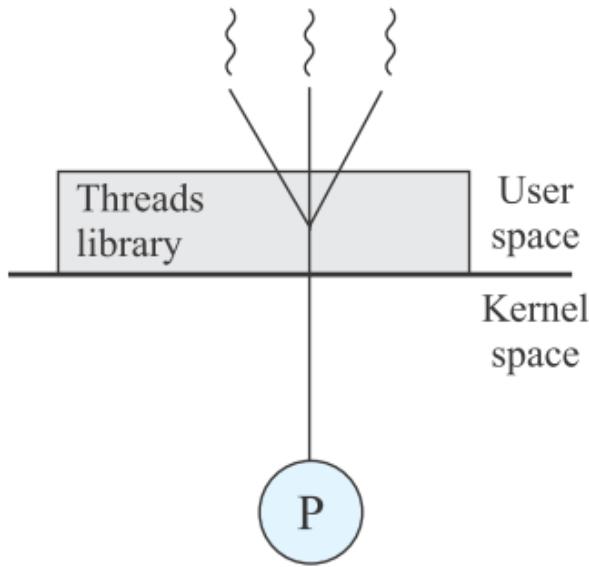
## User-level threads (ULT):

All of the work of thread management is done by the application and the kernel is not aware of the existence of threads.

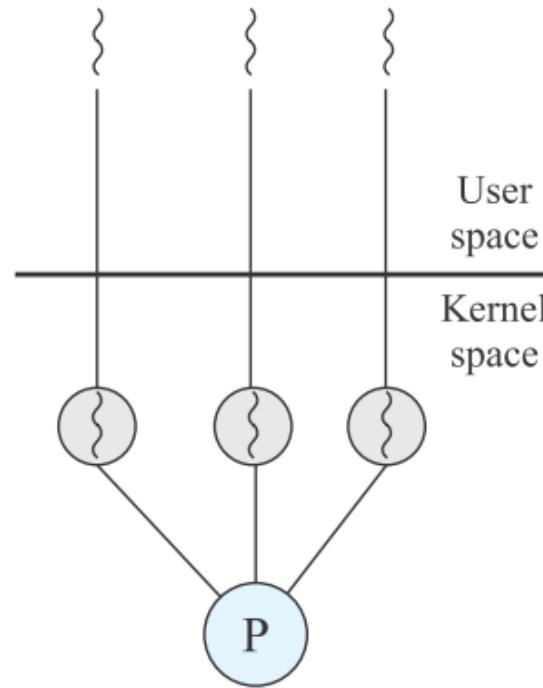
## Kernel-level threads (KLT):

All of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility.

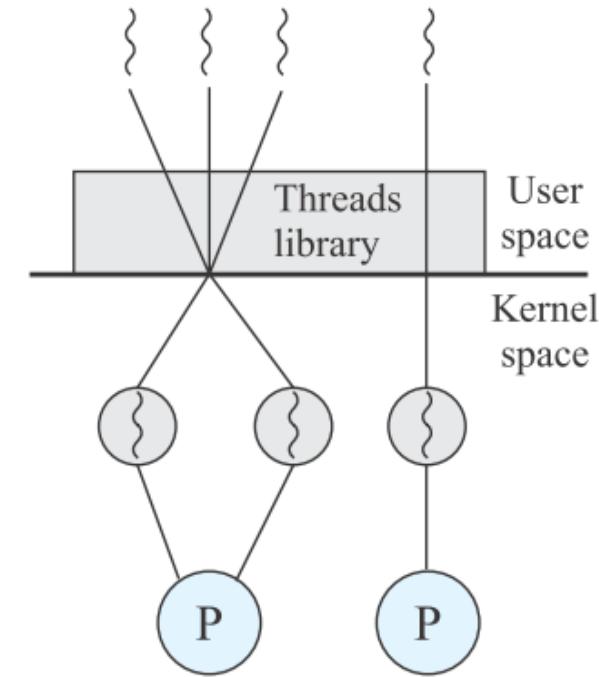
## Implementation of Threads (II)



(a) Pure user-level



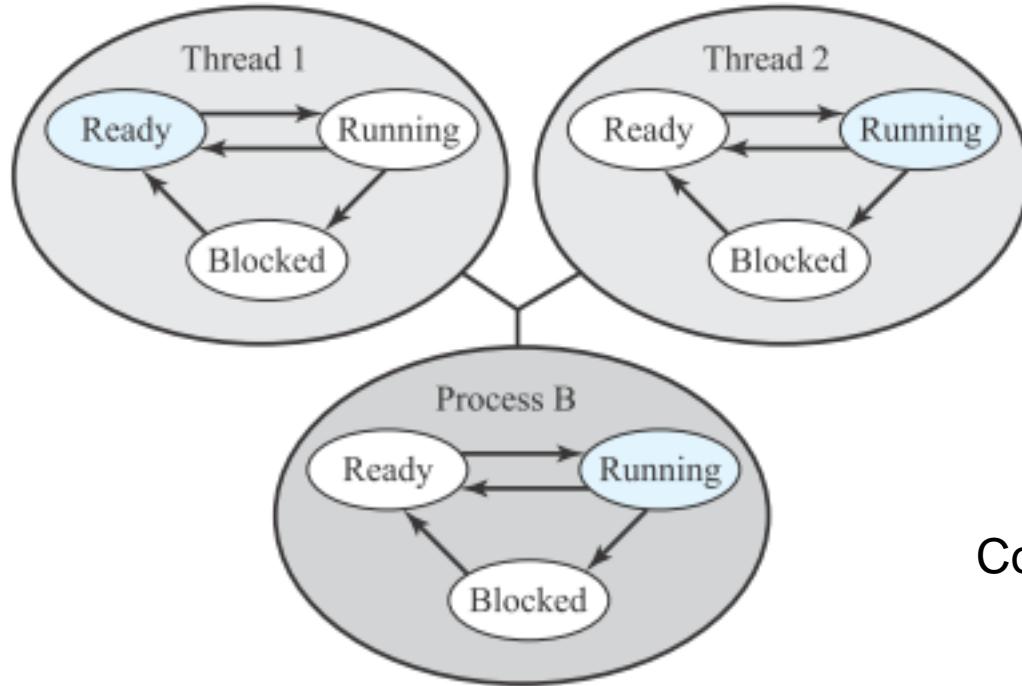
(b) Pure kernel-level



(c) Combined

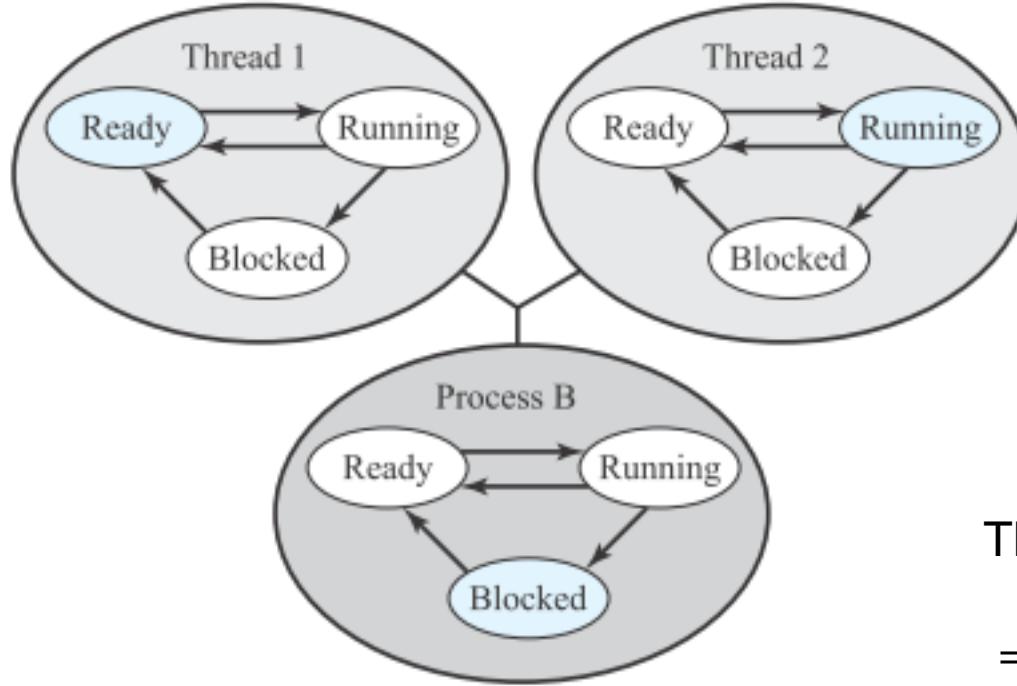
Now we consider **ULT** first.

## User Level Threads: 1. Example (I)



Consider a process B with two threads.

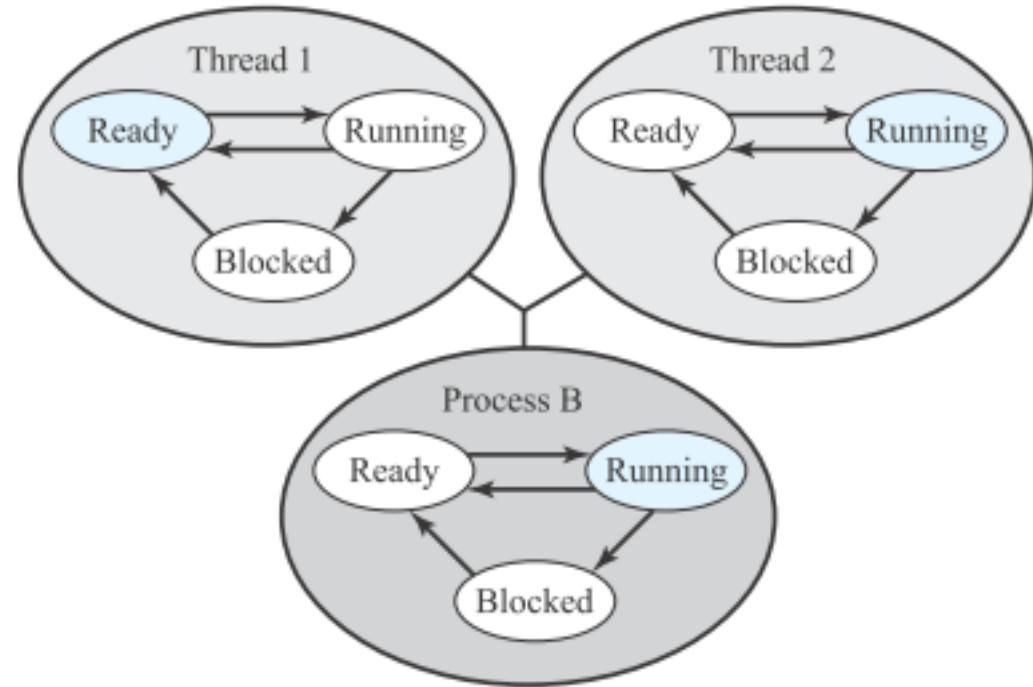
## User Level Threads: 1. Example (II)



Thread 2 makes a system call that blocks B.  
⇒ Thread is still running.  
However, B is blocked.

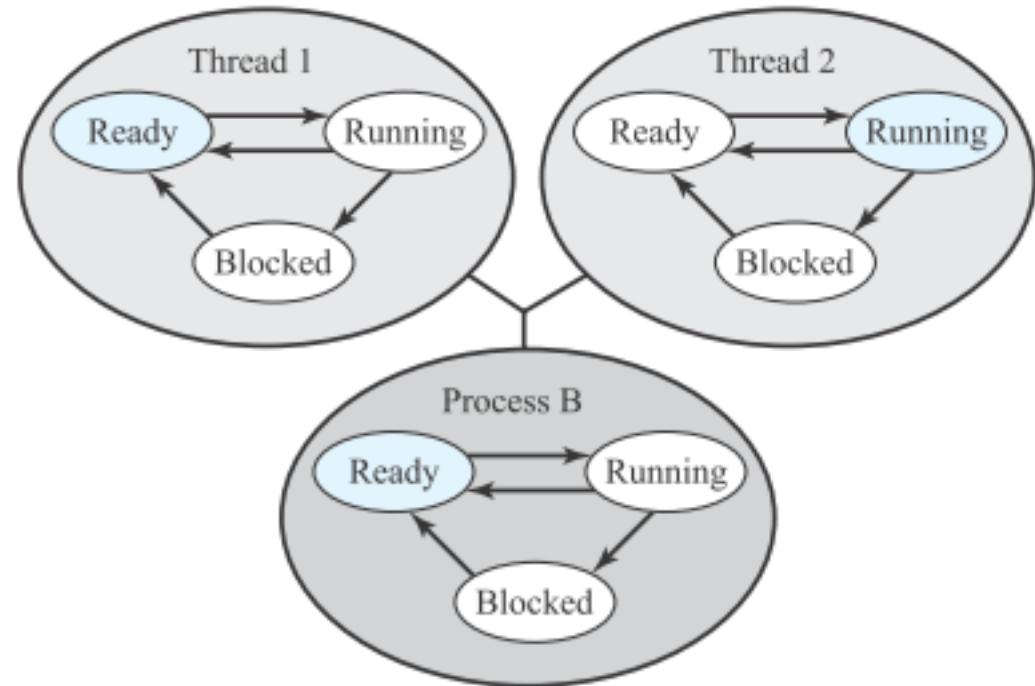
## User Level Threads: 2. Example (I)

Consider a process B with two threads.

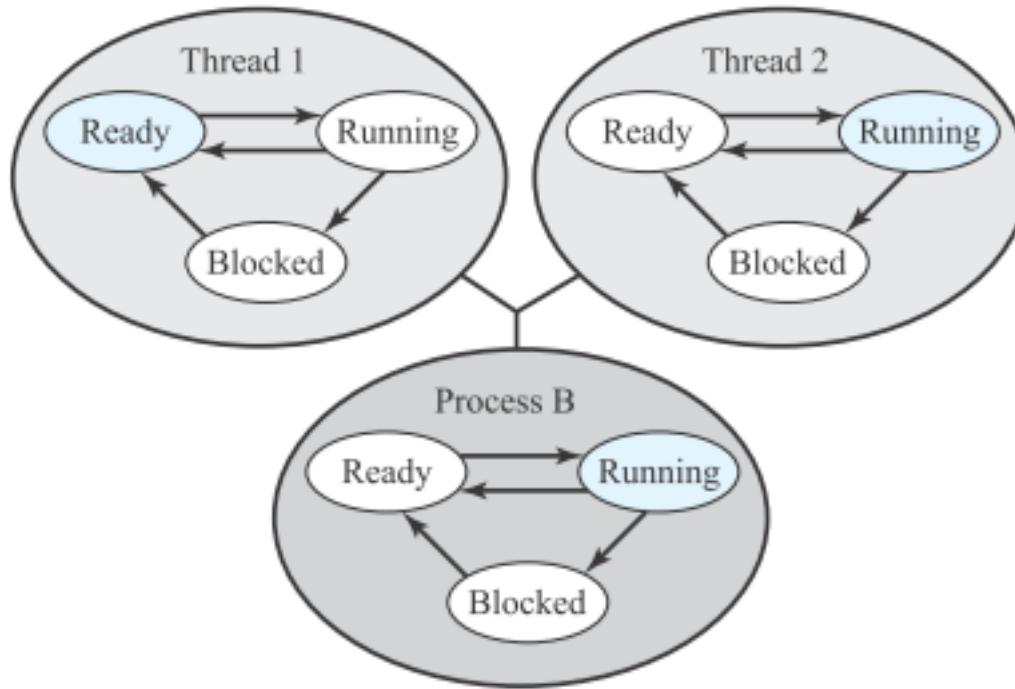


## User Level Threads: 2. Example (II)

B has exhausted its time slice. B becomes Ready.  
However, thread 2 is still in Running.

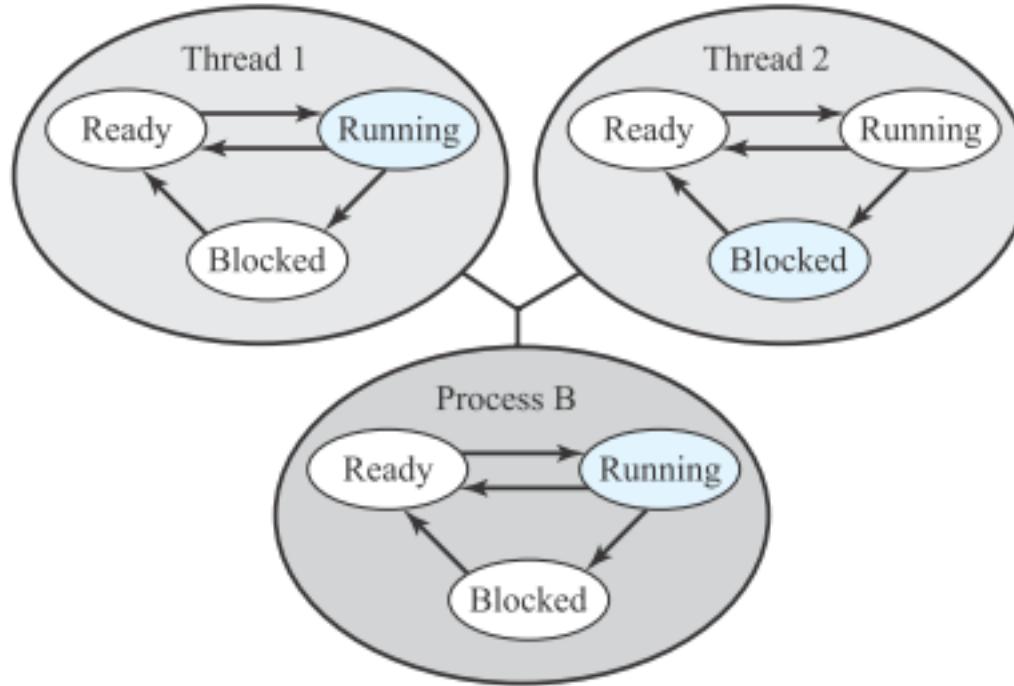


## User Level Threads: 3. Example (I)



Consider a process B with two threads.

## User Level Threads: 3. Example (II)



Thread 2 has reached point where it needs some action performed by thread 1 of process B. Thread 2 becomes Blocked and Thread 1 switches to Running.

# Advantages and Disadvantages of ULT

## PLUS

- Thread switching without kernel privileges. Saves the overhead of two mode switches.
- Application specific scheduling. The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler.
- Independence of the OS. No changes are required to the underlying kernel to support ULTs.

## MINUS

- Many system calls are blocking. Here, all threads of a process are blocked, if one is blocked due to a system call.
- In the presence of multiple processors the kernel cannot execute the threads of a given process on more than one processor.

## Advantages and Disadvantages of KLT

### PLUS

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- The kernel itself can be multithreaded.

### MINUS

- Mode switches required when changing the thread.

| Operation   | User-Level Threads | Kernel-Level Threads | Processes |
|-------------|--------------------|----------------------|-----------|
| Null Fork   | 34                 | 948                  | 11,300    |
| Signal-Wait | 37                 | 441                  | 1,840     |

### Combined Approaches

As usual when two approaches have advantages, combinations are possible. In the best case, you get the best of both worlds. However, this requires excellent knowledge how the application works together with the OS.

## Overview: Relationship between Threads and Processes

| Threads:<br>Processes | Description                                                                                                                          | Example Systems                                |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| 1:1                   | Each thread of execution is a unique process with its own address space and resources.                                               | Traditional UNIX systems                       |
| N:1                   | A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process. | Windows NT, Solaris, Linux, OS/2, OS/390, MACH |
| 1:M                   | A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.        | Ra (Clouds), Emerald                           |
| N:M                   | Combines attributes of M:1 and 1:M cases.                                                                                            | TRIX                                           |

The two latter cases are experimental approaches only.

The galactic imperium plans to send a stardestroyer to attack the rebel's base. This star destroyer shall host 1024 imperial clone warriors. At  $t=0$  there is just one soldier available: the captain. From his first birthday on, a clone warrior is able to clone once a year. The imperator wants to get the stardestroyer ready for action in short time.

The imperial command structure is very simple:

- Every warrior sends commands to his clones,
- there is no communication to the superior.

Write a Linux C-Programm with the following requirements:

- Every clone warrior has to be represented by a separate process.
- Commands have to be transmitted via uniquely (!) named message queues,
- There is an existing message queue "/Imperator" from the imperator to the captain.
- After the cloning phase, every clone warrior has to wait for commands to receive and to transmit to his inferiors.

Hints and requirements:

- Consider, how many soldiers are available in which year:  $t=0$  – just the captain,  $t=1$  – captain and his first clone, etc.
- Don't worry about error handling.

The galactic imperium plans to send a stardestroyer to attack the rebel's base. This star destroyer shall host 1024 imperial clone warriors. At t=0 there is just one soldier available: the captain. From his first birthday on, a clone warrior is able to clone once a year. The imperator wants to get the stardestroyer ready for action in short time.

The imperial command strucuture is very simple:

- Every warrior sends commands to his clones,
- there is no communication to the superior.

Write a Linux C-Programm with the following requirements:

- Every clone warrior has to be represented by a separate process.
- Commands have to be transmitted via uniquely (!) named message queues,
- There is an existing message queue "/Imperator" from the imperator to the captain.
- After the cloning phase, every clone warrior has to wait for commands to receive and to transmit to his inferiors.

Hints and requirements:

- Consider, how many soldiers are available in which year: t=0 – just the captain, t=1 – captain and his first clone, etc.
- Don't worry about error handling.

```
// Exercise „clone warriors“  
  
#define NUM 10  
  
#define SIZE_MSGBUF 5  
  
#define MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH |S_IWOTH)  
  
mqd_t QueueArray[NUM]; // queues to my clones  
  
void cleanQueueArray(void) { // support function for init: start with no queues  
    for (int i=0; i<NUM; i++) QueueArray[i] = 0;  
}  
  
int main(void) {  
    char cNameBossQueue[100] = "/Imperator"; // boss queue's default name  
    mqd_t BossQueue; // boss queue to receive commands of the father's process  
    struct mq_attr attr;  
    attr.mq_maxmsg = 10; attr.mq_msgsize = SIZE_MSGBUF; attr.mq_flags = 0;  
    int nPrio=0;  
    char cMsgbuf[SIZE_MSGBUF+1] = "";
```

```
cleanQueueArray(); // init: no queues to any clones at the beginning

// phase 1 / clone phase takes NUM years:
for (int i=0; i<NUM; i++) {
    pid_t npid_child = fork();
    if (npid_child > 0) { // Father. Create + store command channel to clone:
        char cQueue[100];
        sprintf(cQueue, "/Queue%d", npid_child);
        QueueArray[i] = mq_open(cQueue, O_CREAT|O_WRONLY, MODE, &attr);
    } else { // Child. Remember the name of the boss queue:
        sprintf(cNameBossQueue, "/Queue%d", getpid());
        cleanQueueArray(); // Child has no queues to clones currently
    }
}
```

```
// Phase 2 / battle phase. Receive and transmit orders:  
BossQueue = mq_open(cNameBossQueue, O_RDONLY, MODE, &attr);  
mq_receive(BossQueue, cMsgbuf, SIZE_MSGBUF, &nPrio);  
  
// Send orders to all of my clones:  
for (int i=0; i<NUM; i++) {  
    if (QueueArray[i] > 0) {  
        mq_send (QueueArray[i], cMsgbuf, strlen(cMsgbuf), 0);  
    }  
}  
// Cleanup work...  
return 0;  
}
```

# **Betriebssysteme / Operating Systems**

## **Network Access**

SS 2020

Prof. Dr.-Ing. Holger Gräßner

[12 OS-BS 2020 Network access.pptx]



## Conceptual models

- ISO/OSI: 7 abstraction layers, general reference model
- DoD, TCP/IP: 4 abstraction layers, internet's reference model

| OSI-Schicht     | TCP/IP-Schicht | Example           |
|-----------------|----------------|-------------------|
| 7: Application  | Application    | HTTP, FTP, Telnet |
| 6: Presentation |                |                   |
| 5: Session      |                |                   |
| 4: Transport    | Transport      | TCP, UDP          |
| 3: Network      | Internet       | IP                |
| 2: Data link    | Link           | Ethernet          |
| 1: Physical     |                |                   |

### Link layer (TCP/IP layer 1)

- Common network.
- Global unique addresses of participants (MAC).
- Time division multiplexing.
- Access control by CSMA/CD (in former times, no realtime approach).
- Nowadays: Switches, point-to-point-connections, full duplex operation  
→ no more collisions.

### Internet layer (TCP/IP layer 2)

- Does not depend on a specific transmission medium.
- Addressing of computers (IPv4: IP address, subnet mask)
- Packet orientation.
- Task: send IP packets to a specific address.

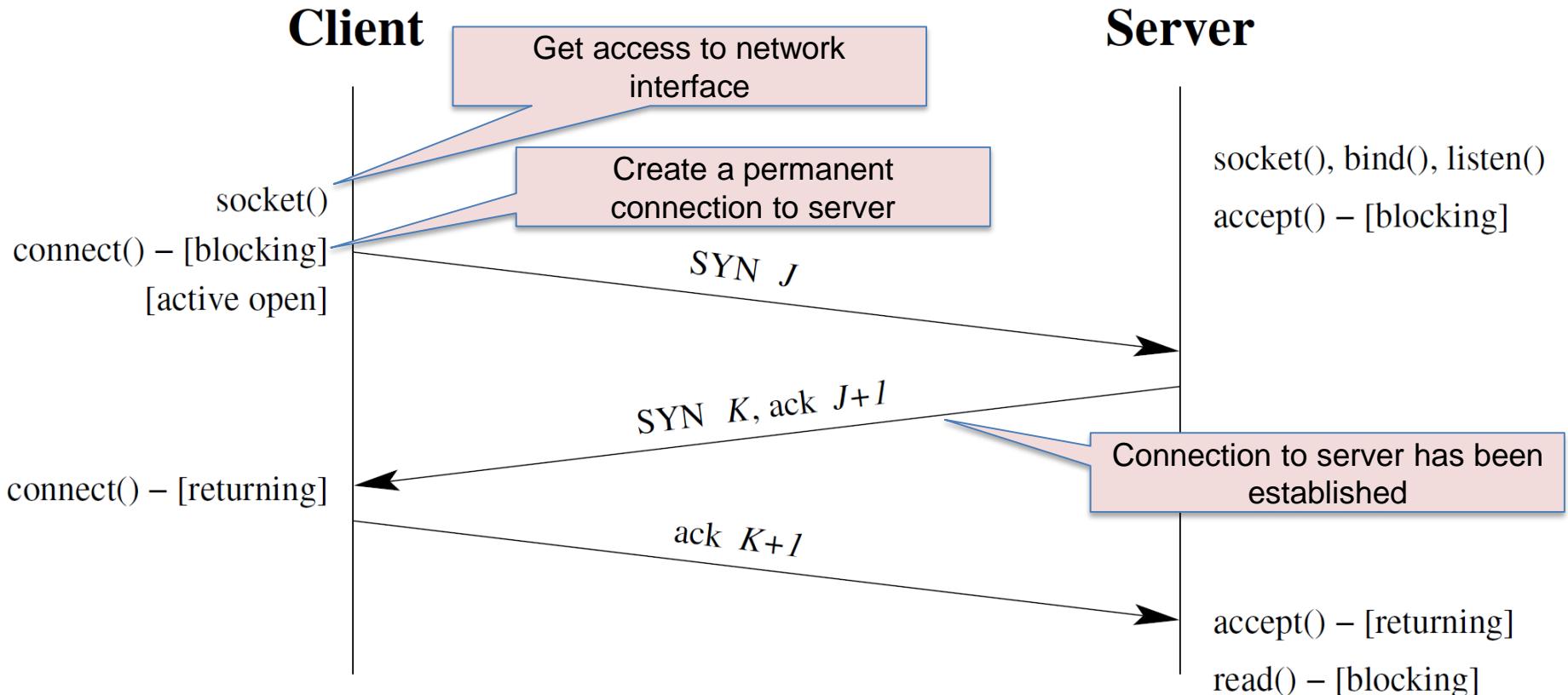
### Challenges:

- No reliable network infrastructure,
- No reliable transmission medium,
- Connections between nodes may change dynamically.
- Packets will be transmitted independently:
- Possible loss,
- Order may change randomly.
- Check sum for packets; packets may be rejected.

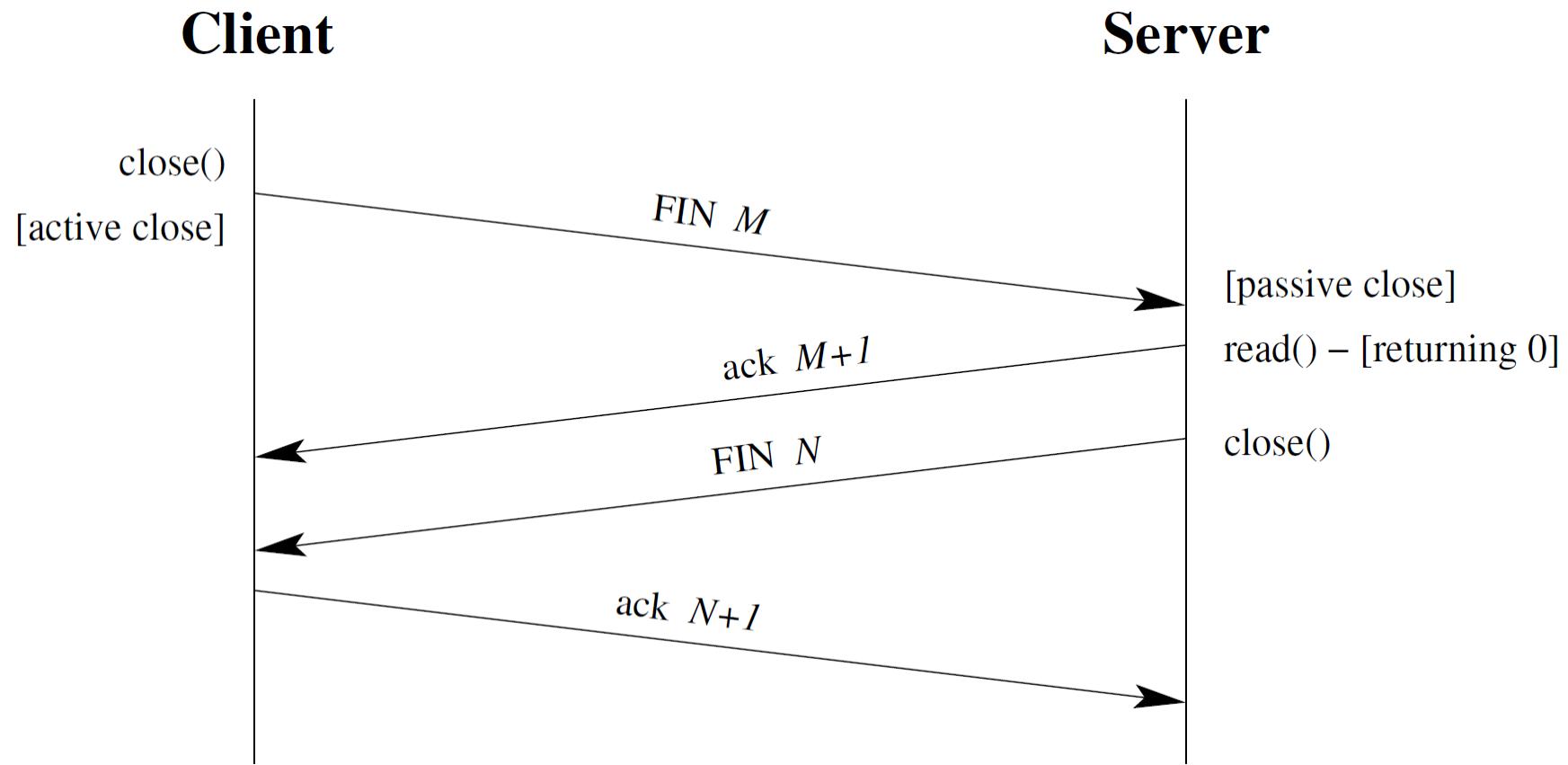
## Transport layer (TCP/IP layer 3)

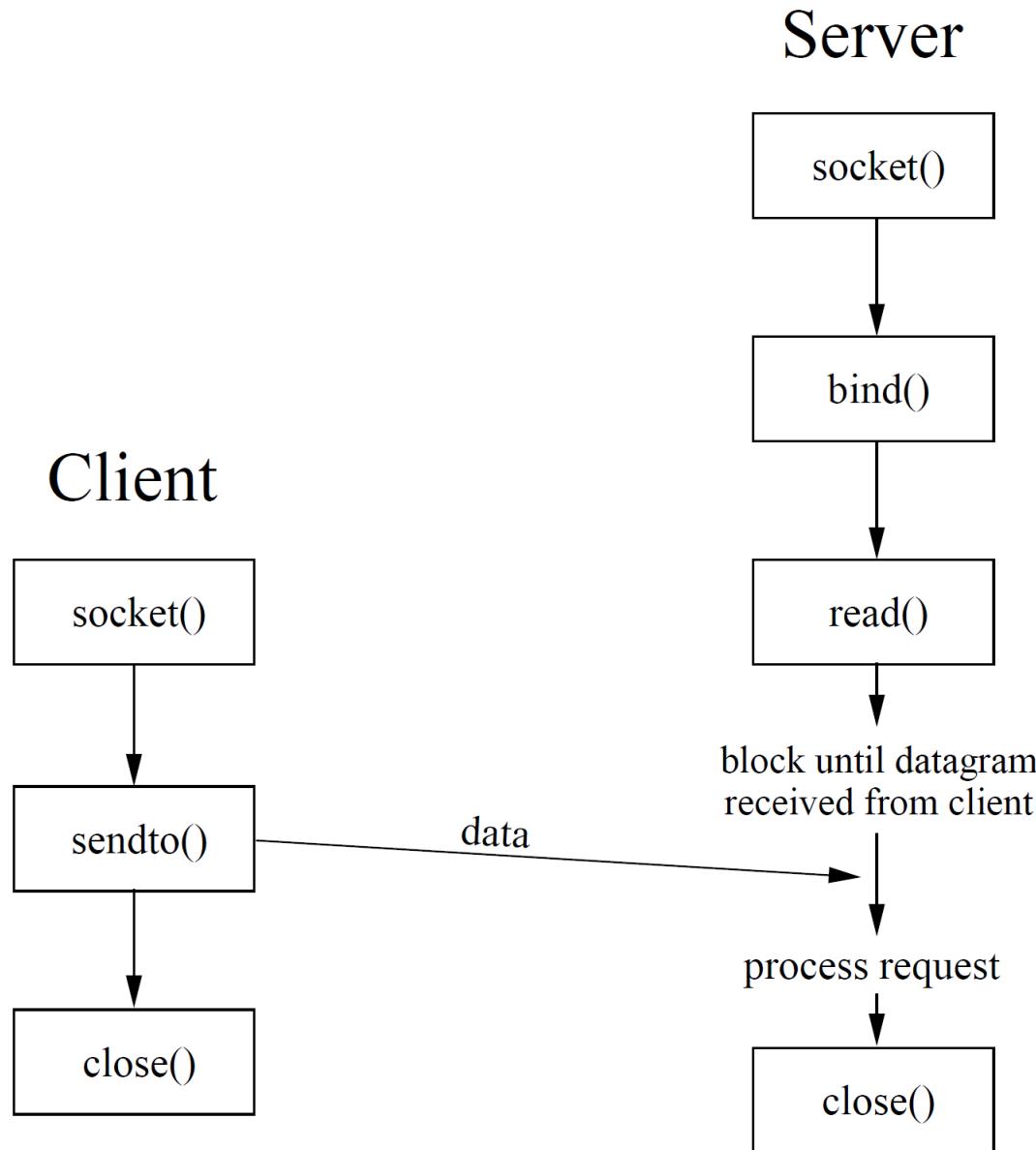
- Task: Get data integrity and reliability.
- Approach: Bidirectional, reliable data stream between two participants.
- Some reactions to guarantee data integrity.
- Checksum in packet's header.
- Sequence numbers will guarantee the correct order.
- Timeouts for packet acknowledge.
- Repeated transmission of packets, if necessary.
- Receiver will do packet buffering.
- Receiver will sort packets and eliminate doublets.

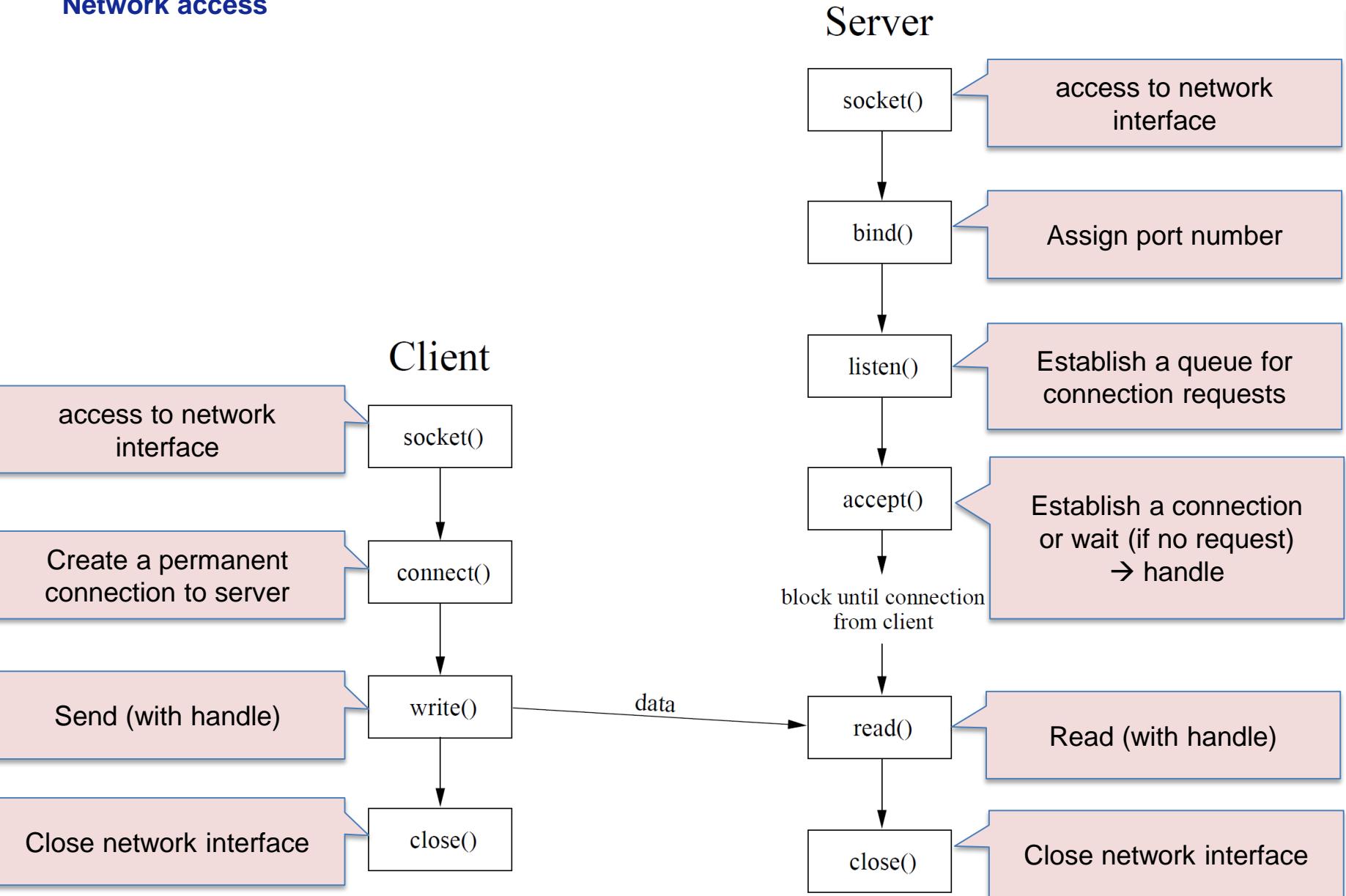
## TCP: connection establishment



## TCP connection termination







## socket()

```
sock = socket(domain, type, protocol);
```

**sock:** socket ID

**domain:** protocol family (**AF\_INET** for **TCP/IP**)

**type:** type of communication

(**SOCK\_DGRAM**: single packets,

**SOCK\_STREAM**: data stream)

**Protocol:** communication protocol

(0: **UDP** if **SOCK\_DGRAM**, **TCP** if **SOCK\_STREAM**)

## Adress structure

```
struct sockaddr_in {  
    short sin_family;          /* protocol family */  
    unsigned short sin_port;   /* port number */  
    struct in_addr sin_addr;   /* IP address */  
    char sin_zero [8];  
};
```

### Nameserver request via gethostbyname()

```
struct hostent* = gethostbyname(*name) ;
```

**hostent**: host structure, containing address etc.

**name**: computer name

## Function connect()

```
rval = connect(sock, *sockaddr, namelen);
```

**rval:** 0 if successful, -1 otherwise

**sock:** Socket ID

**sockaddr:** Internet socket address structure

**namelen:** size of address structure in bytes

## Simple client program (I)

```
#include <sys/socket.h> #include <netinet/in.h> #include <netdb.h> #include <stdio.h>
#include <string.h> #include <stdlib.h> #include <unistd.h>

#define ZMAX 80

int main(int argc, char *argv[])
{
    char string [ZMAX];
    int sock;
    struct sockaddr_in server;
    struct hostent *hp=NULL;
```

→ [client-Programm.c](#)

## Simple client program (II)

```
if (argc != 3) {  
    fprintf (stderr, "Usage: %s hostname  
            portnumber\n", argv[0]);  
    return EXIT_FAILURE;  
}  
  
sock = socket(AF_INET,SOCK_STREAM,0);  
if (sock < 0) {  
    perror("Socket");  
    return EXIT_FAILURE;  
}  
  
server.sin_family = AF_INET;
```

→ [client-Programm.c](#)

## Simple client program (III)

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
    fprintf (stderr ,"%s: %s\n",argv[1],
            hstrerror(h_errno));
    return EXIT_FAILURE;
}
memcpy(&server.sin_addr,hp->h_addr,hp->h_length);
server.sin_port = htons(atoi(argv[2]) );
if (connect(sock,(struct sockaddr*)&server,
            sizeof(server)) < 0) {
    perror("Establishing connection");
    return EXIT_FAILURE;
}
```

→ [client-Programm.c](#)

## Simple client program (IV)

```
printf ("Client: Please type a message:\n ");
fflush (stdin) ;
scanf ("%[^\\n]",string) ;
if (write(sock, string ,sizeof( string ) ) < 0)
    perror ("Sending data");
close(sock);
return EXIT_SUCCESS;
}
```

→ [client-Programm.c](#)

## bind()

```
rval = bind(sock, *sockaddr, namelen);
```

**rval:** 0: success, -1: failure

**sock:** Socket Id

**sockaddr:** Internet socket address structure

**namelen:** size of address structure in bytes

### listen()

```
rval = listen(sock, queuelen);
```

**rval:** 0: success, -1: failure

**sock:** Socket ID

**queuelen:** lenght of connecting queue

### accept()

```
msgsock = accept(sock, *sockaddr, namelen);
```

**msgsock:** handle of connection

**sockaddr:** Internet socket address structure

**namelen:** size of address structure in bytes

## A simple, iterative server program (I)

```
#define PORT 5000

#include <sys/socket.h> #include <netinet/in.h> #include <stdio.h>
#include <string.h> #include <unistd.h> #include <stdlib.h>

int main()
{
    int sock;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval ;
```

→ [simple\\_iterative\\_server.c](#)

## A simple, iterative server program (II)

```
sock = socket(AF_INET,SOCK_STREAM,0);
if (sock < 0) {
    perror("Stream Socket");
    return EXIT_FAILURE;
}

server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(PORT); /* htons: swap bytes */
if (bind(sock,(struct sockaddr *)&server,
          sizeof(server))) {
    perror("Port assignment");
    return EXIT_FAILURE;
}
```

→ simple\_iterative\_server.c

## A simple, iterative server program (III)

```
printf ("Server: Receipt on port #%-d\n",
        ntohs(server.sin_port));
listen (sock,5);
do {
    msgsock = accept(sock,NULL,0);
    if (msgsock == -1) {
        perror("Connection establishment");
        return EXIT_FAILURE;
    }
```

→ [simple\\_iterative\\_server.c](#)

## A simple, iterative server program (IV)

```
else do {  
    memset(buf, 0, sizeof(buf));  
    if (( rval = read(msgsock,buf,1024)) < 0)  
        perror("Data receipt");  
    else if (rval == 0)  
        printf ("Server: Connection terminated\n");  
    else  
        printf ("Client: %s\n", buf);  
        /* do server's job here */  
    } while (rval > 0);  
close(msgsock);  
} while (1);  
}
```

→ `simple_iterative_server.c`

## Internet Superdaemon: file /etc/inetd.conf (extract)

```
#Service  socktype  prot  wait?  user    full-path          full-commandline
#-----
ftp      stream     tcp    nowait root   /usr/sbin/ftpd  ftpd -l
telnet   stream     tcp    nowait root   /usr/sbin/tcpd  /usr/sbin/telnetd
talk     dgram      udp    wait    nobody  /usr/sbin/tcpd  /usr/sbin/ktalkd
myserv   stream     tcp    nowait nobody  /tmp/myservtcp myservtcp
myserv   dgram      udp    wait    nobody  /tmp/myservudp myservudp
```

## Note:

- The internet super daemon will handle all the boring connection stuff.
- If asked for one of the services in the first column, the internet super daemon will execute the **corresponding program in the 6. column.** of **inetd.conf** (these programs do the specific work of user's data).
- The corresponding program just has to use **stdin** and **stdout**.

### Internet Superdaemon: file /etc/services (extract)

```
ftp      21/tcp
telnet   23/tcp
talk     517/udp
tserv    5001/tcp # our test server
tserv    5001/udp # our test server
```

## Internet Superdaemon: Server program (I)

```
#include <stdio.h> #include <stdlib.h> #include <string.h>
#include <unistd.h> #include <sys/uio.h>

#define BUFSIZE 1024

int main()
{
    int rval , slen ;
    char *rbuf=NULL, *wbuf=NULL;

    rbuf = (char*)malloc(BUFSIZE*2);
    wbuf = rbuf+BUFSIZE;
    do {
        if (( rval = read(0,rbuf,1024)) < 0)
            return EXIT_FAILURE;
        else if (rval == 0)
            return EXIT_SUCCESS;
    }
```

→ [server\\_inetd.c](#)

## Internet Superdaemon: Server program (II)

```
else {
    slen = sprintf (wbuf, "Receipt: \"%s\\\"", rbuf)+1;
    if (write(1,wbuf,slen) < 0)
        return EXIT_FAILURE;
}
} while (rval > 0);
return EXIT_SUCCESS;
}
```

→ [server\\_inetd.c](#)

# Betriebssysteme / Operating Systems

## Ressource allocation

SS 2018

Prof. Dr.-Ing. Holger Gräßner

13 OS-BS 2018 Ressource allocation.pptx]

## Mutex functions

```
rval = pthread_mutex_init(pthread_mutex_t *mutex, NULL);
```

```
rval = pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
rval = pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
rval = pthread_mutex_unlock(pthread_mutex_t *mutex);
```

**mutex:** Pointer to state variable

## Ressource allocation using a mutex (I)

```
#define ATIME 5
#define NUM 60
#define FALSE 0
#define TRUE (!FALSE)
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h> #include
<signal.h> #include <pthread.h> #include <sched.h> #include <errno.h>

pthread_mutex_t mutex;
unsigned char myArray[NUM+1];
volatile sig_atomic_t flag = FALSE;

void *thread2(void*) {
    void sigh(int signo)
    {
        flag = TRUE;
    }
}
```

→ [mutex.c](#)

## Ressource allocation using a mutex (II)

```
int main(void)
{
    pthread_t tid ;
    int i ;
    unsigned char *ptr=NULL;

    if (pthread_mutex_init(&mutex,NULL) != 0) {
        fprintf (stderr , "Error pthread_mutex_init(): %s\n",
                 strerror(errno)) ;
        return EXIT_FAILURE;
    }

    if (pthread_create(&tid,NULL,thread2,NULL) != 0) {
        fprintf (stderr , "Error pthread_create() : %s\n",
                 strerror(errno)) ;
        return EXIT_FAILURE;
    }
}
```

→ mutex.c

## Ressource allocation using a mutex (III)

```
if (( ptr = (unsigned char*)malloc((NUM+1)*sizeof(
        unsigned char))) == NULL) {
    fprintf (stderr , "Error malloc(): %s\n", strerror(errno)) ;
    return EXIT_FAILURE;
}

signal(SIGALRM,sigh);
printf ("Creating pattern - please wait...\n");
memset(ptr,'|',NUM); // vertical line: by original thread
*(ptr+NUM) = '\0';
alarm(ATIME); // raise SIGALARM after ATIME
while (1) {
    if (pthread_mutex_lock(&mutex) != 0) {
        fprintf (stderr , "Error pthread_mutex_lock(): %s\n",
                strerror(errno));
        return EXIT_FAILURE;
    }
}
```

→ mutex.c

## Ressource allocation using a mutex (IV)

```
for ( i=0;i<NUM;i++)
    myArray[ i ] = ptr[ i ];
for ( i=0;i<NUM;i++)
    ptr[ i ] = myArray[ i ];
if (pthread_mutex_unlock(&mutex) != 0) {
    fprintf (stderr , "Error pthread_mutex_unlock(): %s\n",
             strerror(errno));
    return EXIT_FAILURE;
}
if (flag) {
    printf ("Array: %s\n", ptr);
    break;
}
return EXIT_SUCCESS;
}
```

→ mutex.c

## Ressource allocation using a mutex (V)

```
void *thread2(void *data)
{
    int i ;
    unsigned char *ptr=NULL;

    if (( ptr = (unsigned char*)malloc( (NUM+1)*sizeof(
        unsigned char))) == NULL) {
        fprintf (stderr , "Error malloc(): %s\n", strerror(errno)) ;
        exit (EXIT_FAILURE);
    }

    memset(ptr, '-' ,NUM); // horizontal lines: created by thread 2
    * (ptr+NUM) = '\0' ;
```

→ mutex.c

## Ressource allocation using a mutex (VI)

```
while (1) {  
    if (pthread_mutex_lock(&mutex) != 0) {  
        fprintf (stderr , "Error pthread_mutex_lock(): %s\n",  
                strerror(errno));  
        exit (EXIT_FAILURE);  
    }  
    for ( i=0;i<NUM;i++)  
        myArray[ i ] = ptr[ i ];  
    for ( i=0;i<NUM;i++)  
        ptr[ i ] = myArray[ i ];  
    if (pthread_mutex_unlock(&mutex) != 0) {  
        fprintf (stderr , "Error pthread_mutex_unlock(): %s\n",  
                strerror(errno));  
        exit (EXIT_FAILURE);  
    }  
}
```

→ mutex.c

## Semaphor functions (I)

```
rval = sem_init(sptr , PSHARED , inval); // create semaphor  
                                // (make *sptr to become a semaphor)
```

**sptr:** pointer to semaphor

**PHARED:** 0: only for threads of this process. 1: also for other processes (shared memory)

**inval:** initial value of counter

```
rval = sem_wait(sptr); // request for ressource allocation:  
                      // counter -= 1; blocks while counter = 0
```

```
rval = sem_post(sptr); // release ressource: counter += 1;
```

```
rval = sem_destroy(sptr); // delete semaphor
```

## Semaphor functions (II)

```
sptr = sem_open(sem_name, oflag, rights, value);
```

or

```
sptr = sem_open(sem_name, oflag);
```

**sptr:** Pointer to the generated semaphor

**sem\_name:** The name of the semaphor

**oflag:** Access mode

**rights:** Read-/write rights

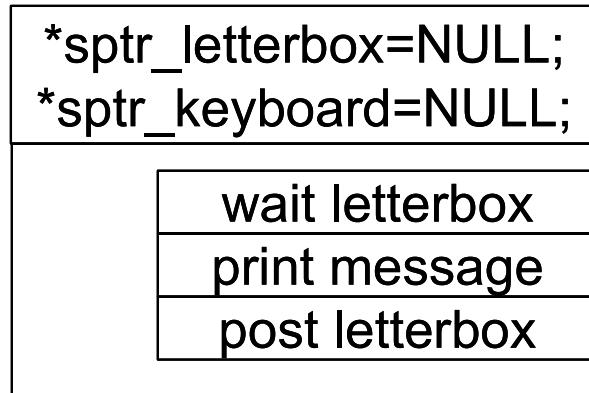
**value:** Initial value

```
rval = sem_close(sptr);           // close connection to semaphor
```

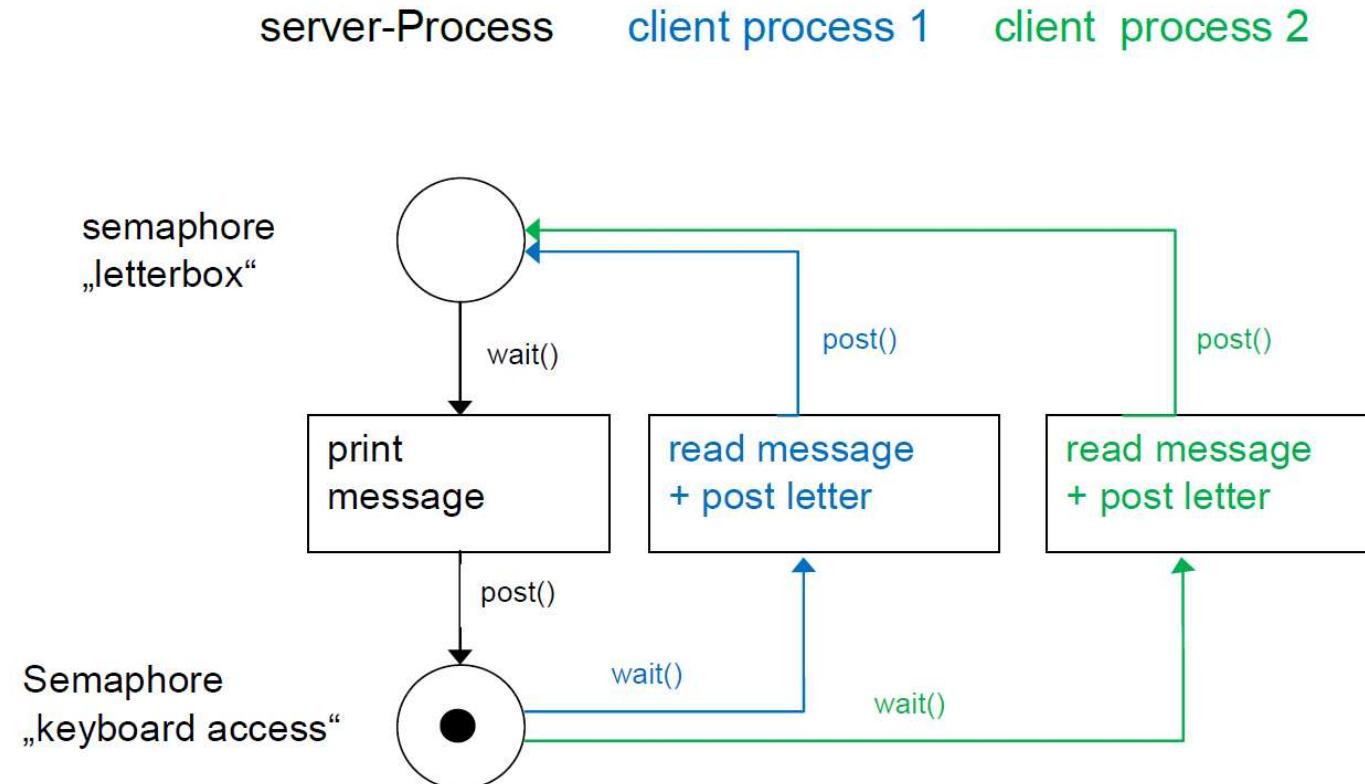
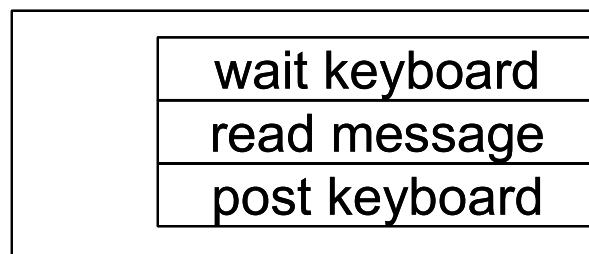
```
rval = sem_unlink(sem_name);    // delete semaphor: forget the name  
                                // as soon as all using processes have called sem_close()
```

## Process synchronisation with semaphor, example

Server:



Client:



## Process synchronisation with semaphor, server (I)

```
#include <stdio.h> #include <unistd.h> #include <stdlib.h>    #include <errno.h>
#include <fcntl.h> #include <string.h> #include <sys/stat.h> #include <sys/mman.h>
#include <semaphore.h> #include <signal.h>

#define FALSE 0
#define TRUE (!FALSE)

#define MODE (S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH)
#define PSHARED 1
#define S_STVAL1 0
#define S_STVAL2 1
#define MSGLEN 100

volatile sig_atomic_t flag = FALSE;

void sigh(int sig_num)
{
    flag = TRUE;
}
```

→ [semaphore\\_server.c](#)

## Process synchronisation with semaphor, server (II)

```
int main(void)
{
    char memfile[] = "shtrans";
    char *ptr=NULL, *hptr=NULL;
    sem_t *sptr_letterbox=NULL, *sptr_keyboard=NULL;
    int fd;

    if ((fd=shm_open(memfile,O_RDWR|O_CREAT,MODE)) == -1) {
        printf ("Error: 'shm open(create) '\n");
        return EXIT_FAILURE;
    }

    if (ftruncate(fd,2*sizeof(sem_t)+MSGLEN) == -1) {
        printf ("Error: 'ftruncate() '\n");
        return EXIT_FAILURE;
    }
}
```

→ [semaphore\\_server.c](#)

## Process synchronisation with semaphor, server (III)

```
if (( ptr=(char*)mmap(0, 2*sizeof(sem_t)+MSGLEN,
                      PROT_READ|PROT_WRITE,MAP_SHARED, fd, 0)) == (char*)-1) {
    printf ("Error: mapping region impossible\n");
    return EXIT_FAILURE;
}

close(fd) ;
printf ("Server: Type <cntrl><c> to terminate!\n");
signal(SIGINT, sigh); // <cntrl> <c> ==> Handler sigh
sptr_letterbox = (sem_t*) ptr; // create semaphore "letterbox"
sptr_keyboard = sptr_letterbox+1;// create semaphore „keyboard“
hptr = (char*)(sptr_keyboard+1);

if (sem_init(sptr_letterbox, PSHARED, S_STVAL1) == -1) {
    printf ("Could not init semaphor 'letterbox' (%s)\n",
           strerror(errno));
    return EXIT_FAILURE;
}
```

→ [semaphore\\_server.c](#)

## Process synchronisation with semaphor, server (IV)

```
if (sem_init(sptr_keyboard, PSHARED, S_STVAL2) == -1) {
    printf ("Could not init semaphor 'keyboard' (%s)\n",
            strerror(errno));
    return EXIT_FAILURE;
}

for (;;) {
    printf ("Server: Waiting for a message ... \n");
    sem_wait(sptr_letterbox); // wait for a letter in letterbox
    if (! flag) {
        printf ("Server: I received this message:\n%s\n", hptr);
        sem_post(sptr_keyboard); // release keyboard to allow
                                // any client to enter a message
    }
}
```

→ [semaphore\\_server.c](#)

## Process synchronisation with semaphor, server (V)

```
    } else {  
        flag = FALSE;  
        if (sem_destroy(sptr_letterbox) == -1) {  
            flag = TRUE;  
            printf ("Could not destroy semaphor 'letterbox'  
                    (%s)\n", strerror(errno));  
        }  
        if (sem_destroy(sptr_keyboard) == -1) {  
            flag = TRUE;  
            printf ("Could not destroy semaphor 'keyboard' (%s)\n",  
                    strerror(errno));  
        }  
    }
```

→ [semaphore\\_server.c](#)

## Process synchronisation with semaphor, server (VI)

```
    if (munmap(ptr,2*sizeof(sem_t)+MSGLEN) == -1) {  
        flag = TRUE;  
        printf ("Could not unmap shared memory (%s) \n",  
               strerror(errno)) ;  
    }  
  
    if (shm_unlink(memfile) == -1) {  
        flag = TRUE;  
        printf ("Could not unlink shared memory (%s) \n",  
               strerror(errno));  
    }  
  
    if (flag)  
        return EXIT_FAILURE;  
    else  
        return EXIT_SUCCESS;  
}  
  
}
```

→ [semaphore\\_server.c](#)

}

17

## Process synchronisation with semaphor, client (I)

```
#include <stdio.h> #include <unistd.h> #include <stdlib.h> #include <fcntl.h>
#include <string.h> #include <sys/mman.h> #include <semaphore.h>

#define MSGLEN 100

int main(void)
{
    sem_t *sptr_letterbox=NULL, *sptr_keyboard=NULL;
    int fd, anz;
    char memfile[] = "shtrans", *ptr=NULL, *hptr=NULL;
    if ((fd=shm_open(memfile, O_RDWR, 0)) == -1) {
        printf ("Error: Can't open shared momory\n");
        return EXIT_FAILURE;
    }
```

→ [semaphore\\_client.c](#)

## Process synchronisation with semaphor, client (II)

```
if (( ptr=(char*)mmap(0, 2*sizeof(sem_t)+MSGLEN,
                      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == (char*)-1) {
    printf ("Error: 'mapping' impossible\n");
    return EXIT_FAILURE;
}

close(fd) ;
sptr_letterbox = (sem_t*) ptr;      // create semaphore "letterbox"
sptr_keyboard = sptr_letterbox+1;   // create semaphore "keyboard"
hptr = (char*) (sptr_keyboard+1);
for (;;) {
    sem_wait(sptr_keyboard); // wait for exclusive keyboard access
    printf ("Client #%-d: Please enter a message (empty line will
            terminate client):\n", getpid());
    fgets (hptr, MSGLEN, stdin);
```

→ [semaphore\\_client.c](#)

## Process synchronisation with semaphor, client (III)

```
if (*hptr == '\n') {  
    sprintf (hptr, "Client #%d to server: Goodbye...",  
            getpid());  
    sem_post(sptr_letterbox); // to server: 'letter in the box'  
    return EXIT_SUCCESS;  
}  
  
anz = strlen (hptr) ;  
if (anz < MSGLEN-1)  
    *(hptr+anz-1) = '\0';  
sem_post(sptr_letterbox);  
}  
}
```

→ [semaphore\\_client.c](#)

### cron

**mm hh dd MM DD Anweisung**

**mm** minute (0-59)

**hh** hour (0-23)

**dd** day of month (1-31)

**MM** month (1-12)

**DD** day of the week (0-6 mit 0: Sonntag)

Example:

0 8-17 \* \* 1-5 /usr/bin/backup

## Timer functions

```
timer_create(CLOCK_REALTIME, *sigevent, *timer_id);
```

**sigevent:** structure to define timer reaction

**timer\_id:** timer descriptor

```
timer_gettime(timer_id, flags, *itimer_v, *itimer_ov);
```

**timer\_id:** timer descriptor

**flags:** timer flags

**itimer\_v:** Timer structure to define the elapsed time

**itimer\_ov:** Timer structure to define the remaining time

## sigevent structure

For activation via signal:

```
.sigev_notify = SIGEV_SIGNAL;  
.sigev_signo = signo;
```

For activation via thread:

```
.sigev_notify = SIGEV_THREAD;  
.sigev_value.sival_int = thread_id;  
.sigev_notify_attributes = NULL;  
.sigev_notify_function = thread;
```

## itimerspec structure

```
struct itimerspec {  
    struct timespec it_interval; // timer interval  
    struct timespec it_value;   // initial expiration  
};
```

```
struct timespec {  
    time_t tv_sec;        // seconds  
    long tv_nsec;         // nanoseconds  
};
```

## Usage of timers with a signal (I)

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h>
#include <signal.h> #include <time.h> #include <errno.h>
#define FALSE 0
#define TRUE (!FALSE)

volatile sig_atomic_t sflag;

void sigh(int signum)
{
    sflag = TRUE;
}
```

→ [timer\\_via\\_signal.c](#)

## Usage of timers with a signal (II)

```
int main()
{
    int i ;
    timer_t id;
    sigset_t set ;
    struct itimerspec timer;
    struct sigevent event;
    struct sigaction act, oldact;

    sigfillset (&set) ;
    act.sa_flags = 0;
    act.sa_mask = set;
    act.sa_handler = &sigh;
    sigaction(SIGUSR1,&act,&oldact);
    printf ("timer program will start with a delay of 4s...\\n");
    → timer_via_signal.c
```

## Usage of timers with a signal (III)

```
event.sigev_notify = SIGEV_SIGNAL;
event.sigev_signo = SIGUSR1;
if (timer_create(CLOCK_REALTIME,&event,&id) == -1) {
    printf ("Unable to attach timer: %s\n", strerror(errno));
    return EXIT_FAILURE;
}
timer.it_value.tv_sec = 4L;
timer.it_value.tv_nsec = 0L;
timer.it_interval.tv_sec = 1L;
timer.it_interval.tv_nsec = 0L;
timer_settime(id,0,&timer,NULL);
for (i=0; i<10; i++) {
    pause();
    printf ("%s tick\n", (i==0 ? "Now: 10 restarts of the timer
in 1 second intervals:\n" : ""));
}
→ timer_via_signal.c
```

## Usage of timers with a signal (IV)

```
timer.it_value.tv_sec = time(NULL)+20;
timer.it_value.tv_nsec = 0L;
timer.it_interval.tv_sec = 0L;
timer.it_interval.tv_nsec = 0L;
timer_settime(id, TIMER_ABSTIME, &timer, NULL);
sflag = FALSE;
while (!sflag) {
    timer_gettime(id, &timer);
    printf ("Remaining time %2lds\r", timer.it_value.tv_sec);
}
printf ("\n");
timer_delete(id) ;
return EXIT_SUCCESS;
}
```

→ [timer\\_via\\_signal.c](#)

## Usage of timers with a thread (I)

```
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h>
#include <errno.h> #include <time.h> #include <signal.h> #include <pthread.h>
#define FALSE 0
#define TRUE (!FALSE)

volatile int flag=FALSE;

void thread2(union sigval sigval)
{
    flag = TRUE;
}

int main()
{
    int i , evid;
    timer_t id;
    struct itimerspec timer;
    struct sigevent event;
```

[→ timer\\_via\\_thread.c](#)

## Usage of timers with a thread (II)

```
printf ("timer program will start with a delay of 4s...\\n");
event.sigev_notify = SIGEV_THREAD;
event.sigev_value.sival_int = evid;
event.sigev_notify_attributes = NULL;
event.sigev_notify_function = thread2;
if (timer_create(CLOCK_REALTIME, &event, &id) == -1) {
    printf ("Unable to attach timer: %s\\n", strerror(errno));
    return EXIT_FAILURE;
}
timer.it_value.tv_sec = 4L;
timer.it_value.tv_nsec = 0L;
timer.it_interval.tv_sec = 1L;
timer.it_interval.tv_nsec = 0L;
timer_settime(id, 0, &timer, NULL);
```

→ [timer\\_via\\_thread.c](#)

## Usage of timers with a thread (III)

```
for ( i=0; i<10; i++) {  
    while(!flag)  
    ;  
    flag = FALSE;  
    printf ("%s tick\n", (i==0 ? "Now: 10 restarts of the timer  
        in 1 second intervals:\n" : ""));  
}  
  
timer.it_value.tv_sec = time(NULL)+20;  
timer.it_value.tv_nsec = 0L;  
timer.it_interval.tv_sec = 0L;  
timer.it_interval.tv_nsec = 0L;  
timer_settime(id, TIMER_ABSTIME, &timer, NULL);
```

→ [timer\\_via\\_thread.c](#)

## Usage of timers with a thread (VI)

```
while (!flag) {  
    timer_gettime(id, &timer);  
    printf ("Remaining time %2lds\r", timer.it_value.tv_sec);  
}  
printf ("\n");  
timer_delete(id) ;  
return EXIT_SUCCESS;  
}
```

→ [timer\\_via\\_thread.c](#)

# Betriebssysteme / Operating Systems

## awk

SS 2020

Prof. Dr.-Ing. Holger Gräßner

### awk

History of awk: A program to read in files line by line and to split these into single words. Purpose: To do a user specified output.

Name awk: Initials of its developers (year 1977):  
**Aho, Weinberger and Kernighan.**

Today, we have some branches of awk development.

Most popular: GNU-version gawk (GNU awk).

Often, there is a symbolic link: awk → gawk

Syntax of awk is similar to C

Today, awk is nearly a full programming language.

### awk's data flow

Data flow:

Input stream (standard input or some files)

→ awk (processing line by line until file ends or **(Strg)+(D)** )

→ output stream

### Processing a line in awk

- Split line into multiple words.
- The separation character is defined by the variable IFS (is called FS in awk), or blanks or tabs.
- Compare the line with a pattern or a regular expression,
- Execute some commands, if there is a match.

## “Hello World” in awk

```
#! /bin/sh

awk '

BEGIN {
    printf("Hello World\n");
}

'
```

→ [awk-world](#)

### awk program's execution

```
...  
awk '          # awk Call  
BEGIN { ... } # Program BEGIN  
pattern1 { ... } # some pattern matching programs...  
...  
conditions  
commands  
patternN { ... }  
END { ... } # Program END  
' Variables, files  
...
```

## awk list processing

```
#!/bin/sh
awk '

# Demonstration, how awk processes a list ("memberslist"):

BEGIN {
    print "BEGIN block\n"
}

# Print every line with just 1 field during line processing:
NF == 1 {
    print $1
}
                $1: content of first field
}

END {
    print("END block\n")
} ' memberslist
```

→ awklist

## Predefined awk functions:

| Funktion                 | Bedeutung                                                                                           |
|--------------------------|-----------------------------------------------------------------------------------------------------|
| <b>cos(expr)</b>         | Cosinus function                                                                                    |
| <b>exp(expr)</b>         | Exponential function                                                                                |
| <b>getline()</b>         | Read next line; 0: file end, 1: otherwise                                                           |
| <b>index(zk1, zk2)</b>   | Position of substring <b>zk2</b> in <b>zk1</b> ; 0 if not existing                                  |
| <b>int(expr)</b>         | Integer of <b>expr</b>                                                                              |
| <b>length(zk)</b>        | Length of the string <b>zk</b>                                                                      |
| <b>log(expr)</b>         | natural logarithm                                                                                   |
| <b>sin(expr)</b>         | Sinus function                                                                                      |
| <b>split(zk, a, c)</b>   | Split <b>zk</b> into <b>a[1] ... a[n]</b><br>by <b>c</b> (separation character); result is <b>n</b> |
| <b>sprintf(fmt, ...)</b> | Format string; format ist defined by <b>fmt</b>                                                     |
| <b>substr(zk, m, n)</b>  | Get substring of <b>zk</b> : Start at <b>m</b> , count <b>n</b>                                     |

## Associative arrays in awk

Declaration and initialization:

```
glossar["UDP"] = "User Datagram Protocol";  
glossar["IP"] = "Internet Protocol";
```

Access and output:

```
for (item in glossar)  
printf("%s: %s\n", item, glossar[item]);
```

### Predifined Variables in awk

**NF** total number of fields in a record

**NR** Example: Number of Records Variable

**FS** is any single character or regular expression which you want to use as a input field separator

OFS Example: Output Field Separator Variable

...even more awk

<http://www.gnu.org/software/gawk/manual/gawk.html>

## awk example program (I)

```
#!/bin/sh

pwd

ls -l | awk '

BEGIN {
    printf("%-25s\t%10s\n", "FILE", "BYTES")
}

# 2 test for 9 fields ; files begin with "-"

NF == 9 && /^-/ {
    sum += $5 # accumulate size of file
    ++filenum # count number of files
    printf("%-25s\t%10d\n", $9, $5)
}

# 3 test for 9 fields ; directory begins with "d"

NF == 9 && /^d/ {
    print "<dir>", "\t", $9
}
```

/xyz/: line matches xyz.

→ awkpwddemo

## awk example program (II)

```
# 4 test for ls -lR line ./ dir:  
$1 ~ /^.*: $/ {  
    print "\t" $0  
}  
  
# 5 once all is done  
  
END {  
    printf("Total: %d bytes (%d files)\n", sum, filenum)  
}  
'
```

\$1 ~ /xyz/: field 1 matches xyz.

→ awkpwddemo

## awk exercise „sports club statistics (I)

Task: Process the following members list (`memberslist`) of a sports club:

### Indoor sports:

|   |          |           |    |           |
|---|----------|-----------|----|-----------|
| F | Sandra   | Dosenkohl | 30 | Aerobic   |
| F | Lieschen | Meier     | 80 | Gymnastik |

### Endurance sports:

|   |           |         |    |           |
|---|-----------|---------|----|-----------|
| M | Jan       | Frodeno | 38 | Triathlon |
| M | Sebastian | Kienle  | 35 | Triathlon |
| M | Patrick   | Lange   | 33 | Triathlon |

### Ball sports:

|   |       |           |    |          |
|---|-------|-----------|----|----------|
| M | Erwin | Lottemann | 55 | Fussball |
|---|-------|-----------|----|----------|

1) Print first names and family names of all triathletes!

2) Print a statistic:

- number of members in total,
- number of triathletes,
- number of men,
- number of women,
- average age of the men,
- average age of the women.

→ `memberslist`

# Betriebssysteme / Operating Systems

## Concurrency: Deadlock + Starvation

SS 2019

Prof. Dr.-Ing. Holger Gräßner

[18 OS-BS 2019 Deadlock.pptx]

## Concurrency: Deadlock and Starvation

Principles of Deadlock

Deadlock Prevention

Deadlock Avoidance

Deadlock Detection

An Integrated Deadlock Strategy

# Concurrency: Deadlock and Starvation

## Principles of Deadlock

Deadlock Prevention

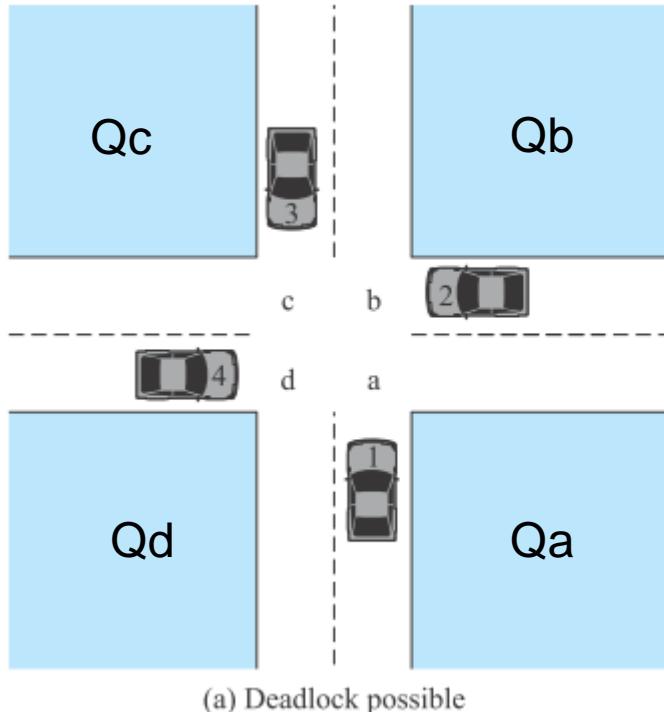
Deadlock Avoidance

Deadlock Detection

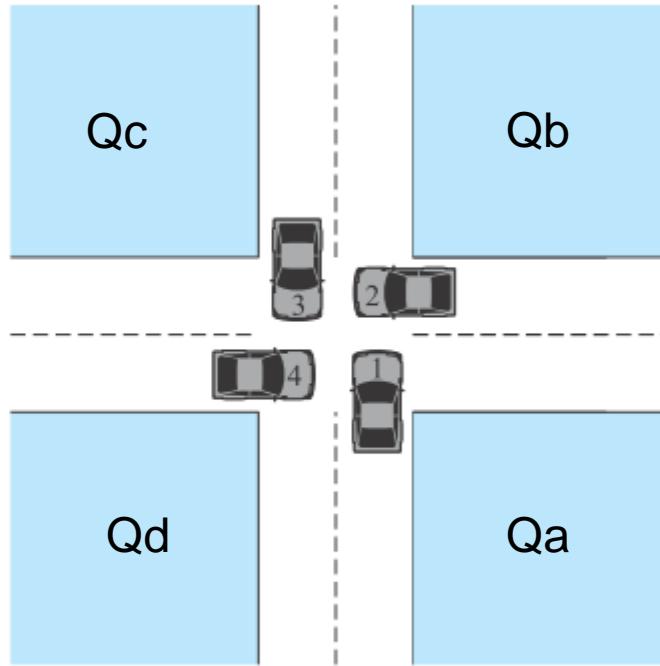
An Integrated Deadlock Strategy

## Principles of Deadlock

Processes = cars (1, 2, 3, 4).  
Ressources = quadrants (Qa, Qb, Qc, Qd).



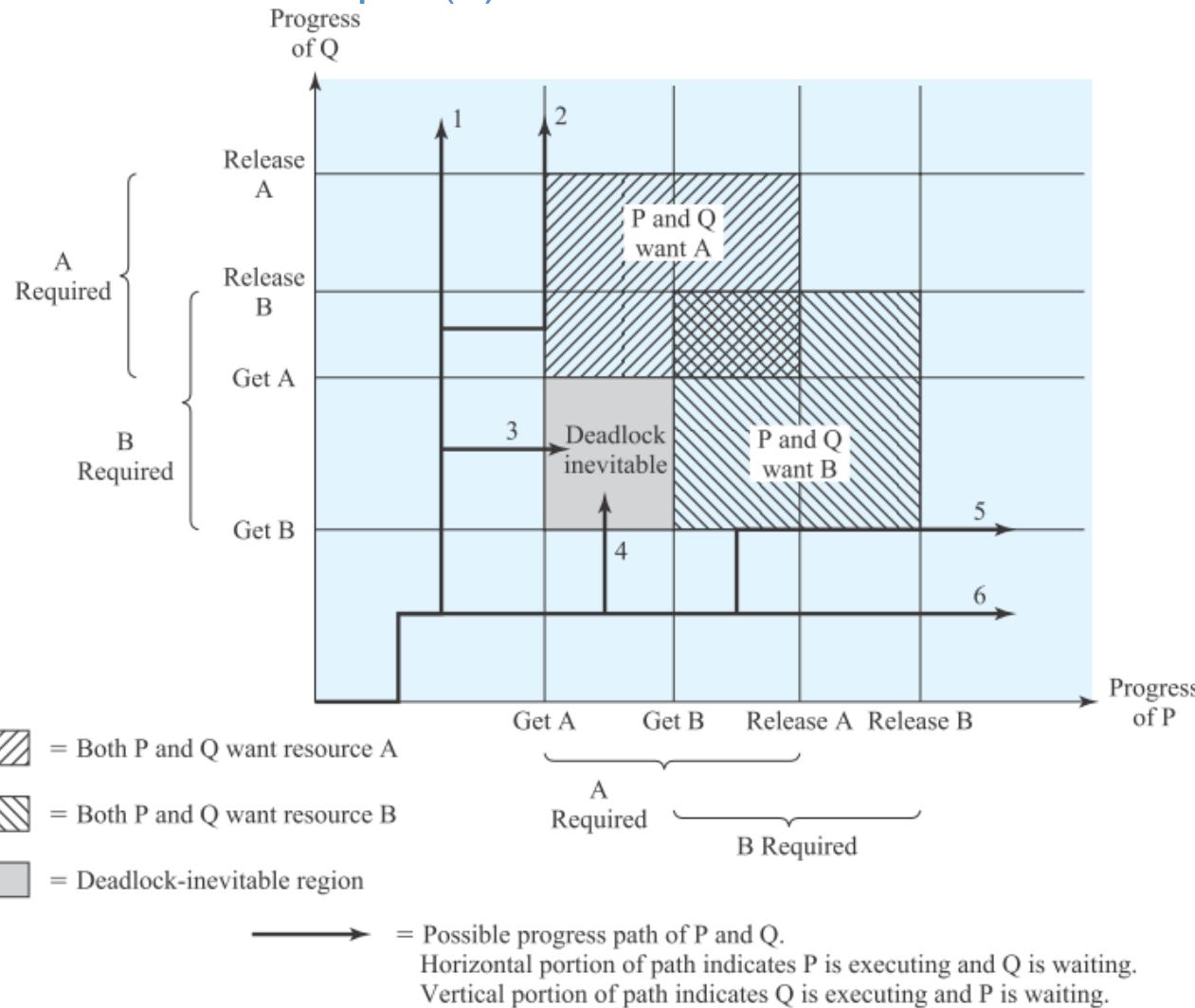
(a) Deadlock possible



(b) Deadlock

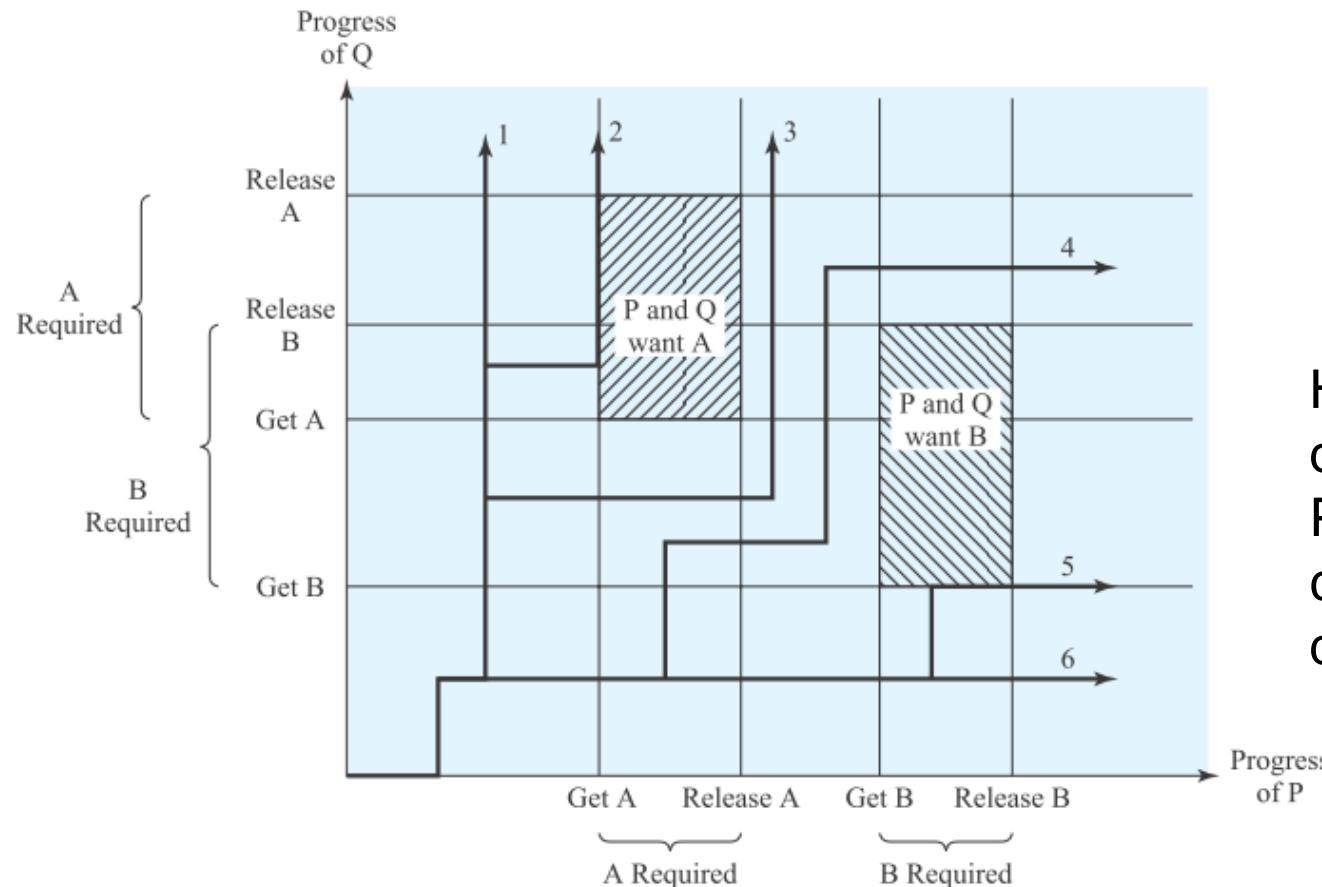
A deadlock is a system state where a set of processes is **permanently blocked** due to their competition of system resources or communication.

## Deadlock: Example (1)



- Paths 1 & 2: OK
- Paths 3 & 4: Deadlock
- Paths 5 & 6: OK

## Deadlock: Example (2)



Here the situation is different:  
Process P is changed and thus no deadlock can occur.

## Reusable Resources

**Reusable** Resources are I/O devices, processors, memory, files, . . . , ie. they are **not consumed** by processes.

Yet another example how deadlocks occur (using the reusable resources disk (D) and tape (T)):

| Process P      |                  | Process Q      |                  |
|----------------|------------------|----------------|------------------|
| Step           | Action           | Step           | Action           |
| p <sub>0</sub> | Request (D)      | q <sub>0</sub> | Request (T)      |
| p <sub>1</sub> | Lock (D)         | q <sub>1</sub> | Lock (T)         |
| p <sub>2</sub> | Request (T)      | q <sub>2</sub> | Request (D)      |
| p <sub>3</sub> | Lock (T)         | q <sub>3</sub> | Lock (D)         |
| p <sub>4</sub> | Perform function | q <sub>4</sub> | Perform function |
| p <sub>5</sub> | Unlock (D)       | q <sub>5</sub> | Unlock (T)       |
| p <sub>6</sub> | Unlock (T)       | q <sub>6</sub> | Unlock (D)       |

Which sequence leads to a deadlock?

Answer: p<sub>0</sub> p<sub>1</sub> q<sub>0</sub> q<sub>1</sub> p<sub>2</sub> q<sub>2</sub>

### Reusable Resources

**Reusable Resources** are I/O devices, processors, memory, files, . . . , ie. they are not **consumed** by processes.

Yet another example how deadlocks occur due to memory.

Assume that 200 kBytes are available.

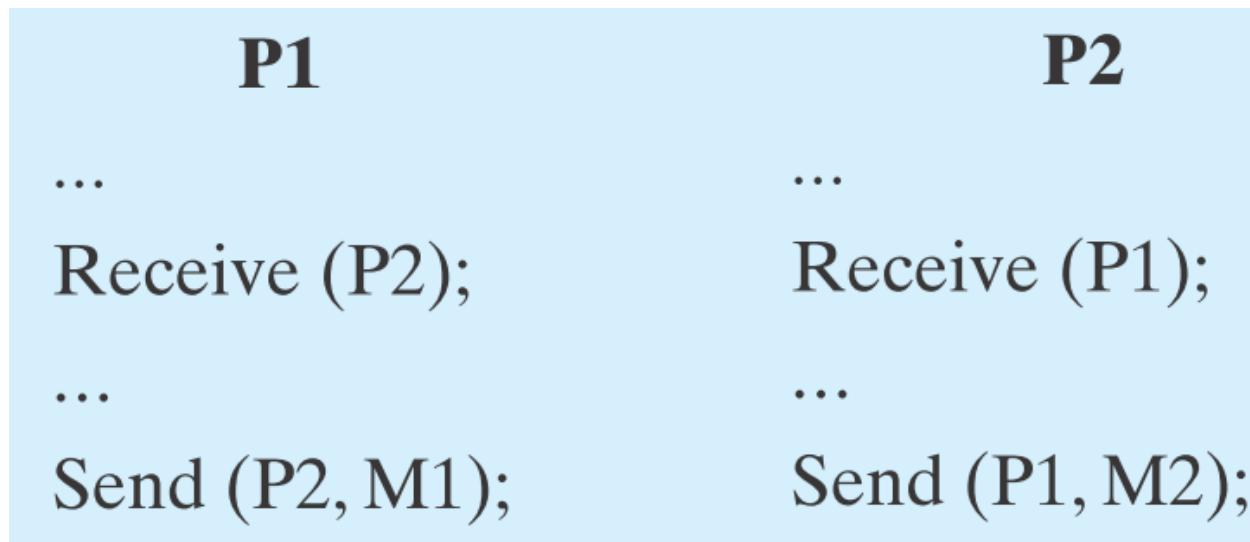
| P1                 | P2                 |
|--------------------|--------------------|
| ...                | ...                |
| Request 80 Kbytes; | Request 70 Kbytes; |
| ...                | ...                |
| Request 60 Kbytes; | Request 80 Kbytes; |

Explain why this leads to a deadlock.

### Consumable Resources

Examples of **consumable resources** are interrupts, signals, messages, and information in I/O buffers.

An example how deadlocks occur due to messages:



A deadlock occurs if the receive is blocking.

So, how to get rid of deadlocks?

## Three Approaches against Deadlocks (1)

| Approach   | Resource Allocation Policy                      | Different Schemes                         | Major Advantage                                                                                                                                                                   | Major Disadvantage                                                                                                                                                    |
|------------|-------------------------------------------------|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Prevention | Conservative:<br>Undercommits resources         | Requesting all resources at once          | <ul style="list-style-type: none"> <li>• Works well for processes that perform a single burst of activity</li> <li>• No preemption necessary</li> </ul>                           | <ul style="list-style-type: none"> <li>• Inefficient</li> <li>• Delays process initiation</li> <li>• Future resource requirements must be known in advance</li> </ul> |
|            |                                                 | Preemption                                | <ul style="list-style-type: none"> <li>• Convenient when applied to resources whose state can be saved and restored easily</li> </ul>                                             | <ul style="list-style-type: none"> <li>• Preempts more often than necessary</li> </ul>                                                                                |
|            |                                                 | Resource ordering                         | <ul style="list-style-type: none"> <li>• Feasible to enforce via compile-time checks</li> <li>• Needs no run-time computation since problem is solved in system design</li> </ul> | <ul style="list-style-type: none"> <li>• Preempts more often than necessary</li> </ul>                                                                                |
| Avoidance  | Midway between that of detection and prevention | Manipulate to find at least one safe path | <ul style="list-style-type: none"> <li>• No preemption necessary</li> </ul>                                                                                                       | <ul style="list-style-type: none"> <li>• Future resource requirements must be known by OS</li> <li>• Processes can be blocked for long periods</li> </ul>             |

## Three Approaches against Deadlocks (2)

| Approach  | Resource Allocation Policy                                   | Different Schemes                        | Major Advantage                                                                                                     | Major Disadvantage                                                         |
|-----------|--------------------------------------------------------------|------------------------------------------|---------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Detection | Very liberal; requested resources are granted where Possible | Invoke periodically to test for deadlock | <ul style="list-style-type: none"><li>Never delays process initiation</li><li>Facilitates online handling</li></ul> | <ul style="list-style-type: none"><li>Inherent preemption losses</li></ul> |

## Resource Allocation Graphs

In the following we will use **resource allocation graphs** to characterise the allocation of resources to processes.

### Definition (Resource Allocation Graph)

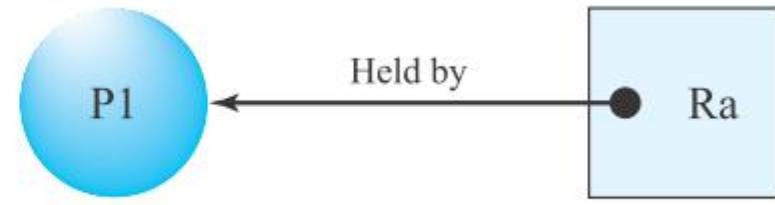
This is a directed graph where both resources and processes are nodes. Within a resource node, a dot is shown for each instance of that resource.

- An edge from a process to a resource indicates a resource that has been requested by the process (but not yet granted).
- An edge from a reusable resource node dot to a process indicates a request that has been granted.
- An edge from a consumable resource node dot to a process indicates that the process is the producer of that resource.

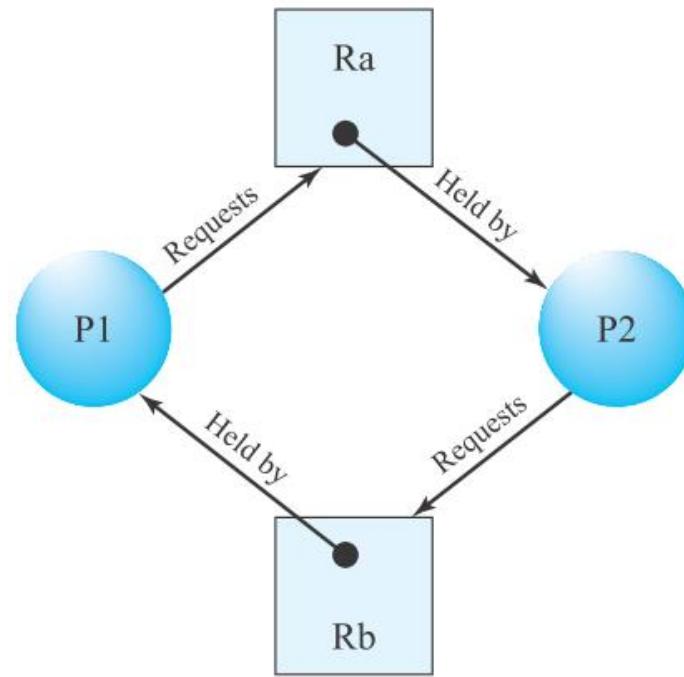
## Resource Allocation Graphs: Examples (1)



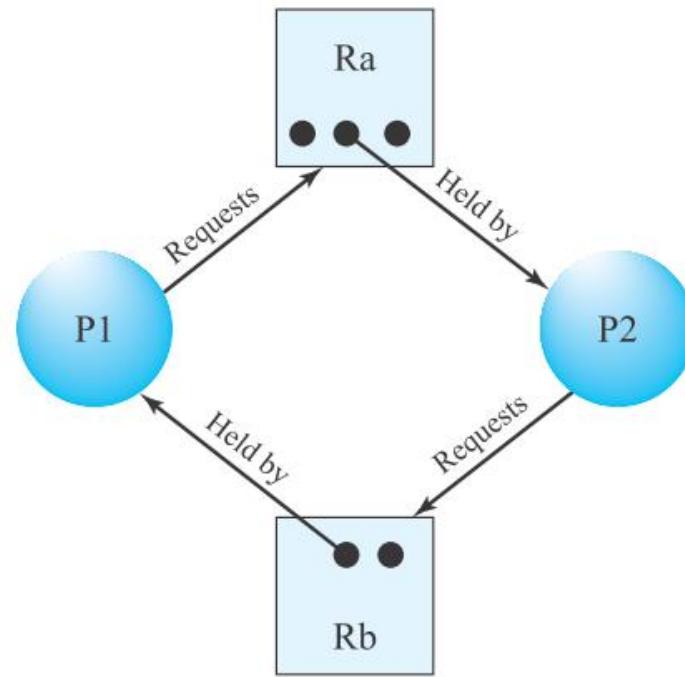
(a) Resource is requested



(b) Resource is held

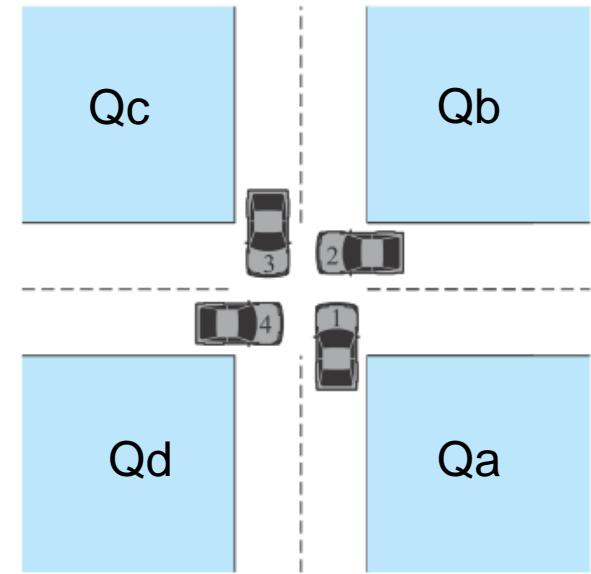
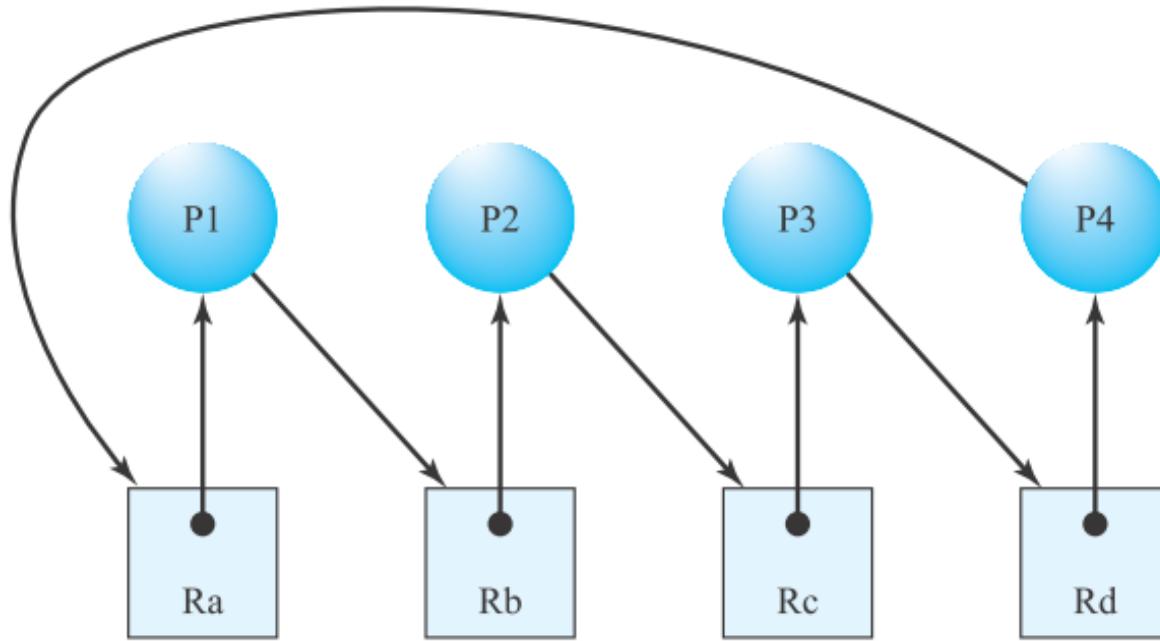


(c) Circular wait



(d) No deadlock

## Resource Allocation Graphs: Examples (2)



## The Conditions for Deadlock

There are three necessary conditions for a deadlock:

- **Mutual exclusion:** Only one process may use a resource at a time.  
No process may access a resource unit that has been allocated to another process.
- **Hold and Wait:** A process may hold allocated resources while awaiting assignment of other resources.
- **No preemption:** No resource can be forcibly removed from a process holding it.

Together all of them are necessary, but not sufficient.

Note that they are all desirable.

- **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

| Possibility of Deadlock                                                                                               | Existence of Deadlock                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"><li>1. Mutual exclusion</li><li>2. No preemption</li><li>3. Hold and wait</li></ol> | <ol style="list-style-type: none"><li>1. Mutual exclusion</li><li>2. No preemption</li><li>3. Hold and wait</li><li>4. Circular wait</li></ol> |

## Concurrency: Deadlock and Starvation

Principles of Deadlock

Deadlock Prevention

Deadlock Avoidance

Deadlock Detection

An Integrated Deadlock Strategy

### Deadlock Prevention

The strategy of **deadlock prevention** is to design a system in such a way that the possibility of deadlock is excluded, ie. to pick one of the four conditions of deadlock and to prevent its occurrence.

Let's go through the four options. . .

### Deadlock Prevention: No Mutual Exclusion

This is hardly to achieve. There are various situations where processes need mutual exclusive access to a resource.

⇒ not an option.

### Deadlock Prevention: No Hold and Wait

The hold-and-wait condition can be prevented by requiring that a process requests all of its required resources **at one time** and blocking the process until all requests can be granted simultaneously.

#### MINUS

- A process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources.
- Resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.
- A process may not know in advance all of the resources that it will require.

### Deadlock Prevention: Preemption

Several options for preemption:

- If a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.
- If a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources. (Needs unique priorities).

#### MINUS

- Only applicable to resources whose state can be easily saved and restored later, for instance the processor.

## Deadlock Prevention: No Circular Wait

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering. In this way, no circular-wait is possible: Assume A has a resource  $R_i$  and waits for  $R_j$ , while B has  $R_j$  and waits for  $R_i$ . With the above scheme we get  $i < j$  and  $j < i$ , which is a contradiction.

### MINUS

- see hold-and-wait prevention

## Concurrency: Deadlock and Starvation

Principles of Deadlock

Deadlock Prevention

**Deadlock Avoidance**

Deadlock Detection

An Integrated Deadlock Strategy

### Deadlock Avoidance

Deadlock **prevention** tackles the problem of deadlocks in a static way. This leads to inefficient use of resources and inefficient execution of processes.

In case of deadlock **avoidance** allows the first three conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. The decision is made **dynamically** whether the current resource allocation request will, if granted, potentially lead to a deadlock. Thus, this technique requires knowledge of future process resource requests.

Two approaches:

- Do not start a process if its demands might lead to deadlock.  
**Process initiation denial**
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.  
**Resource allocation denial**

## Process Initiation Denial (I)

Consider a system of  $n$  processes and  $m$  different types of resources.

|                                                                                                                                                                                                         |                                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|
| [Resource = $\mathbf{R} = (R_1, R_2, \dots, R_m)$                                                                                                                                                       | total amount of each resource in the system                  |
| Available = $\mathbf{V} = (V_1, V_2, \dots, V_m)$                                                                                                                                                       | total amount of each resource not allocated to any process   |
| Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$      | $C_{ij}$ = requirement of process $i$ for resource $j$       |
| Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix}$ | $A_{ij}$ = current allocation to process $i$ of resource $j$ |

Note that  $\mathbf{C}$  describes the maximum requirement of each process for each resource.

## Process Initiation Denial (II)

Then holds:

$$R_j = V_j + \sum_{i=1}^n A_{ij} \quad \text{for all } j$$

$$C_{ij} \leq R_j \quad \text{for all } i, j$$

$$A_{ij} \leq C_{ij} \quad \text{for all } i, j$$

Process initiation denial policy: Start a new process  $P_{n+1}$  only if

$$R_j \geq C_{(n+1),j} + \sum_{i=1}^n C_{ij} \quad \text{for all } j$$

That is, a process is only started if the maximum claim of all current processes plus those of the new process can be met.

**This is far too pessimistic!**

## Resource Allocation Denial (I)

This approach is based on Dijkstra's banker's algorithm (1965).

### Definition

We call the vectors R, V and matrices C, A above a **state**.

Moreover, we say that a state is **safe** if there is at least one sequence of resource allocations to processes that does not result in a deadlock.

Otherwise, we call it **unsafe**.

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 6  | 1  | 2  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 0  | 0  | 1  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

## Resource Allocation Denial (II)

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 6  | 1  | 2  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 0  | 0  | 1  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

Intermediate question: Can any of the four processes be run to completion with the resources available?

That means, is there an i such that  $C_{ij} - A_{ij} \leq V_j$  for all j?

This is given, but only of i = 2.

Assume now that process 2 runs to completion. We get. . .

## Resource Allocation Denial (III)

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 0  | 0  | 0  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 0  | 0  | 0  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 6  | 2  | 3  |

Available vector V

Intermediate question: Can any of the four processes be run to completion with the resources available?

That means, is there an i such that  $C_{ij} - A_{ij} \leq V_j$  for all j?

This is given for all i.

Assume now that process 1 runs to completion. We get. . .

## Resource Allocation Denial (IV)

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 7  | 2  | 3  |

Available vector V

Intermediate question: Can any of the four processes be run to completion with the resources available?

That means, is there an i such that  $C_{ij} - A_{ij} \leq V_j$  for all  $j$ ?

Now complete process 3 . . .

## Resource Allocation Denial (V)

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  |
| P4 | 4  | 2  | 0  |

C – A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 4  |

Available vector V

and finally process 4 and we are done.

## Resource Allocation Denial (VI)

### Strategy

When a process makes a request for a set of resources, assume that the request is granted, update the system state accordingly, and determine if the result is a safe state. If so, grant the request and, if not, block the process until it is safe to grant the request.

## Resource Allocation Denial (VII)

Current state:

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 5  | 1  | 1  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 1  | 0  | 2  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 1  | 1  | 2  |

Available vector V

P2 request one additional unit of R1 and of R3.

Can we grant this request?

Yes, because the result is safe:

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 6  | 1  | 2  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 0  | 0  | 1  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

## Resource Allocation Denial (VIII)

Current state:

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 5  | 1  | 1  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 1  | 0  | 2  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 1  | 1  | 2  |

Available vector V

P1 requests one additional R1 and R3.

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 0  | 1  |
| P2 | 5  | 1  | 1  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 2  | 1  |
| P2 | 1  | 0  | 2  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

Can we grant  
this request?No, because the  
result is unsafe:

## Resource Allocation Denial (IX)

Note that this state is not a deadlock:

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 0  | 1  |
| P2 | 5  | 1  | 1  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 2  | 1  |
| P2 | 1  | 0  | 2  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C – A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

It is just unsafe!

## Resource Allocation Denial (X)

Deadlock avoidance:

### PLUS

- No need for preemption or rollback
- less restrictive than deadlock prevention

### MINUS

- Maximum resource requirement for each process must be stated in advance.
- The processes under consideration must be independent; that is, the order in which they execute must be unconstrained by any synchronisation requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.

# Concurrency: Deadlock and Starvation

Principles of Deadlock

Deadlock Prevention

Deadlock Avoidance

**Deadlock Detection**

An Integrated Deadlock Strategy

### Deadlock Detection

**Deadlock detection** strategies do not limit resource access or restrict process actions. Requested resources are granted to processes whenever possible. Periodically, the OS performs an algorithm that allows to detect the circular wait condition.

## Deadlock Detection

|    | R1 | R2 | R3 | R4 | R5 |
|----|----|----|----|----|----|
| P1 | 0  | 1  | 0  | 0  | 1  |
| P2 | 0  | 0  | 1  | 0  | 1  |
| P3 | 0  | 0  | 0  | 0  | 1  |
| P4 | 1  | 0  | 1  | 0  | 1  |

Request matrix Q

|    | R1 | R2 | R3 | R4 | R5 |
|----|----|----|----|----|----|
| P1 | 1  | 0  | 1  | 1  | 0  |
| P2 | 1  | 1  | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  | 1  | 0  |
| P4 | 0  | 0  | 0  | 0  | 0  |

Allocation matrix A

|  | R1 | R2 | R3 | R4 | R5 |
|--|----|----|----|----|----|
|  | 2  | 1  | 1  | 2  | 1  |

Resource vector

|  | R1 | R2 | R3 | R4 | R5 |
|--|----|----|----|----|----|
|  | 0  | 0  | 0  | 0  | 1  |

Available vector

The request matrix Q has the following meaning:

$Q_{ij}$  represents the amount of resources of type j requested by process i  
(in addition to the resources the process already has)

⇒ Do not mismatch:

- Matrix C in previous section (deadlock avoidance):  
**Maximum requirement** of resources
- Matrix Q here:  
**Additional resources** requested in excess to those already allocated.

## Deadlock Detection Algorithm

The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:

1. Mark each process that has a row in the A matrix of all zeros.
2. Initialise a temporary vector W to equal the available vector V.
3. Find an index i such that process i is currently unmarked and the  $i^{\text{th}}$  row of Q is less than or equal to W . That is,  $Q_{ik} \leq W_k$  for  $1 \leq k \leq m$ . If no such row is found, terminate.
4. If such a row is found, mark process i and add the corresponding row of A to W . That is, set  $W_k := W_k + A_{ik}$  for  $1 \leq k \leq m$ . Goto 3.

A deadlock exists if (and only if) there are unmarked processes at the end of the algorithm. Each unmarked process is deadlocked.

## Deadlock Detection: Example (I)

|    | R1 | R2 | R3 | R4 | R5 |
|----|----|----|----|----|----|
| P1 | 0  | 1  | 0  | 0  | 1  |
| P2 | 0  | 0  | 1  | 0  | 1  |
| P3 | 0  | 0  | 0  | 0  | 1  |
| P4 | 1  | 0  | 1  | 0  | 1  |

Request matrix Q

|    | R1 | R2 | R3 | R4 | R5 |
|----|----|----|----|----|----|
| P1 | 1  | 0  | 1  | 1  | 0  |
| P2 | 1  | 1  | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  | 1  | 0  |
| P4 | 0  | 0  | 0  | 0  | 0  |

Allocation matrix A

|  | R1 | R2 | R3 | R4 | R5 |
|--|----|----|----|----|----|
|  | 2  | 1  | 1  | 2  | 1  |

Resource vector

|  | R1 | R2 | R3 | R4 | R5 |
|--|----|----|----|----|----|
|  | 0  | 0  | 0  | 0  | 1  |

Available vector

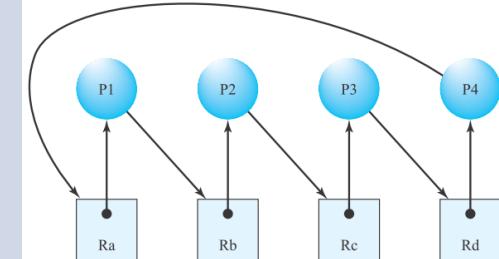
1. Mark P4
2.  $W := (0, 0, 0, 0, 1)$
3. Look at Q: Row 3 is less or equal than W
4. Mark P3.  $W := W + (0, 0, 0, 1, 0) = (0, 0, 0, 1, 1)$
5. Look at Q: No remaining (unmarked) row is  $\leq W$ . Terminate.

P1 and P2 are unmarked and hence deadlocked.

## Deadlock Detection: Example (II)

|   | Content                                                  | Mark                                             | Q                                                                                                                | A                                                                                                                | V           | W(t)                        | Comment                                 |
|---|----------------------------------------------------------|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|-------------|-----------------------------|-----------------------------------------|
| 0 | Init                                                     | $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ | [0 0 0 0 1] |                             |                                         |
| 1 | Mark I, if $A_{ij} = 0$                                  | $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ | [0 0 0 0 1] |                             | Row 4 of A is all zero                  |
| 2 | Initialize W(t)                                          |                                                  |                                                                                                                  |                                                                                                                  |             | $W(0) = V$<br>= [0 0 0 0 1] |                                         |
| 3 | Find an index i such<br>... unmarked<br>and $Q_i \leq W$ | $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ |             | $[0 0 0 0 1]$               | Row 3 matches the condition             |
| 4 | Mark the row,<br>$W := W + A_i$                          | $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ |             | [0 0 0 1 1]                 | Goto 3                                  |
| 3 | Find an index i such<br>... unmarked<br>and $Q_i \leq W$ | $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ |             | [0 0 0 1 1]                 | No such row,<br>terminate               |
| 4 | Rows 1 and 2 are<br>unmarked                             |                                                  |                                                                                                                  |                                                                                                                  |             |                             | So processes 1 and 2<br>are deadlocked. |

## Deadlock Detection: Example street crossing

|   | Content                                                | Mark                                             | Q                                                                                                | A                                                                                                | V                 | W(t)                                                                                | Comment                                       |
|---|--------------------------------------------------------|--------------------------------------------------|--------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|-------------------|-------------------------------------------------------------------------------------|-----------------------------------------------|
| 0 | Init                                                   | $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ | $[0 \ 0 \ 0 \ 0]$ |  |                                               |
| 1 | Mark i, if $A_{ij} = 0$                                |                                                  |                                                                                                  |                                                                                                  |                   |                                                                                     | No such row                                   |
| 2 | Initialize W(t)                                        |                                                  |                                                                                                  |                                                                                                  |                   | $W(0) = V$<br>$= [0 \ 0 \ 0 \ 0]$                                                   |                                               |
| 3 | Find an index i such<br>... unmarked and<br>$Q \leq W$ | $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ | $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ | $[0 \ 0 \ 0 \ 0]$ | $[0 \ 0 \ 0 \ 0]$                                                                   | No such row., so<br>terminate                 |
| 4 | Lines 1, 2, 3 and 4<br>are unmarked                    |                                                  |                                                                                                  |                                                                                                  |                   |                                                                                     | So processes 1, 2, 3<br>and 4 are deadlocked. |

### Recovery

Once a deadlock is detected, some strategy is needed for recovery.

There are several approaches:

1. Abort all deadlocked processes. This is one of the most common adopted in OS's.
2. Back up each deadlock process to some previously defined checkpoint, and restart. Requires rollback and restart mechanisms be built in to the system. The deadlock may recur.
3. Successively abort deadlocked processes until deadlock no longer exists. The order should be on the basis of some criterion of minimum cost.
4. Successively preempt resources until deadlock no longer exists. Requires rollback and restart mechanisms be built in to the system.

## Concurrency: Deadlock and Starvation

Principles of Deadlock

Deadlock Prevention

Deadlock Avoidance

Deadlock Detection

An Integrated Deadlock Strategy

### An Integrated Deadlock Strategy (I)

All three approaches have strengths and weaknesses. In practise, an integrated deadlock strategy is needed, that uses a mixture of all approaches above.

One solution could be the following strategy:

1. Group resources into a number of different classes.
2. Use the linear ordering strategy defined above for the prevention of circular wait to prevent deadlocks between classes.
3. Within a class, use the algorithm that is most appropriate for that class.

## An Integrated Deadlock Strategy (II)

Example for classes (already in a linear order):

1. Swappable space: Blocks on secondary storage for swapping.
2. Process resources: Assignable devices (disks, file, . . . ).
3. Main memory: Assignable to processes in pages or segments.
4. Internal resources: Such as I/O channels.

## An Integrated Deadlock Strategy (III)

Within the class use the following deadlock strategy:

1. Swappable space:

Could be: Deadlock prevention by no hold-and-wait, ie. a process has to allocate it completely in one step.

2. Process resources:

Could be: Deadlock avoidance.

3. Main memory:

Should be: Deadlock prevention by preemption.

4. Internal resources:

Could be: Prevention by resource ordering.

# **Betriebssysteme / Operating Systems**

## **Windows I:**

**Shells, threads, processes, files, network,  
interprocess-communication**

SS 2020

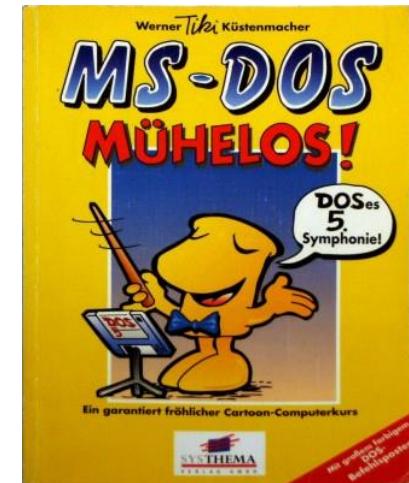
Prof. Dr.-Ing. Holger Gräßner

## History: DOS prompt and batches

Commandline interpreter **Command.COM**

```
set outdir=C:\temp
set srccdir=%~dp1
echo copy text files from %srccdir% to %outdir% ...
PAUSE
if exist %outdir% goto LOS
echo target directory %outdir% does not exist!
PAUSE
goto ENDE
:LOS
copy %srccdir%*.txt %outdir%
echo Done!!!
PAUSE
:ENDE
```

→ [CopyTxt.bat](#)



## Command prompt CMD . EXE

- all programs → Windows system → command prompt
- similar to DOS
- it's a Win32 program, no DOS!
- some additional functions, e. g. pipes



The screenshot shows a Windows Command Prompt window with the title bar "C:\Windows\system32\cmd.exe". The command history and output are as follows:

```
C:\temp>mkdir sub
C:\temp>copy c:\temp\*.txt sub
c:\temp\kannweg.txt
c:\temp\Log.txt
      2 Datei(en) kopiert.

C:\temp>cd sub
C:\temp\sub>dir
  Volume in Laufwerk C: hat keine Bezeichnung.
  Volumeseriennummer: 1669-52F8

  Verzeichnis von C:\temp\sub

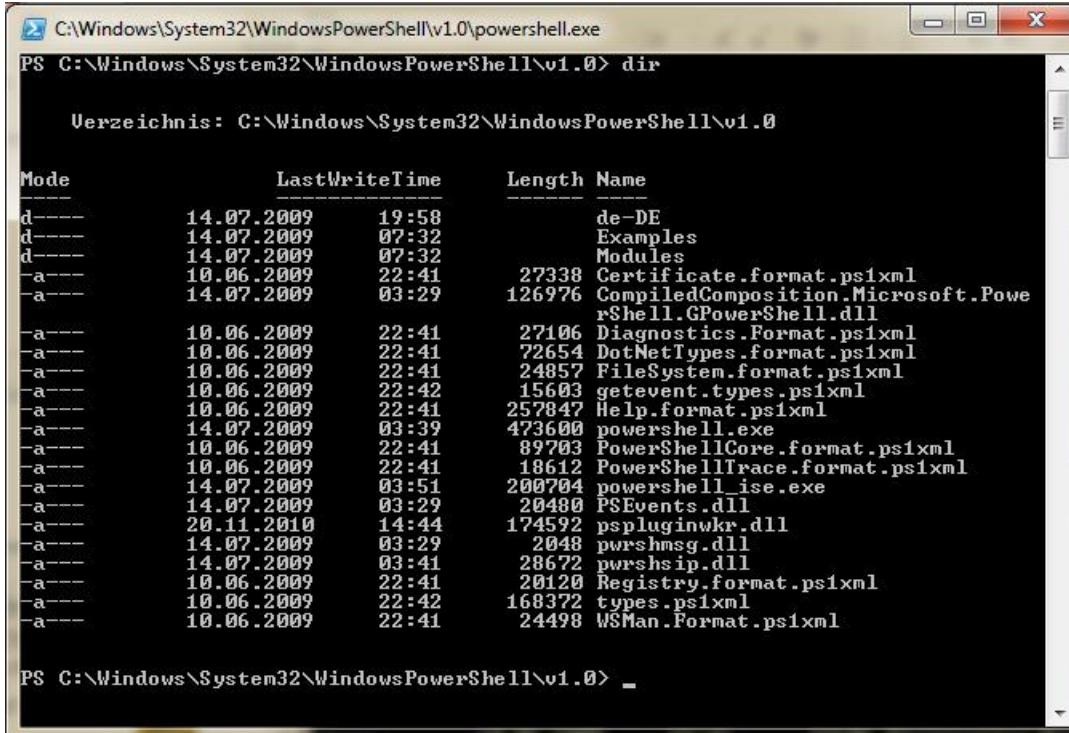
07.11.2015  14:03    <DIR>
07.11.2015  14:03    <DIR>
07.10.2015  16:00            41.269 kannweg.txt
07.10.2015  16:00            41.269 Log.txt
                  2 Datei(en),   82.538 Bytes
                  2 Verzeichnis(se), 602.535.976.960 Bytes frei

C:\temp\sub>cd..
C:\temp>
```

## Windows Power Shell

- It's usually included since Windows 7
- „PowerShell.exe“
- Based on .NET framework
- pipes
- filter
- Object oriented concepts
- scripting via

*Power Shell Scripting Language*



```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
PS C:\Windows\System32\WindowsPowerShell\v1.0> dir

Verzeichnis: C:\Windows\System32\WindowsPowerShell\v1.0

Mode                LastWriteTime       Length Name
d---          14.07.2009     19:58              de-DE
d---          14.07.2009     07:32              Examples
d---          14.07.2009     07:32              Modules
-a--          10.06.2009     22:41      27338 Certificate.format.ps1xml
-a--          14.07.2009     03:29      126976 CompiledComposition.Microsoft.PowerShell.GPowerShell.dll
-a--          10.06.2009     22:41      27106 Diagnostics.Format.ps1xml
-a--          10.06.2009     22:41      72654 DotNetTypes.Format.ps1xml
-a--          10.06.2009     22:41      24857 FileSystem.Format.ps1xml
-a--          10.06.2009     22:42      15683 getevent.types.ps1xml
-a--          10.06.2009     22:41      257847 Help.Format.ps1xml
-a--          14.07.2009     03:39      473600 powershell.exe
-a--          10.06.2009     22:41      89703 PowerShellCore.Format.ps1xml
-a--          10.06.2009     22:41      18612 PowerShellTrace.Format.ps1xml
-a--          14.07.2009     03:51      200704 powershell_ise.exe
-a--          14.07.2009     03:29      20480 PSEvents.dll
-a--          20.11.2010    14:44      174592 pspluginwkr.dll
-a--          14.07.2009     03:29      2048 pwrshmsg.dll
-a--          14.07.2009     03:41      28672 pwrshsip.dll
-a--          10.06.2009     22:41      20120 Registry.Format.ps1xml
-a--          10.06.2009     22:42      168372 types.ps1xml
-a--          10.06.2009     22:41      24498 WSMan.Format.ps1xml

PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

### Windows Scripting Host

It's usually included since Windows 98SE + Windows 2000

Purpose: Do periodic tasks automatically

WSH uses script engines.

Included (default):

- JScript
- VBScript

Script engines are to be included by other programs, e. g.

- Internet Explorer
- Office
- FixFoto

## CreateProcess()

```
BOOL WINAPI CreateProcess( _In_opt_ LPCTSTR lpApplicationName,
_Inout_opt_ LPTSTR lpCommandLine, _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
_In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes, _In_ BOOL bInheritHandles,
_In_ DWORD dwCreationFlags, _In_opt_ LPVOID lpEnvironment,
_In_opt_ LPCTSTR lpCurrentDirectory, _In_ LPSTARTUPINFO lpStartupInfo,
_Out_ LPPROCESS_INFORMATION lpProcessInformation );
```

Some parameters:

- **lpApplicationName** [in, optional]: program to be executed, e. g. “C:\temp\test.exe”
- **lpCommandLine** [in, out, optional]: command line, may contain the program’s name (if **lpApplicationName**=NULL)
- **lpProcessAttributes** [in, optional]: pointer to [SECURITY\\_ATTRIBUTES](#) structure to organize inheritance. NULL: no inheritance of handles to new child processes.
- **lpThreadAttributes** [in, optional]: ditto, inheritance of handles to new child threads.
- **bInheritHandles** [in]: TRUE: inheritance of all handles to the new process.
- **dwCreationFlags** [in]: priority flags, e. g. `NORMAL_PRIORITY_CLASS`, `IDLE_PRIORITY_CLASS`
- **lpProcessInformation** [out]: pointer to [PROCESS\\_INFORMATION](#), to receive information about the new process. Handles have to be closed by [CloseHandle\(\)](#).

## Process creation using CreateProcess() (1)

```
// Create a process under Windows OS

// source: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682512\(v=vs.85\).aspx
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;                      // Window information and handles to be inherited
   // by child processes
    PROCESS_INFORMATION pi;               // handles and IDs of the new child process
   // (or thread) to create

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
}
```

→ DemoCreateThreadProcess → CreateProcess.cpp

### Process creation using CreateProcess() (2)

```
// Filename of child programm (child.exe) is the first parameter of the
// command line.
// Usage (if named Demo.exe): „Demo C:\temp\child.exe“.

if( argc != 2 )
{
    printf("Usage: %s [cmdline] \n", argv[0]);
    return;
}
```

→ DemoCreateThreadProcess → CreateProcess.cpp

## Process creation using CreateProcess() (3)

```
// Start the child process:  
  
if( !CreateProcess( NULL, // usually: name of the module to start.  
                    // NULL: filename will be the first argument...  
                    argv[1], // ... of this command line  
                    NULL, // Process handle not inheritable  
                    NULL, // Thread handle not inheritable  
                    FALSE, // Set handle inheritance to FALSE  
                    0, // No creation flags  
                    NULL, // Use parent's environment block  
                    NULL, // Use parent's starting directory  
                    &si, // Pointer to STARTUPINFO structure  
                    &pi ) // Pointer to PROCESS_INFORMATION structure  
)  
{  
    printf( "CreateProcess failed (%d). \n", GetLastError() );  
    return;  
}
```

→ DemoCreateThreadProcess → CreateProcess.cpp

## Process creation using CreateProcess() (4)

```
// Child process is now running.  
  
// Wait until child process exits.  
WaitForSingleObject( pi.hProcess, INFINITE );  
  
// Close process and thread handles.  
CloseHandle( pi.hProcess );  
CloseHandle( pi.hThread );  
}
```

→ DemoCreateThreadProcess → CreateProcess.cpp

## Thread creation using CreateThread()

source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms682453\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682453(v=vs.85).aspx)

```
HANDLE WINAPI CreateThread(_In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
_In_ SIZE_T dwStackSize, _In_ LPTHREAD_START_ROUTINE lpStartAddress,
_In_opt_ LPVOID lpParameter, _In_ DWORD dwCreationFlags,
_Out_opt_ LPDWORD lpThreadId );
```

parameters:

- **lpThreadAttributes [in, optional]**: pointer to a [SECURITY\\_ATTRIBUTES](#) structure to organize inheritance.  
**NULL**: no inheritance of handles to child thread.  
Structure holds **lpSecurityDescriptor** for the new thread.
- **dwStackSize [in]**: stack's start size in bytes, round up to page address.  
**NULL**: default size.
- **lpStartAddress [in]**: Pointer to thread function.
- **lpParameter [in, optional]**: Pointer to thread arguments.
- **dwCreationFlags [in]**: Defines the new thread's startup behaviour.. **NULL**: immediately.  
**CREATE\_SUSPENDED**: thread will start in state "suspend" until **ResumeThread()** is called.
- **lpThreadId [out, optional]**: Pointer to thread ID variable.

## Thread creation using CreateThread() (1)

```
// Generate a thread using CreateThread() under Windows OS

// source: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682516\(v=vs.85\).aspx

#include <windows.h>
#include <tchar.h>
#include <strsafe.h>

#define MAX_THREADS 3
#define BUF_SIZE 255

DWORD WINAPI MyThreadFunction( LPVOID lpParam );
void ErrorHandler(LPTSTR lpszFunction);
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (2)

```
// Sample custom data structure for threads to use.  
// This is passed by void pointer so it can be any data type  
// that can be passed using a single void pointer (LPVOID).  
  
typedef struct MyData {  
    int val1;  
    int val2;  
} MYDATA, *PMYDATA;  
  
int _tmain()  
{  
    PMYDATA pDataArray[MAX_THREADS];  
    DWORD dwThreadIdArray[MAX_THREADS];  
    HANDLE hThreadArray[MAX_THREADS];
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (3)

```
// Create MAX_THREADS worker threads.

for( int i=0; i<MAX_THREADS; i++ )
{
    // Allocate memory for thread data.

    pDataArray[i] = (PMYDATA) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
   sizeof(MYDATA));

    if( pDataArray[i] == NULL )
    {
        // If the array allocation fails, the system is out of memory
        // so there is no point in trying to print an error message.
        // Just terminate execution.

        ExitProcess(2);
    }
}
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (4)

```
// Generate unique data for each thread to work with.

pDataArray[i]->val1 = i;
pDataArray[i]->val2 = i+100;

// Create the thread to begin execution on its own.

hThreadArray[i] = CreateThread(
    NULL,                      // default security attributes
    0,                         // use default stack size
    MyThreadFunction,          // thread function name
    pDataArray[i],             // argument to thread function
    0,                         // use default creation flags
    &dwThreadIdArray[i]);     // returns the thread identifier
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (5)

```
// Check the return value for success.  
// If CreateThread fails, terminate execution.  
// This will automatically clean up threads and memory.  
  
if (hThreadArray[i] == NULL)  
{  
    ErrorHandler(TEXT("CreateThread"));  
    ExitProcess(3);  
}  
} // End of main thread creation loop.  
  
// Wait until all threads have terminated.  
  
WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (6)

```
// Close all thread handles and free memory allocations.

for(int i=0; i<MAX_THREADS; i++)
{
    CloseHandle(hThreadArray[i]);
    if(pdataArray[i] != NULL)
    {
        HeapFree (GetProcessHeap () , 0, pDataArray[i]);
        pDataArray[i] = NULL;      // Ensure address is not reused.
    }
}

return 0;
}
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (7)

```
DWORD WINAPI MyThreadFunction( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pDataArray;

    TCHAR msgBuf [BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    // Make sure there is a console to receive output results.

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (8)

```
// Cast the parameter to the correct data type.  
// The pointer is known to be valid because  
// it was checked for NULL before the thread was created.  
  
pDataArray = (PMYDATA) lpParam;  
  
// Print the parameter values using thread-safe functions.  
  
StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n") ,  
    pDataArray->val1, pDataArray->val2);  
StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);  
WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize, &dwChars, NULL);  
  
return 0;  
}
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (9)

```
void ErrorHandler(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code.
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );
}
```

→ DemoCreateThreadProcess → CreateThread.cpp

## Thread creation using CreateThread() (10)

```
// Display the error message.

lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT, (lstrlen((LPCTSTR) lpMsgBuf)
    + lstrlen((LPCTSTR) lpszFunction) + 40) * sizeof(TCHAR));
StringCchPrintf((LPTSTR)lpDisplayBuf,
    LocalSize(lpDisplayBuf) / sizeof(TCHAR),
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);

MessageBox(NULL, (LPCTSTR) lpDisplayBuf, TEXT("Error"), MB_OK);

// Free error-handling buffer allocations.

LocalFree(lpMsgBuf);
LocalFree(lpDisplayBuf);
}
```

→ DemoCreateThreadProcess → CreateThread.cpp

## File management using fopen(), fread(), fwrite()

Source: <https://msdn.microsoft.com/de-de/library/yeby3zcb.aspx>

```
FILE *fopen( const char *filename, const char *mode );           // normal version
FILE *_wfopen(const wchar_t *filename, const wchar_t *mode); // wide character vers.
```

Usage of mode :

- **r** (read only),
- **w** (write only. Delete existing file, if any.)
- **a** (write in append mode / write at the end)
- **R+** (read and write)
- **w+** (read and write. Delete existing file, if any.)
- **a+** (read and write in append mode / write at the end)

### File management using fopen(), fread(), fwrite()

source: <https://msdn.microsoft.com/de-de/library/kt0etdcs.aspx>

```
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
```

source: <https://msdn.microsoft.com/de-de/library/h9t88zwz.aspx>

```
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

## File management using fopen(), fread(), fwrite() – example (1)

```
// File access using fopen() und fwrite() under Windows OS
// source: https://msdn.microsoft.com/de-de/library/kt0etdcs.aspx

// crt_fread.c

// This program opens a file named FREAD.OUT and writes 25 characters to the file.
// It then tries to open FREAD.OUT and read in 25 characters.
// If the attempt succeeds, the program displays the number of actual items read.

#include <stdio.h>

int main( void )
{
    FILE *stream;
    char list[30];
    int i, numread, numwritten;

    // Open file in text mode:
    if( fopen_s( &stream, "fread.out", "w+t" ) == 0 )
    {
```

→ fopen.c

## File management using fopen(), fread(), fwrite() – example (2)

```
for ( i = 0; i < 25; i++ )
    list[i] = (char)('z' - i);
// Write 25 characters to stream
numwritten = fwrite( list, sizeof( char ), 25, stream );
printf( "Wrote %d items\n", numwritten );
fclose( stream );

}

else
    printf( "Problem opening the file\n" );
```

→ fopen.c

## File management using fopen(), fread(), fwrite() – example (3)

```
if( fopen_s( &stream, "fread.out", "r+t" ) == 0 )
{
    // Attempt to read in 25 characters
    numread = fread( list, sizeof( char ), 25, stream );
    printf( "Number of items read = %d\n", numread );
    printf( "Contents of buffer = %.25s\n", list );
    fclose( stream );
}
else
    printf( "File could not be opened\n" );
}
```

→ fopen.c

### File management using CreateFile(), OpenFile(), ReadFile(), WriteFile()

Remarks:

- Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx),
- Check availability; should be available since Windows XP.
- Powerful family of file access functions.
- Huge scope for design.
- Comfort for developers.
- Side effect: Movement of code to different operating systems will be difficult...

## File management, CreateFile() etc. – a list of file management functions

AddUsersToEncryptedFile, AreFileApisANSI, CancelIo, CancelIoEx, CancelSynchronousIo,  
CheckNameLegalDOS8Dot3, CloseEncryptedFileRaw, CopyFile, CopyFile2, CopyFile2ProgressRoutine,  
CopyFileEx, CopyFileTransacted, CopyProgressRoutine, **CreateFile**, CreateFile2,  
CreateFileTransacted, CreateHardLink, CreateHardLinkTransacted, CreateIoCompletionPort,  
CreateSymbolicLink, CreateSymbolicLinkTransacted, DecryptFile, DeleteFile, DeleteFileTransacted,  
DuplicateEncryptionInfoFile, EncryptFile, EncryptionDisable, ExportCallback, FileEncryptionStatus,  
FileIOCompletionRoutine, FindClose, FindFirstFile, FindFirstFileEx, FindFirstFileNameTransactedW,  
FindFirstFileNameW, FindFirstFileTransacted, FindFirstStreamTransactedW, FindFirstStreamW,  
FindNextFile, FindNextFileNameW, FindNextStreamW, FlushFileBuffers,  
FreeEncryptionCertificateHashList, GetBinaryType, GetCompressedFileSize,  
GetCompressedFileSizeTransacted, GetExpandedName, GetFileAttributes, GetFileAttributesEx,  
GetFileAttributesTransacted, GetFileBandwidthReservation, GetFileInformationByHandle,  
GetFileInformationByHandleEx, GetFileSize, GetFileSizeEx, GetFileType, , GetFinalPathNameByHandle,  
GetFullPathName, GetFullPathNameTransacted, GetLongPathName, GetLongPathNameTransacted,  
GetQueuedCompletionStatus, GetQueuedCompletionStatusEx, GetShortPathName, GetTempFileName,  
GetTempPath, ImportCallback, LockFile, LockFileEx, LZClose, LZCopy, LZInit, LZOpenFile, LZRead,  
LZSeek, MoveFile, MoveFileEx, MoveFileTransacted, MoveFileWithProgress, OpenEncryptedFileRaw,  
**OpenFile**, OpenFileById, PostQueuedCompletionStatus, QueryRecoveryAgentsOnEncryptedFile,  
QueryUsersOnEncryptedFile, ReadEncryptedFileRaw, **ReadFile**, ReadFileEx, ReadFileScatter,  
RemoveUsersFromEncryptedFile, ReOpenFile, ReplaceFile, SearchPath, SetEndOfFile,  
SetFileApisToANSI, SetFileApisToOEM, SetFileAttributes, SetFileAttributesTransacted,  
SetFileBandwidthReservation, SetFileCompletionNotificationModes, SetFileInformationByHandle,  
SetFileIoOverlappedRange, SetFilePointer, SetFilePointerEx, SetFileShortName, SetFileValidData,  
SetSearchPathMode, SetUserFileEncryptionKey, UnlockFile, UnlockFileEx,  
Wow64DisableWow64FsRedirection, Wow64EnableWow64FsRedirection, Wow64RevertWow64FsRedirection,  
WriteEncryptedFileRaw, **WriteFile**, WriteFileEx, WriteFileGather

## File management using CreateFile(), C++

Quelle: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx)

```
HANDLE WINAPI CreateFile( _In_ LPCTSTR lpFileName, _In_ DWORD dwDesiredAccess,
_In_ DWORD dwShareMode, _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
_In_ DWORD dwCreationDisposition, _In_ DWORD dwFlagsAndAttributes,
_In_opt_ HANDLE hTemplateFile );
```

Parameter:

- **lpFileName [in]:** Filename
- **dwDesiredAccess [in]:** Access, e. g. GENERIC\_READ, GENERIC\_WRITE
- **dwShareMode [in]:** how to deal with concurrent access, e. g. NULL, FILE\_SHARE\_READ, FILE\_SHARE\_WRITE
- **lpSecurityAttributes [in, optional]:** How to deal with child processes, inheritance aspects:  
Inheritance of file handle?
- **dwCreationDisposition [in]:** How to treat already existing files / first creation:  
e. g. CREATE\_ALWAYS, OPEN\_EXISTING
- **dwFlagsAndAttributes [in]:** File attributes, e. g. FILE\_ATTRIBUTE\_ENCRYPTED, FILE\_ATTRIBUTE\_HIDDEN
- **hTemplateFile [in, optional]:** Template to copy file attributes from an existing file.

## File management using OpenFile(), C++

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365430\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365430(v=vs.85).aspx)

```
HFILE WINAPI OpenFile( _In_  LPCSTR lpFileName, _Out_  LPOFSTRUCT lpReOpenBuff,  
                      _In_  UINT uStyle );
```

Parameter:

- **lpFileName [in]:**      Filename
- **lpReOpenBuff [out]:**    A pointer to the [OFSTRUCT](#) structure that receives information about a file when it is first opened.
- **uStyle [in]:**            Action, e. g. `OF_CREATE` , `OF_EXIST` , `OF_READWRITE` , `OF_REOPEN`

## File management using ReadFile(), C++

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx)

```
BOOL WINAPI ReadFile( _In_ HANDLE hFile, _Out_ LPVOID lpBuffer,
                      _In_ DWORD nNumberOfBytesToRead,
                      _Out_opt_ LPDWORD lpNumberOfBytesRead,
                      _Inout_opt_ LPOVERLAPPED lpOverlapped );
```

### Parameter:

- **hFile [in]:** handle of file, file stream, volume, socket, mailslot, pipe
- **lpBuffer [out]:** A pointer to the buffer that receives the data read from a file or device.
- **nNumberOfBytesToRead [in]:** The maximum number of bytes to be read.
- **lpNumberOfBytesRead [out, optional]:** Pointer to the variable that receives the number of bytes read.
- **lpOverlapped [in, out, optional]:** NULL, or pointer to an [OVERLAPPED](#) structure, if file is already open in [FILE\\_FLAG\\_OVERLAPPED](#) mode

## File management using WriteFile(), C++

Quelle: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx)

```
BOOL WINAPI WriteFile( _In_ HANDLE hFile, _In_ LPCVOID lpBuffer,
                      _In_ DWORD nNumberOfBytesToWrite,
                      _Out_opt_ LPDWORD lpNumberOfBytesWritten,
                      _Inout_opt_ LPOVERLAPPED lpOverlapped );
```

Parameter:

- **hFile [in]:** handle of file, file stream, volume, socket, mailslot, pipe
- **lpBuffer [out]:** A pointer to the buffer containing the data to be written to the file or device..
- **nNumberOfBytesToWrite [in]:** Number of bytes to write.
- **lpNumberOfBytesWritten [out, optional]:** A pointer to the variable  
that receives the number of bytes written
- **lpOverlapped [in, out, optional]:** NULL, or pointer to an [OVERLAPPED](#) structure,  
if file is already open in FILE\_FLAG\_OVERLAPPED mode

### Function socket()

// Source: [https://msdn.microsoft.com/de-de/library/windows/desktop/ms740506\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/ms740506(v=vs.85).aspx)

```
SOCKET WSAAPI socket( __in int af, __in int type, __in int protocol );
```

**af [in]**: The address family specification, e. g. **AF\_INET**= Internet Protocol v. 4

**type [in]**: The type specification for the new socket, e. g. **SOCK\_STREAM**(=TCP)

**protocol [in]**: The protocol to be used, e. g.  **IPPROTO\_TCP**

### Functions accept(), bind(), listen()

// Source: [https://msdn.microsoft.com/de-de/library/windows/desktop/ms737526\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/ms737526(v=vs.85).aspx)

```
SOCKET accept( __in SOCKET s, __out struct sockaddr *addr, __inout int *addrlen );
```

// Quelle: [https://msdn.microsoft.com/de-de/library/windows/desktop/ms737550\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/ms737550(v=vs.85).aspx)

```
int bind( __in SOCKET s, __in const struct sockaddr *name, __in int namelen );
```

// Quelle: [https://msdn.microsoft.com/de-de/library/windows/desktop/ms739168\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/ms739168(v=vs.85).aspx)

```
int listen( __in SOCKET s, __in int backlog );
```

# Windows interprocess communication

- Clipboard:
- COM: Component Object Model, communication using client-/server methods.  
DCOM/Distributed COM: Spread over the network.
- Data Copy: Usage of the Windows message WM\_COPYDATA.
- DDE: Dynamic Date Exchange. Like the clipboard, old fashioned.
- **File Mapping:** Content of file will be mapped to main memory for pointer access.  
Aspects of synchronization have to be handled separately.
- Mailslots: Unidirectional communication for short messages, even from one computer to another. Mailslots will be addressed by their names.  
Broadcast possible (if multiple mailslots with the same name).
- Pipes:
  - Anonymous pipes: Communication between related processes.  
Typical situation: Pipe the standard IO of a child process to its parent.
  - **Named pipes:** Communication between processes, name of pipe must be known.
- RPC: Remote Procedure Call. Similar to a function call, but done by another process.  
Communication between different operating systems possible.  
Open Software Foundation (OSF) spec. Distributed Computing Environment (DCE).
- Windows Sockets: Based on sockets of Berkeley Software Distribution, but have been developed since then. Communication between different OS is possible, if only the basic features are used.

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx)

## Named Pipes

Features of named pipes:

- Unidirection or bidirectional
- 1 pipe server
- Multiple pipe clients
- Usage between different processes
- Access via name
- Individual instances for each process  
(individual channel, individual buffers, individual handles)!

Actions of the pipe server:

- Create an instance of the named pipe via **CreateNamedPipe()**
- Wait for connection requests done by a client and establish a connection via **ConnectNamedPipe()**

Actions of the pipe client:

- Connect to a pipe via **CreateFile()** or **CallNamedPipe()**

Quelle: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365590\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365590(v=vs.85).aspx)

## CreateNamedPipe()

```
HANDLE WINAPI CreateNamedPipe(_In_ LPCTSTR lpName, _In_ DWORD dwOpenMode,
                           _In_ DWORD dwPipeMode,
                           _In_ DWORD nMaxInstances, _In_ DWORD nOutBufferSize,
                           _In_ DWORD nInBufferSize, _In_ DWORD nDefaultTimeOut,
                           _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Parameters:

- **lpName [in]**: Name of the pipe. Pattern: <\\.\pipe\pipename>
- **dwOpenMode [in]**: Some flags for opening, e. g. **PIPE\_ACCESS\_DUPLEX**,  
**PIPE\_ACCESS\_INBOUND**, **FILE\_FLAG\_WRITE\_THROUGH**
- **dwPipeMode [in]**: Some flags for usage, e. g. **PIPE\_TYPE\_BYTE**, **PIPE\_NOWAIT**
- **nMaxInstances [in]**: Maximum number of instances of this pipe.
- **nOutBufferSize [in]**: Size of output buffer.
- **nInBufferSize [in]**: Size of input buffer.
- **nDefaultTimeOut [in]**: Default time-out [milliseconds]. Default = 50 msec.
- **lpSecurityAttributes [in, optional]**: How to deal with child processes:  
Inheritance of the file handle?

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365150\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365150(v=vs.85).aspx)

## ConnectNamedPipe()

```
BOOL WINAPI ConnectNamedPipe(_In_ HANDLE hNamedPipe,
                            _Inout_opt_ LPOVERLAPPED lpOverlapped );
```

### Parameters:

- **hNamedPipe [in]**: Handle of the pipe.
- **lpOverlapped [in, out, optional]**: Pointer to an **OVERLAPPED** structure  
(mode **FILE\_FLAG\_OVERLAPPED**)

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365146\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365146(v=vs.85).aspx)

## CallNamedPipe()

```
BOOL WINAPI CallNamedPipe(_In_ LPCTSTR lpNamedPipeName, _In_ LPVOID lpInBuffer,
                           _In_ DWORD nInBufferSize, _Out_ LPVOID lpOutBuffer,
                           _In_ DWORD nOutBufferSize, _Out_ LPDWORD lpBytesRead,
                           _In_ DWORD nTimeOut );
```

Parameters:

- **lpNamedPipeName [in]**: Name of the pipe.
- **lpInBuffer [in]** The data to be written to the pipe.
- **nInBufferSize [in]** The size of the write buffer, in bytes.
- **lpOutBuffer [out]** A pointer to the buffer that receives the data read from the pipe.
- **nOutBufferSize [in]** The size of the read buffer, in bytes.
- **lpBytesRead [out]** Pointer to a variable that receives the number of bytes read from the pipe.
- **nTimeOut [in]** Max time until pipe is ready to use.  
NMPWAIT\_NOWAIT or NMPWAIT\_WAIT\_FOREVER are possible too.

Demo project: **NamedPipes → NamedPipeServer.cpp + NamedPipeClient.cpp**

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365144\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365144(v=vs.85).aspx)

## File mapping + memory mapping

By usage of these methods, files and memory may be used by different processes.

### Sharing files:

1) (1. Process:) `Handle ← CreateFile()`

2) (1. Process:) `CreateFileMapping (Handle)`

3) (2. Process:) `OpenFileMapping ()`

4) (both:) `MapViewOfFile ()`

5) (both:) use of shared file

6) (both:) `UnmapViewOfFile ()`

7) (both:) `CloseHandle ()`

### Sharing memory:

1) (1. Process:) `CreateFileMapping (`

`INVALID_HANDLE_VALUE )`

2) (2. Process:) `OpenFileMapping ()`

3) (both:) `MapViewOfFile ()`

4) (both:) use of shared memory

5) (both:) `UnmapViewOfFile ()`

6) (both:) `CloseHandle ()`

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366878\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366878(v=vs.85).aspx)

## File mapping + memory mapping

```
HANDLE WINAPI CreateFileMapping(_In_ HANDLE hFile,
                                _In_opt_ LPSECURITY_ATTRIBUTES lpAttributes,
                                _In_ DWORD f1Protect, _In_ DWORD dwMaximumSizeHigh,
                                _In_ DWORD dwMaximumSizeLow, _In_opt_ LPCTSTR lpName );
```

Parameters:

- **hFile [in]** A handle to the file oder **INVALID\_HANDLE\_VALUE** (→ Memory Mapping)
- **lpAttributes [in, optional]** How to deal with child processes:  
Do childs inherit the handle of the FileMapping object?
- **f1Protect [in]** Page protection of the file mapping objects,  
e. g. **PAGE\_READONLY**, **PAGE\_READWRITE**
- **dwMaximumSizeHigh [in]** Maximum size of the file mapping object (high order DWORD).
- **dwMaximumSizeLow [in]** Maximum size of the file mapping object (low order DWORD).
- **lpName [in, optional]** The name of the file mapping object.

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366537\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366537(v=vs.85).aspx)

## File mapping + memory mapping

```
LPVOID WINAPI MapViewOfFile(_In_ HANDLE hFileMappingObject,
                            _In_ DWORD dwDesiredAccess,
                            _In_ DWORD dwFileOffsetHigh,
                            _In_ DWORD dwFileOffsetLow,
                            _In_ SIZE_T dwNumberOfBytesToMap );
```

Parameter:

- **hFileMappingObject [in]** Handle
- **dwDesiredAccess [in]** Access mode, e. g. **FILE\_MAP\_READ**.
- **dwFileOffsetHigh [in]** A high-order DWORD of the file offset where the view begins.
- **dwFileOffsetLow [in]** A low-order DWORD of the file offset where the view is to begin.
- **dwNumberOfBytesToMap [in]** The number of bytes of a file mapping to map to the view.

Demo project: **FileMapping → FileMappingDemoProcess1.cpp + FileMappingDemoProcess2.cpp**

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366761\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366761(v=vs.85).aspx)

# **Betriebssysteme / Operating Systems**

## **Windows II: Ressource access, timing, I/O**

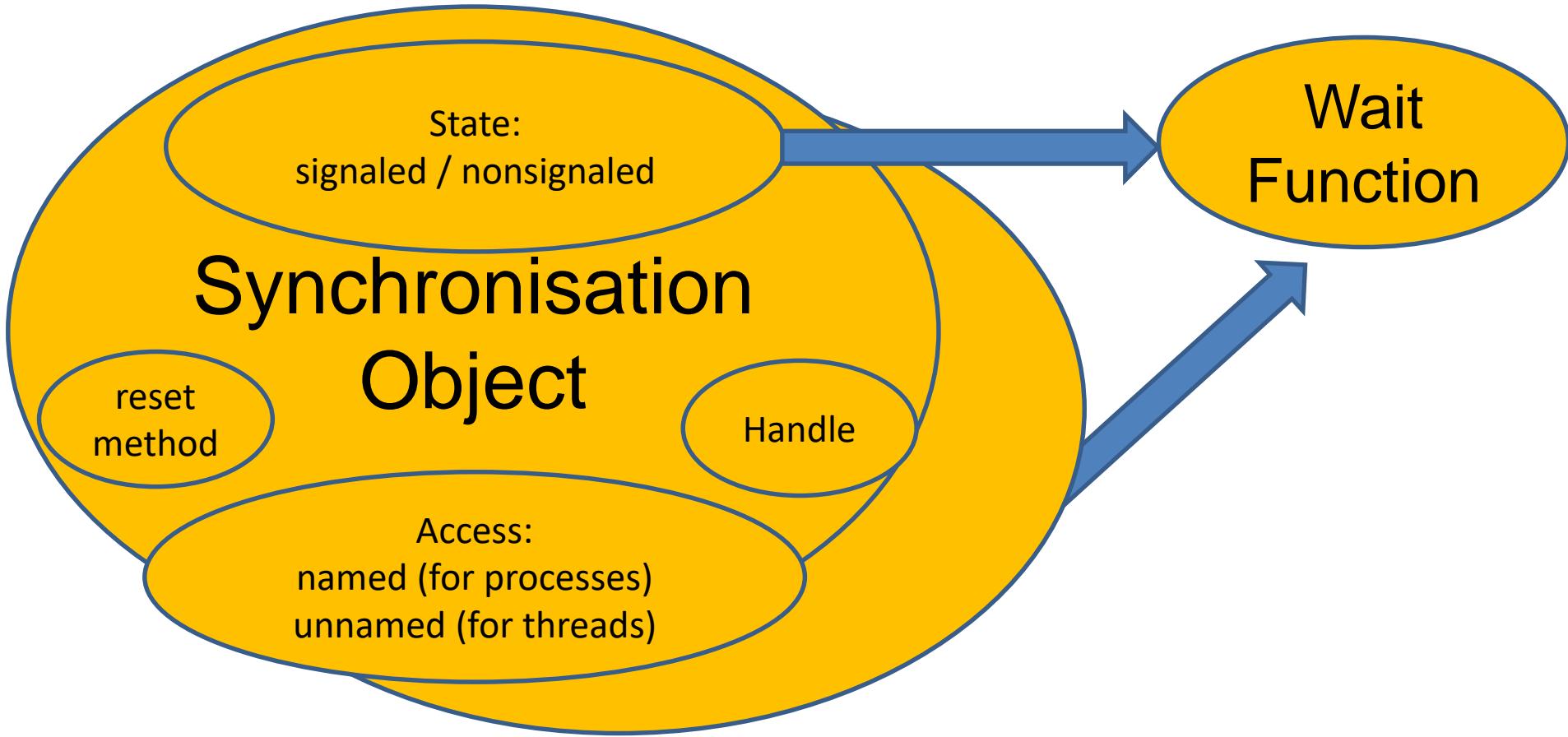
SS 2019

Prof. Dr.-Ing. Holger Gräßner

16 OS-BS 2019 Windows II.pptx

## Synchronisation with Windows OS

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms686353\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686353(v=vs.85).aspx)



## Synchronisation Objects

Quelle: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms686364\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686364(v=vs.85).aspx)

### Features

Handle for access

Name: To get access from different processes

State: Signaled oder nonsignaled

Reset method: manually or automatically

Objects for synchronization:

**Events**,

**Mutex**

**Semaphore**

**Waitable timer**

## Wait Functions

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms687069\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687069(v=vs.85).aspx)

Wait functions:

- **Single-object Wait Functions**
- **Multiple-object Wait Functions**
- Alertable Wait Functions
- Registered Wait Functions
- Waiting on an Address
- Wait Functions and Time-out Intervals
- Wait Functions and Synchronization Objects
- Wait Functions and Creating Windows



**WaitForXYZObject()** functions:

- **WaitForSingleObject()** :  
Wait for the signaled state of a single object
- **WaitForMultipleObjects()** :  
Wait for the signaled state of multiple objects

Both:

- Timeout conditions available
- Reset of the signaled object will be done automatically (if configured)
- Return value: Handle of the signaled object

### WaitForSingleObject()

```
DWORD WINAPI WaitForSingleObject( _In_ HANDLE hHandle,
                                  _In_ DWORD   dwMilliseconds );
```

Parameters:

**hHandle** [in] handle

**dwMilliseconds** [in] Timeout [milliseconds]

Return value: Reason (**WAIT\_OBJECT\_0**, **WAIT\_TIMEOUT**, **WAIT\_FAILED**)

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms687032\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687032(v=vs.85).aspx)

## WaitForMultipleObjects()

```
DWORD WINAPI WaitForMultipleObjects( _In_ DWORD nCount,
                                     _In_ const HANDLE *lpHandles,
                                     _In_ BOOL bWaitAll, _In_ DWORD dwMilliseconds );
```

Parameters:

**nCount [in]** Number of handles to wait for

**lphHandles [in]** Array of handles to wait for

**bWaitAll [in]** TRUE: Function will return, if all objects are signaled,  
FALSE: Function will return, if one of the objects is signalled.

**dwMilliseconds [in]** Timeout [milliseconds]

Return value: Index of the signalled object (**WAIT\_OBJECT\_0**, **WAIT\_OBJECT\_0 + 1**, ...,  
**WAIT\_TIMEOUT**, **WAIT\_FAILED**)

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms687025\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687025(v=vs.85).aspx)

### Events are pure synchronization objects:

State:

- **Signaled**
- **NonSignaled**

Reset mode:

- Automatical reset: Reset as soon as this event raised a **WaitFor** function
- Manual reset: Reset by use of **ResetEvent()**.

Access functions:

**CreateEvent()**: Create an events (with name, if needed), get handle.

**OpenEvent()**: Open an existing event (with name, if needed).

**SetEvent()**: Set event's state to **Signaled**.

**ResetEvent()**: Set event's state to **Nonsignaled**.

Demos: **Events WaitForMultipleObjects.cpp**

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms682655\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx)

# Mutexes – synchronization objects with a special behaviour:

State:

- **Signaled**, as soon as free (not blocked by any thread),
- **NonSignaled**, while blocked by a thread.

Access functions:

- CreateMutex ()**: Create mutex, get handle.
- OpenMutex ()**: Open an existing mutex, get handle (with name, if needed), get handle.
- WaitForxyz ()**: Wait for mutex and access to it. Will be in **NonSignaled** state automatically.
- ReleaseMutex ()**: Release mutex, set to **Signaled** state.

Examples:

**Mutex Windows .cpp**: Access without using names (communication between threads)

**Named Objects Process 1 .cpp / Named Objects Process 1 .cpp**:

Access via names (communication between processes)

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms684266\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684266(v=vs.85).aspx)

# Semaphores – counting synchronization objects:

Counter between 0 and maximum value

State:

- **Signaled**, if counter > 0 (just one waiting thread at a time will be set to „running“),
- **NonSignaled**, if counter = 0.

Access:

- Decrement every time a thread finished it's **wait** for this semaphore,
- Increment with **ReleaseSemaphore ()**.

Access functions:

**CreateSemaphore ()**: Create a semaphore, get handle.

**OpenSemaphore ()**: Open an existing semaphore (using name, if needed). Get handle.

**Waitforxyz ()**: Wait for semaphore, get access. **NonSignaled** state automatically.

**ReleaseSemaphore ()**: Increment semaphore

**CloseHandle ()**: Release semaphore's handle

Example: **Semaphore Windows .cpp**

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms685129\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685129(v=vs.85).aspx)

### Waitable timer – state will change after a specified time:

State:

- **Signaled**, if timer expired,
- **NonSignaled**, otherwise.

Methods to reset timer:

- Manual Reset Timer stays in **signaled** state until next call of **SetWaitableTimer()**
- Synchronization Timer will be set to **NonSignaled** after every call of **wait()**.
- Periodic Timer will be reactivated after a specific period.

Access functions:

**CreateWaitableTimer()**: Create timers, get handle.

**OpenWaitableTimer()**: Open an existing timer (using name, if needed), get handle.

**Waitforxyz()**: Wait for timer.

**ResetWaitableTimer()**: Set timer to inactive.

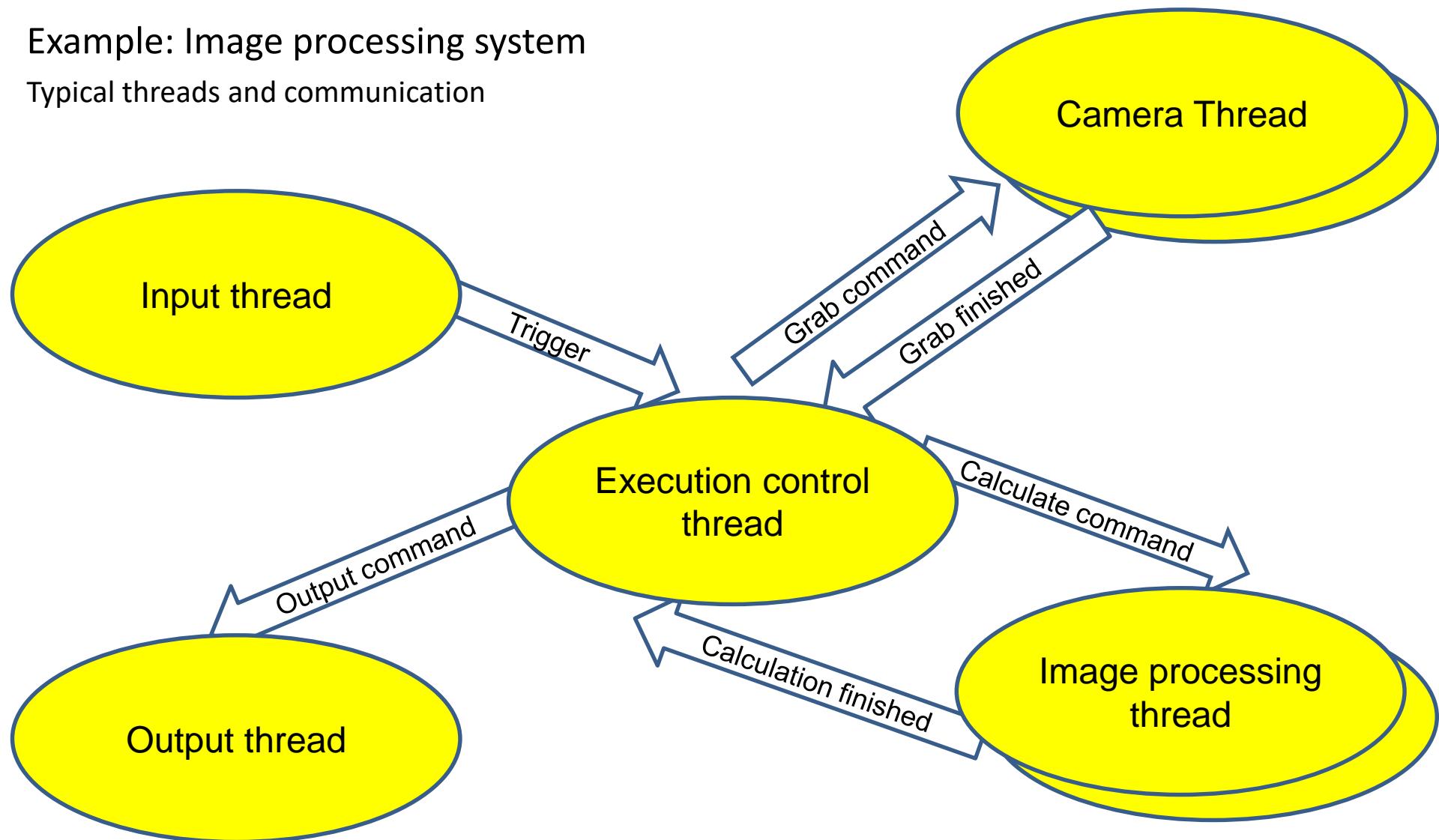
**CloseHandle()**: Release timer's handle.

Example: **Waitable Timer.cpp**

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms687012\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687012(v=vs.85).aspx)

## Example: Image processing system

Typical threads and communication



### Example: Image processing system

Visual studio solution containing a small, executable vision system:

`VisionSystem-empty\VisionSystem.sln`.

This solution contains three projects, each of them to generate a separate program:

- ExecutionControl:  
Creates the TriggerEvent and waits until it gets signaled by another process..
- InputOutput:  
Opens the TriggerEvent and sets its state to „signaled“ after a hit of the key <T>.
- ImageProcessing:  
Creates an empty thread for image processing.

Task:

Add all necessary interprocess communication to get a full functional vision system!

→ `VisionSystem-full\VisionSystem.sln`.

# Windows Messages and Message Queues

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms644927\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms644927(v=vs.85).aspx)

Windows' communication

# **Betriebssysteme / Operating Systems**

## **Windows III: Applications with windows**

SS 2020

Prof. Dr.-Ing. Holger Gräßner

17 OS-BS 2020 WindowsIII.pptx



# Architectural principles

## Common architecture

Programs are responsible to read information actively, e. g. via

- `fgetc()`
- `fread()`

## Windows program architecture / applications with windows

Program waits for external events.

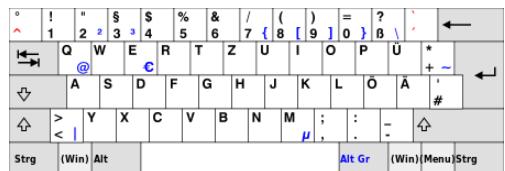
System sends information to the application.

Template for applications' architecture.

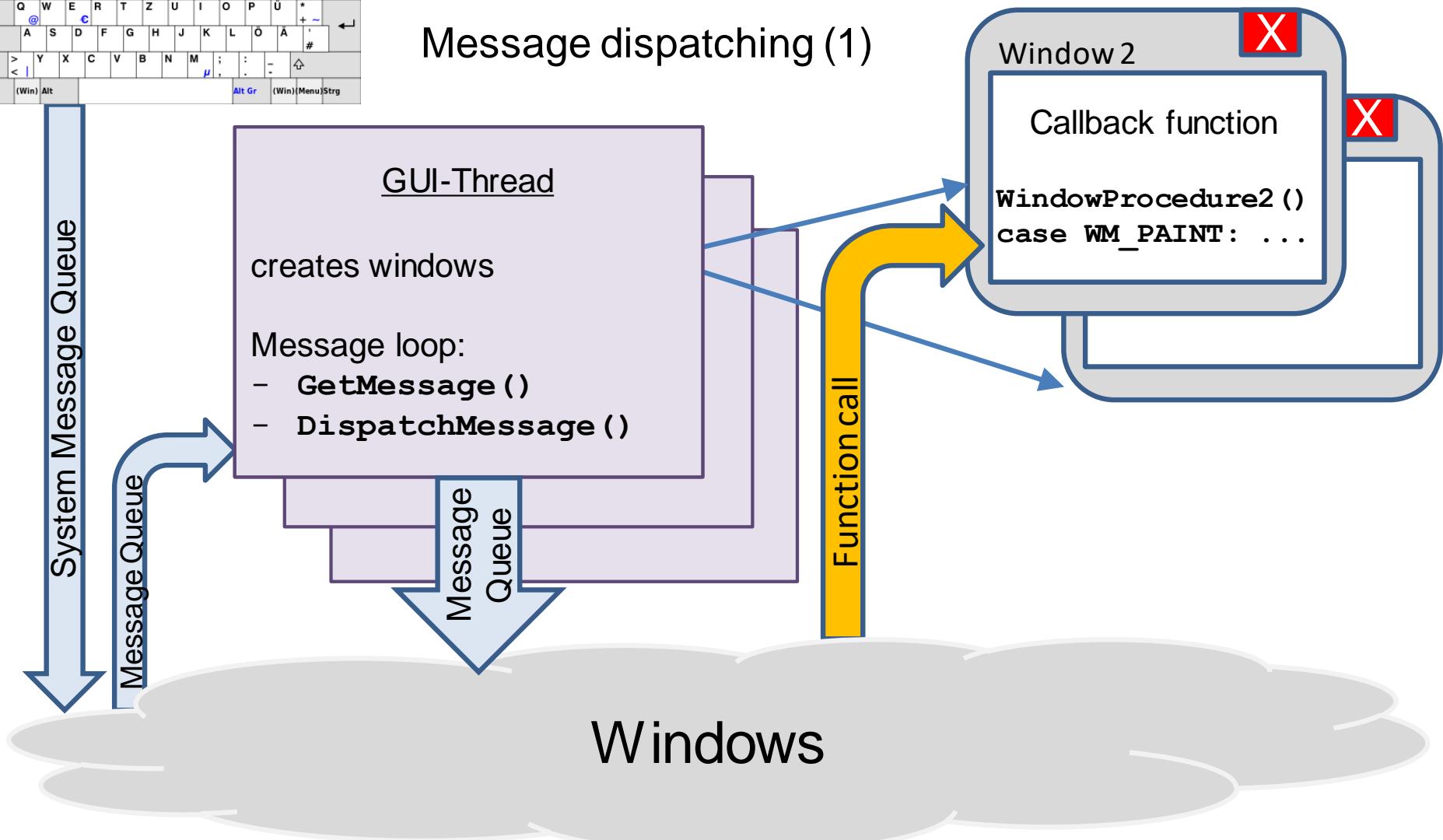
System sends information to every window of an application.

**Message** information will be send by use of message queues.

Source: <https://msdn.microsoft.com/de-de/library/windows/desktop/ms644927>



## Message dispatching (1)



### Message dispatching (2)

1. Device driver: Process keyboard hits, mouse clicks etc. → generate messages for the system message queue.
2. Operating system: Get these messages from system message queue.
3. Operating system: Identify the target window for the message.
4. Operating system: Put the message to the message queue of the GUI thread.
5. GUI Thread: Run a message loop:
  - a. `GetMessage()` from message queue
  - b. `TranslateMessage()`, if keyboard hits are to be processed.
  - c. Call the associated window procedure via `DispatchMessage()`.  
These messages will be delivered to one or multiple of associated windows.
  - d. End of message loop: As soon as a `WM_QUIT` message occurs.

### Message dispatching (3)

6. Any thread may generate messages:
  - a. For a specific window via `PostMessage()` .
  - b. For a specific thread via `PostThreadMessage()` .
7. The following steps are required to bring all relevant messages to the associated window:
  - a. Create the window by the GUI thread.
  - b. Show the window.
  - c. Update the window.
  - d. Implement a callback function for messages.
  - e. Register this handler function via `RegisterClassEx()` .
  - f. Process all messages for the window by this callback function.

# Messages

## Queued messages:

- Global system message queue.
- Specific message queue for each GUI thread.

This queue will be created by Windows as soon as the threads calls a specific user function. This will make him become the GUI thread.

## Nonqueued messages:

- Will be sent to the associated windows by Windows OS,
- Does not require a system message queue or thread message queue.
- Examples:
- Window manipulation by the user (e. g. message **WM\_WINDOWPOSCHANGED**)
- Special functions like **BroadcastSystemMessage()**.

### Message types

System defined messages:

- message identifier = 0...**WM\_USER** -1.
- **DMxyz**: Dialog messages,
- **HKMxyz**: Hot key messages,
- **PBMxyz**: Progress bar messages,
- **WMxyz**: General messages.

Application defined messages:

- May be picked randomly from the set **WM\_USER**, **WM\_USER+1**, . . . **WM\_APP-1**,
- A unique message ID can be ordered via **RegisterWindowMessage()**.

..

## Message structure

Message structure for usage by the message loop

```
typedef struct tagMSG {  
    HWND    hwnd;        // Window-Handle, if dedicated to window procedure,  
                        // NULL, if dedicated to owner thread  
    UINT    message;    // Message ID  
    WPARAM wParam;    // Additional information, context sensitive  
    LPARAM lParam;    // Additional information, context sensitive  
    DWORD   time;       // Time stamp of message trigger  
    POINT   pt;         // Cursor position of message trigger  
} MSG,
```

Source: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms644958\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms644958(v=vs.85).aspx)

# Window procedure / window's callback function

```
LRESULT CALLBACK MyWindowProcedure(
    HWND hWnd,          // Window handle
    UINT msg,           // Message-ID
    WPARAM wParam,     // additional information - context
    LPARAM lParam) {  // additional information - context
    switch (msg) {
        case WM_CREATE:
            // . .
            break;
        // . .
    }
    return 0L;
}
```

Source: <https://msdn.microsoft.com/de-de/library/windows/desktop/ms633573>

List of system messages:

[https://msdn.microsoft.com/de-de/library/windows/desktop/ms644927#system\\_defined](https://msdn.microsoft.com/de-de/library/windows/desktop/ms644927#system_defined)

## Windows-Demo „Countdown“ (1)

```
// global variables:  
  
int nAlarmTime = 10; // Countdown seconds  
HDC hDC; // handle device context - for output to a window  
  
// forward declaration of functions:  
  
// Callback function for messages to be processed by our window:  
HRESULT CALLBACK TimerWindowCallback(HWND hWnd, UINT msg,  
                                     WPARAM wParam, LPARAM lParam);  
  
// Auxiliary function: Display the remaining time in our window  
void WriteTime(HWND hWnd);
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (2)

```
// Entrance function WinMain():

int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
             LPSTR lpszCmdLine, int cmdShow)
{
    // Define application's main window and register it:

    WNDCLASSEX     wndclass;
    wndclass.lpfnWndProc = TimerWindowCallback;      // callback function to
   // process messages to our window
    wndclass.hInstance = hInstance;   // window is assigned to this instance
    // ...

    RegisterClassEx(&wndclass);      // register the window
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (3)

```
// . . . still WinMain() . . .

// Create application's main window:

HWND hwnd = CreateWindow(TEXT("TIMER:MAIN") , // windows's class name
                         TEXT("Countdown") , // window title
                         (WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME) , // window's attributes
                         1, 1, 190, 70 , // location and size
                         NULL , // no parent window
                         NULL , // no separate menu required;
                         // we will just make use of the class menu
                         hInstance , // windows's owner / creator
                         NULL) ; // no parameter to pass
```

→ Timer lecture.sln → Timer lecture.cpp

### Windows-Demo „Countdown“ (4)

```
// . . . still WinMain() . . .

// Show application's main window:
ShowWindow(hwnd, cmdShow);

// prepare output to main window:
hDC = GetDC(hwnd);      // Get device context for output operations
WriteTime(hwnd);        // write remaining time
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (5)

```
// . . . still WinMain() . . .

// Main thread's message loop:
MSG msg;

while (GetMessage(&msg, NULL, 0, 0)) { // Get message from thread's
   // queue...
    TranslateMessage(&msg); // ... Translate message
                           // (e. g. keyboard input)...
    DispatchMessage(&msg); // ... Dispatch messages to dedicated windows
}

// End of loop - we received WM_QUIT
// (this has been send by our own window function)
ReleaseDC(hwnd, hDC); // release device context
return(0);
}
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (6)

```
// Callback function TimerWindowCallback()
// to process messages to our window:

LRESULT CALLBACK TimerWindowCallback(HWND hWnd, UINT msg,
                                      WPARAM wParam, LPARAM lParam)
{
    #define ID_EVENT 1
    PAINTSTRUCT ps;

    switch (msg) {
        case WM_CREATE:                      // (first) creation fo the window
            SetTimer(hWnd, ID_EVENT, 1000, NULL); // create timer:
  // 1000 ms elapsed time,
        // timer function NULL ==> we will get a WM_TIMER message, if timer elapses
        break;
    }
}
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (7)

```
// . . . still TimerWindowCallback() . . .

case WM_TIMER:                      // one timer message every second
    nAlarmTime--;                    // countdown seconds
    WriteTime(hWnd);               // show remaining time
    if (nAlarmTime == 0) {          // now t=0

        // Show message Box to be acknowledged by the user
        // (this function blocks, until OK has been clicked):
        MessageBox(hWnd, TEXT(„It's tea time!“),
                    TEXT("Alarm!"), MB_OK);

        KillTimer(hWnd, ID_EVENT);   // delete timer
        PostQuitMessage(0);          // place WM_QUIT message in thread's queue
    }
break;
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (8)

```
// . . . still TimerWindowCallback() . . .

case WM_DESTROY:          // window's close box has been clicked
    KillTimer(hWnd, ID_EVENT);
    PostQuitMessage(0);    // place WM_QUIT message in thread's queue
    break;

case WM_PAINT:            // window's content has to be refreshed
    BeginPaint(hWnd, &ps);
    WriteTime(hWnd);    // show remaining time
    EndPaint(hWnd, &ps);
    break;
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (9)

```
// . . . still TimerWindowCallback() . . .

case WM_SYSCOMMAND:           // Window's menu, close box etc.
default:                      // all other actions
    return(DefWindowProc(hWnd, msg, wParam, lParam)); // just use
  // default operation
}
return 0L;
}
```

→ Timer lecture.sln → Timer lecture.cpp

## Windows-Demo „Countdown“ (10)

```
// Auxiliary function: Display the remaining time in our window

void WriteTime(HWND hWnd)
{
    RECT R;
    GetClientRect(hWnd, &R); // get the rectangular area for window output

#define TEXTBUFFERSIZE 100
    TCHAR Buffer[TEXTBUFFERSIZE];
    StringCbPrintf(Buffer, TEXTBUFFERSIZE * sizeof(TCHAR),
                    TEXT(„%d seconds remaining“), nAlarmTime);

    TextOut(hDC, R.right / 2, R.bottom / 2 + 6, Buffer, _tcslen(Buffer));
}
```

→ Timer lecture.sln → Timer lecture.cpp

You have to set up a server to provide Lotto numbers.

Environment: Linux – unfortunately without any inetd superdaemont available.

Programming language: C.

Task:

- Name all OS calls to be used by your server program!
- Specify the C functions and describe their purpose!

## Synchronisation exercise „meat balls for the king”

At the king's palace, two chefs are serving meat balls to their king.

The king is very hungry and attacks everything on his dish with the fork immediately.

The chefs are afraid to get injured by the royal fork.

Write a program to avoid any danger for the chefs. Use semaphores!

The king should not pick with his fork, if there is no food on his dish.

Name the Linux calls you need!

Find the errors in the following program!

Correct the code to get the expected behaviour of the program!

```
pthread_mutex_t mutex;

int printFrac(int n) {
    if (n==0) return 1;
    printf("1/%d=%f\n", n, 1.0/(float) n);
    return 0;
}

void *thread2(void *data) {
    for (int i=0; i<300; i++) {
        if (pthread_mutex_lock(&mutex) != 0)
            return EXIT_FAILURE;
        if (printFrac(i) != 0)
            return EXIT_FAILURE;
        if (pthread_mutex_unlock(&mutex) != 0)
            return EXIT_FAILURE;
    }
}
```

```
int main(void) {
    if (pthread_mutex_init(&mutex, NULL) != 0)
        return EXIT_FAILURE;
    if (pthread_create(&tid, NULL, thread2, NULL) != 0)
        return EXIT_FAILURE;
    for (int i=100; i<200; i++) {
        if (pthread_mutex_lock(&mutex) != 0)
            return EXIT_FAILURE;
        if (printFrac(i) != 0) return EXIT_FAILURE;
        if (pthread_mutex_unlock(&mutex) != 0)
            return EXIT_FAILURE;
    }
    if (pthread_mutex_destroy(&mutex) != 0)
        return EXIT_FAILURE;
    return EXIT_SUCCESS;
}
```

Have a look at the following attempt to make use of shared memory.

Name the problem!

Suggest a better solution!

```
sd = shm_open(name, flags, rights);  
char *ptr = mmap(0, len, prot, MAP_SHARED, sd, offset);  
sprintf(sd, "Hello World!");
```