# „Systemnahe Programmierung“
# (BA-INF 034)
# Wintersemester 2023/2024

Dr. Matthias Frank, Dr. Matthias Wübbeling

Institut für Informatik 4

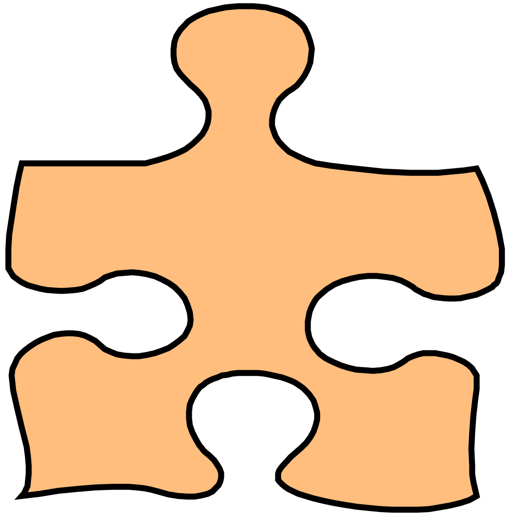
Universität Bonn

E-Mail: *{matthew, matthias.wuebbeling}@cs.uni-bonn.de*

Sprechstunde: nach der Vorlesung bzw. nach Vereinbarung

# 2. Fortgeschrittene Konzepte der Systemprogrammierung

2. Betriebssysteme/Threads
„Kommunikation innerhalb von Prozessen bzw. zwischen Prozessen eines Rechners“



Teil 1

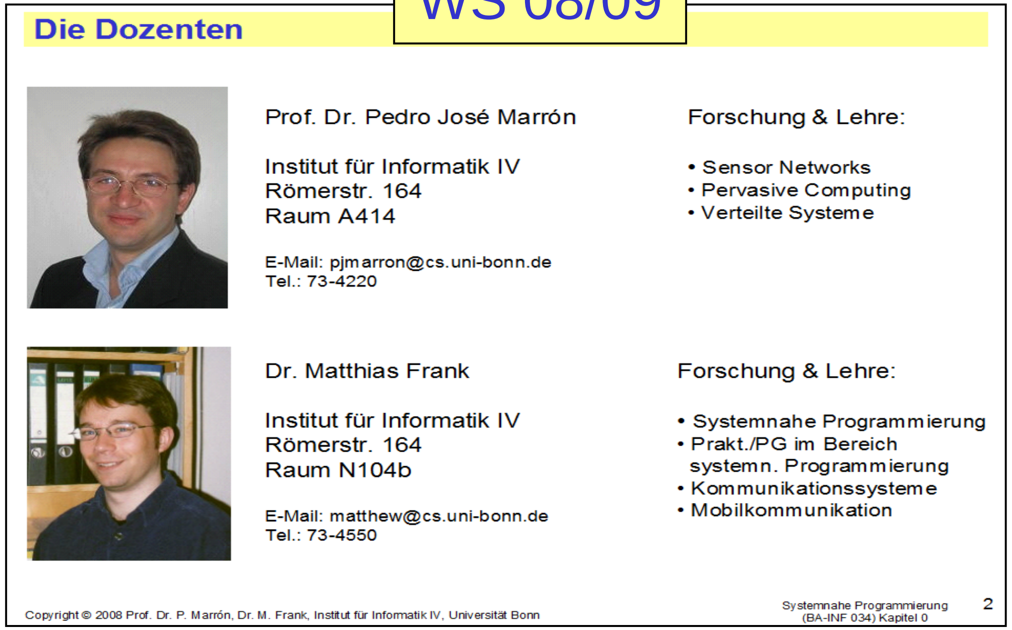
- 2.1. Processes
- 2.2. Threads
- 2.3. Interprocess Communication IPC

# Danksagung - Acknowledgement

WS 08/09

- **WS 2008/2009 erste Durchführung**
  dieser Vorlesung im Bachelor-Studiengang
  mit zwei Dozenten.

- **WS 2014/2015 erste Durchführung**
  Vorlesung mit Go





- Im Wesentlichen werden die theoretischen Grundlagen von Prof. Marrón und Mitarbeitern in unserem **Kapitel 2** weitergeführt.

- Die **Folien** waren und bleiben **in Englisch**,
  da sie sich eng an der vorhandenen (klassischen)
  englisch-sprachigen Literatur orientieren.

# 2.1. Processes - Literature

- [Tanenbaum07] Andrew S. Tanenbaum: **Modern Operating Systems** (International Ed. of 3rd Revsied Edition), *Prentice Hall International / Pearson International Edition*, 2013

„Moderne Betriebssysteme“
3. Auflage deutschsprachig, April 2009

- [Silberschatz08] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: **Operating System Concepts** (8th Edition), *Wiley & Sons*, 2008

- [Mitchell01] Mark Mitchell, Jeffrey Oldham, Alex Samuel: **Advanced Linux Programming** (1st Edition), *New Riders Publishing*, June 2001,
  Online: http://www.advancedlinuxprogramming.com

9. Auflage, International Student Version
Mai 2013

# Outline

- **2.1.1. Overview**
  - Motivation
  - Concept
  - Implementation
  - Scheduling
  - Basic operations
- **2.1.2. Utilization**
  - Examples (Linux)
- **2.1.3. Summary**

# Outline

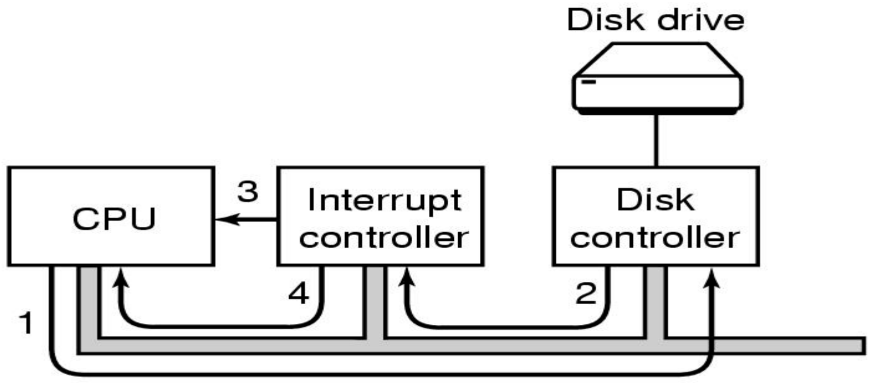
- **2.1.1. Overview**
  - Motivation
  - Concept
  - Implementation
  - Scheduling
  - Basic operations
- 2.1.2. Utilization
  - Examples (Linux)
- 2.1.3. Summary

# 2.1.1. Overview – High-level Processor Review

[Tanenbaum01]

- Sequential execution of one program
  - Fetches instruction at program counter (PC)
  - location, executes instruction and increases PC

- Interrupt-driven hardware handling
  - I/O controllers signal events via interrupts
  - changes PC location to specific value




**=> CPU only knows one program** but **user wants to execute several** in parallel

- Note: current hardware is more complicated
  - Pipelines, hyper-threading, DMA, etc. (c.f. lecture on computer architecture)
  - Different execution modes
    - Kernel mode: full instruction set
    - User mode: limited instruction set (e.g. no I/O instructions, memory protection)

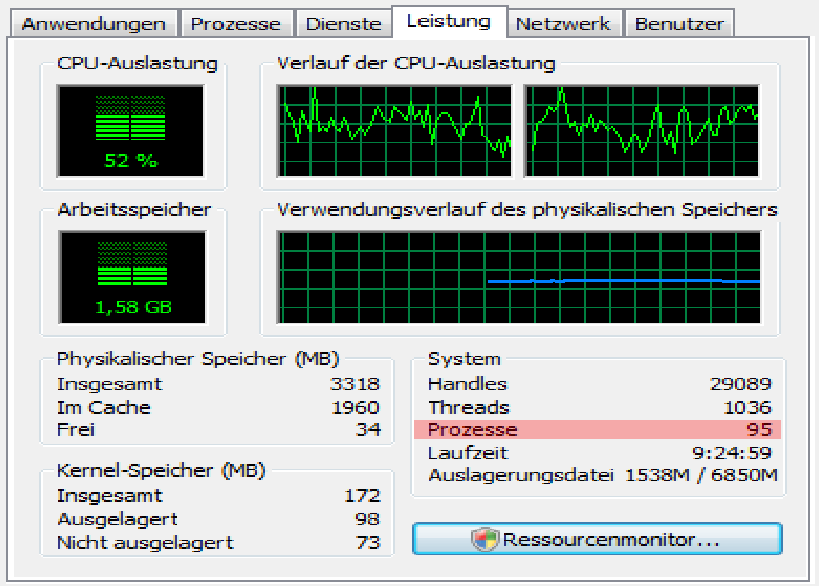
Technische Informatik

# The Process Concept

- **A process is an abstraction**
  - Provided by the operating system
  - For a **running instance of a program**
  - Enabling the “simultaneous“ execution of multiple sequentially running instances

- Conceptually, each process
  - Is executed sequentially
  - Has its own virtual processor
  - Runs independent from other processes

- Remarks
  - Historically, differentiation between job (batch) and tasks (interactive)
  - Typically less processors than processes
  - OS switches processes on processors (fast)

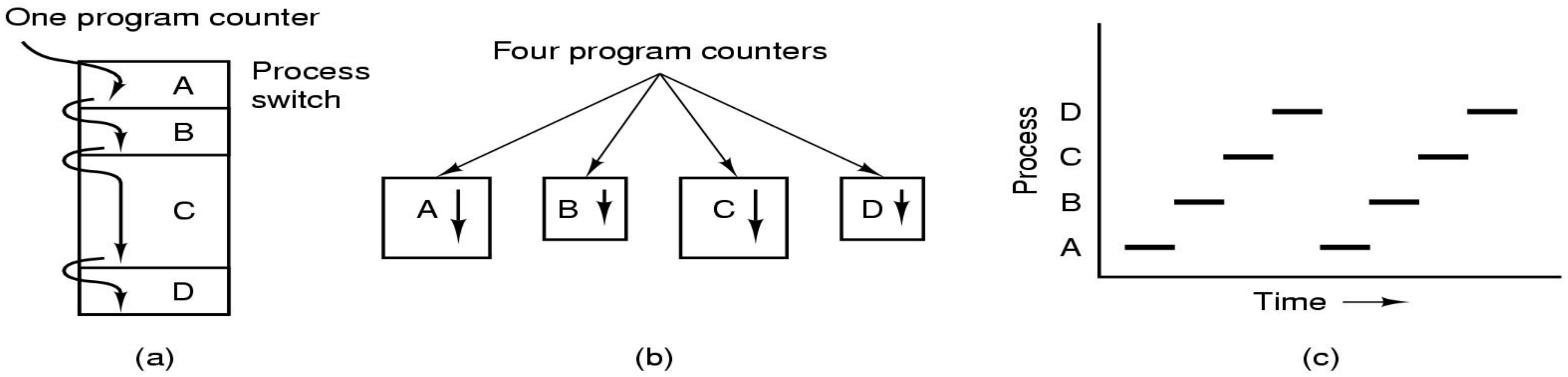



***2 processors***
***95 processes***


UNIVERSITÄT BONN

# The Process Model

[Tanenbaum01]




- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes **=> fig. (b)**
- Only one program active at any instant **=> fig. (a) + (c)**
- Fast switching by operating system creates illusion of parallelism

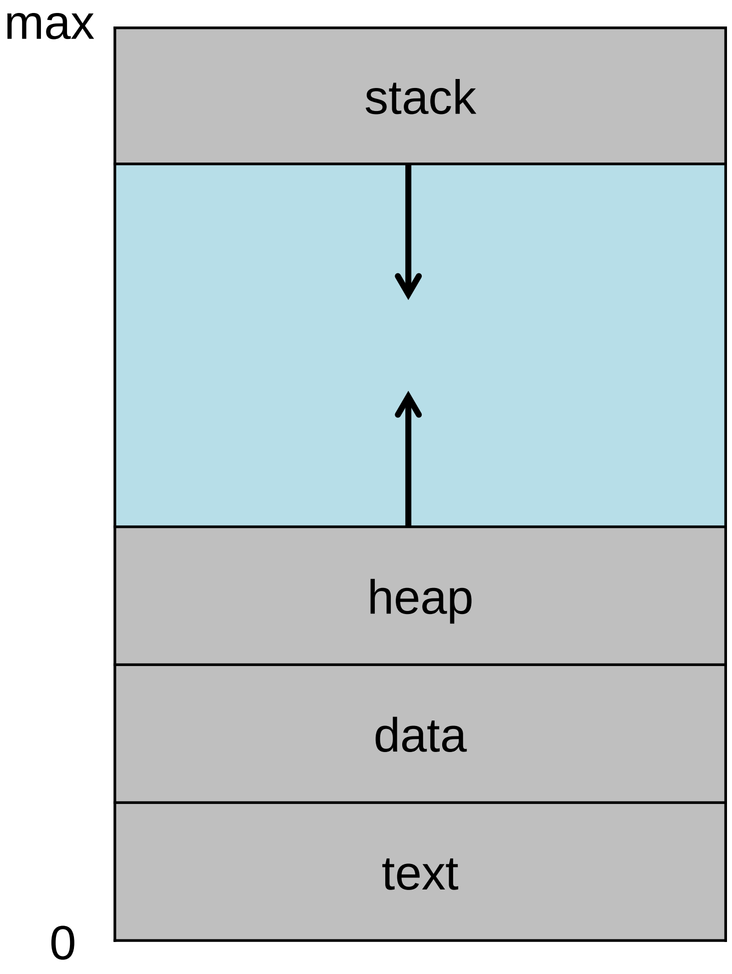
# Process in Memory

- A process in memory is more than program code

- The memory is limited and consists of sections for
  - Text (program code)
  - Data (global variables)
  - Stack (frames for unfinished procedure calls)
    - Parameters
    - Local variables
    - Return address
  - Heap (dynamically allocated data)

- Processes can only access their assigned memory
  - Processor runs in user mode
  - Memory segregation

[Silberschatz04]



# System Calls

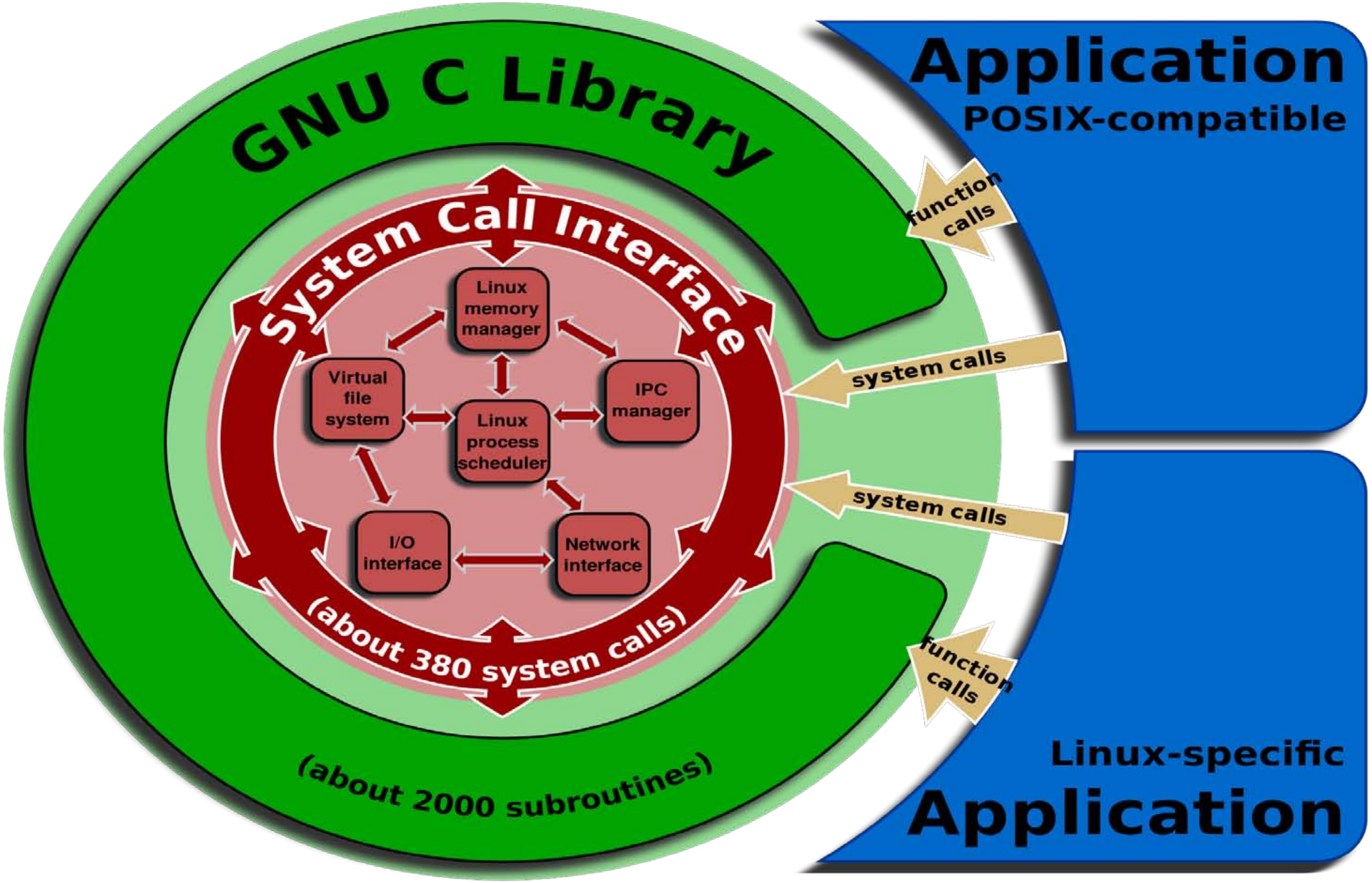
- Most operating systems (e.g. Linux and Windows) distinguish between **kernel-mode** and **user-mode**

- Kernel-mode processes (privileged) can access everything
  - Drivers
  - Kernel modules

- User-mode processes are only allowed to read their own memory

- **System-Calls** are the interface to the OS
  - Reading / writing Files, Sockets, etc.
  - Inter-Process Communication (IPC)
  - read from the C standard library `sys_read` internally

# System Calls

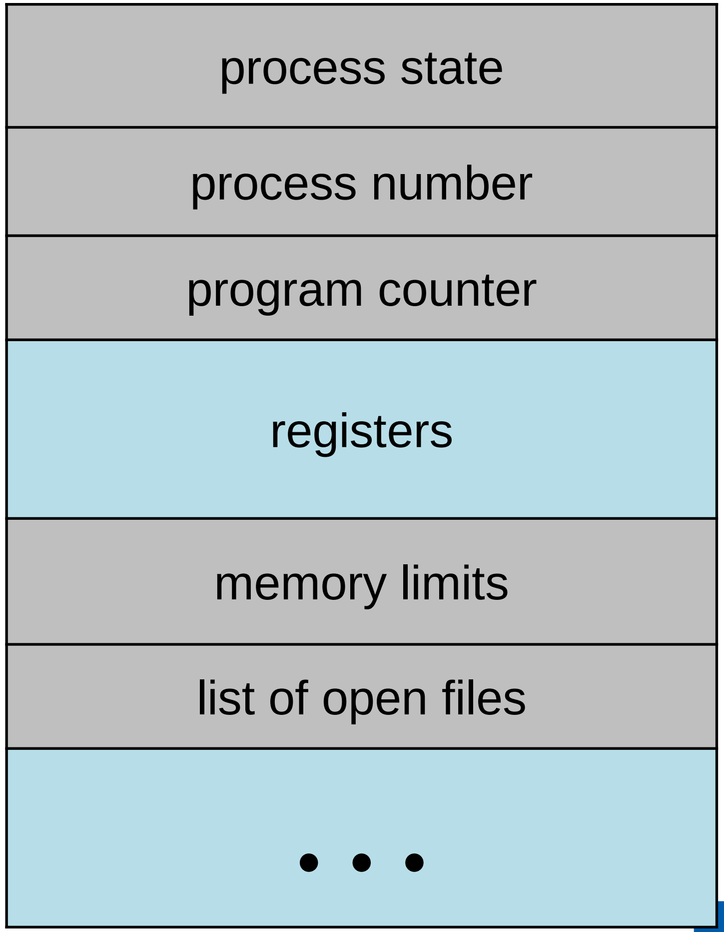



(Source: Wikipedia)

# Process Control Block (PCB)

- Operating system must associate more information with each process
- This information is stored within a process control block (PCB)
  - Process state (more details follow)
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

[Silberschatz04]



# Context Switch

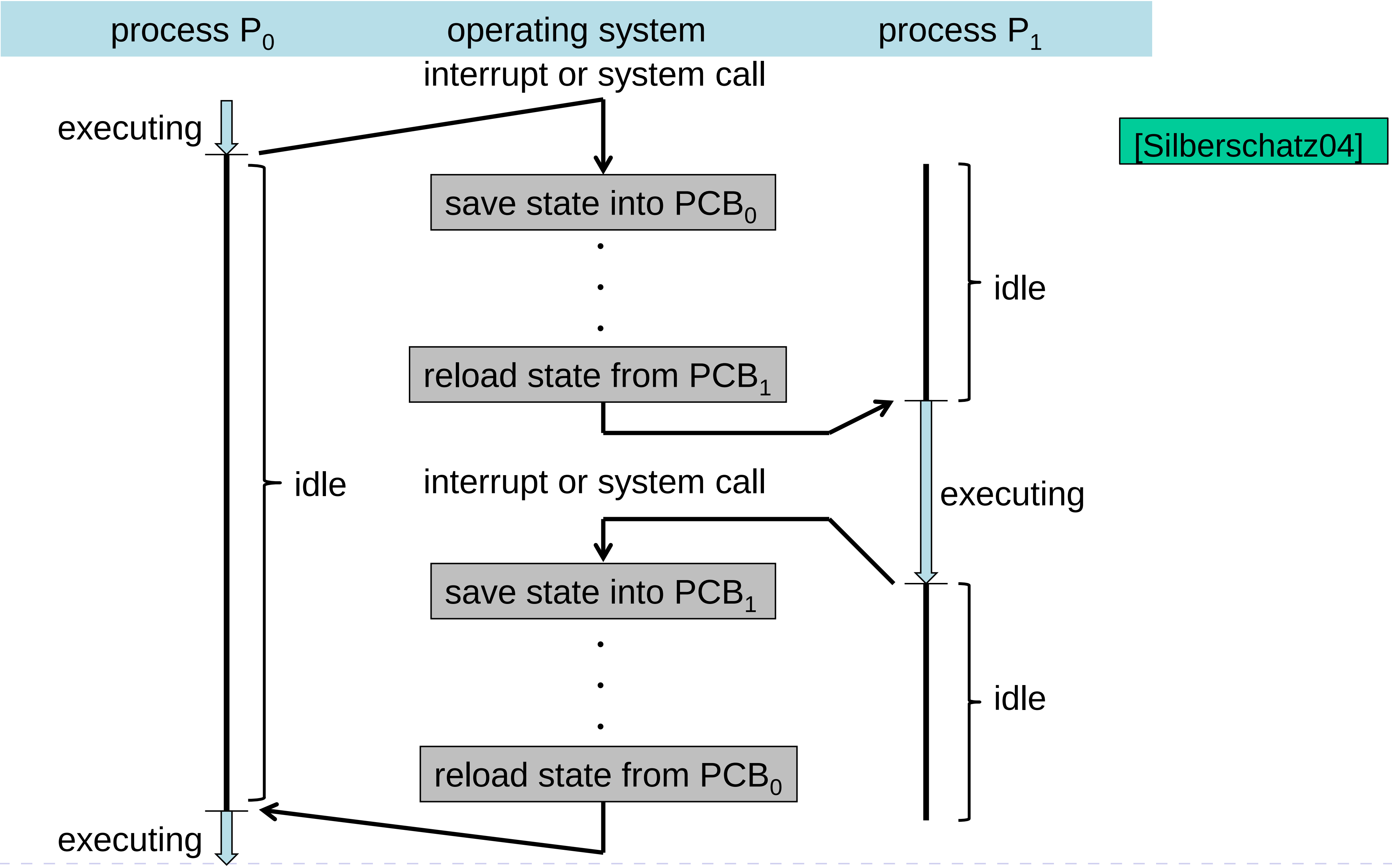
- Switching the processor from one process to another is commonly referred to as **context switch**
  - OS must save the state of the old process
  - OS must load the state for the new process

- **Context** of a process represented in the **PCB**

- For a context switch the **OS must be in control**
  - **System call** – process calls operating system procedure
  - **Interrupt handler** – device controller signals interrupt

- Context-switch time is overhead
  - The system does no useful work while switching
  - Time dependent on hardware support

# Switch From Process to Process

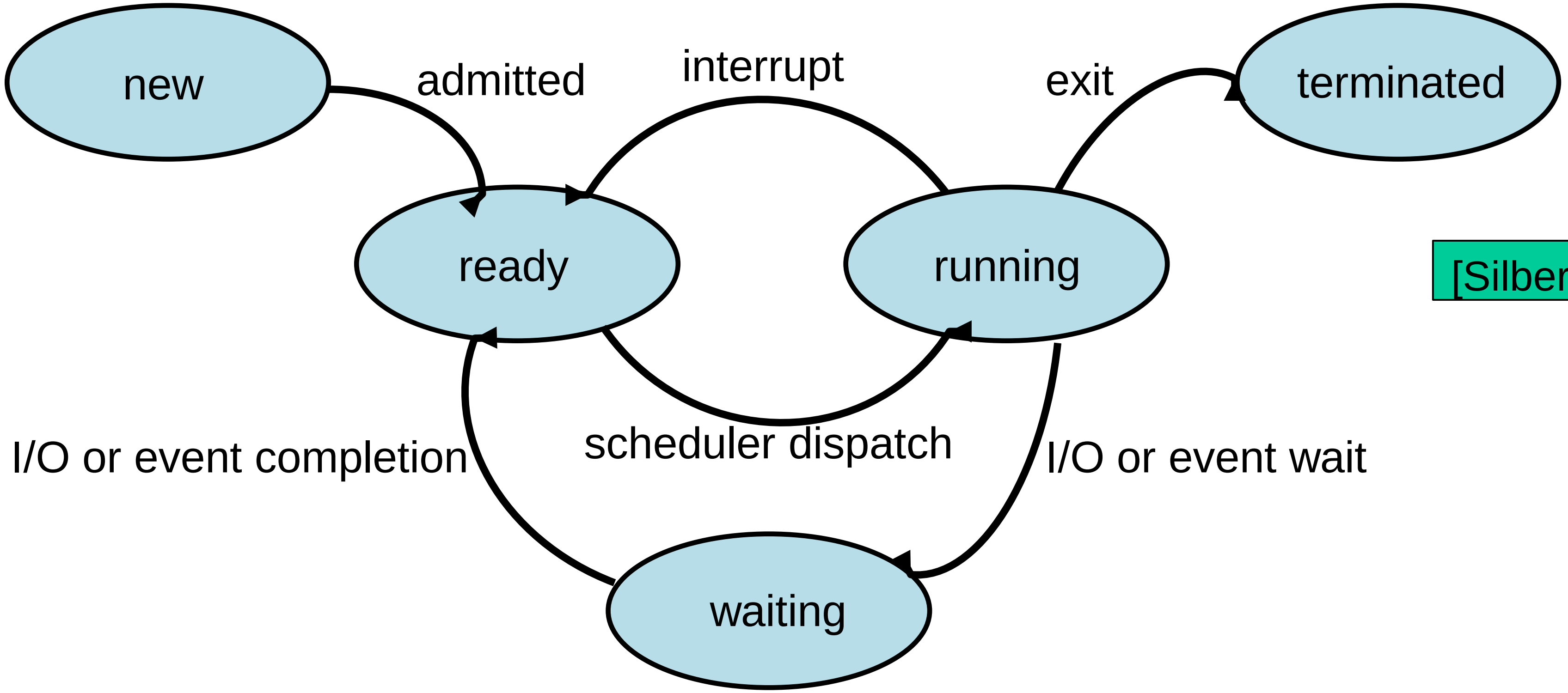





UNIVERSITÄT BONN

# Process State

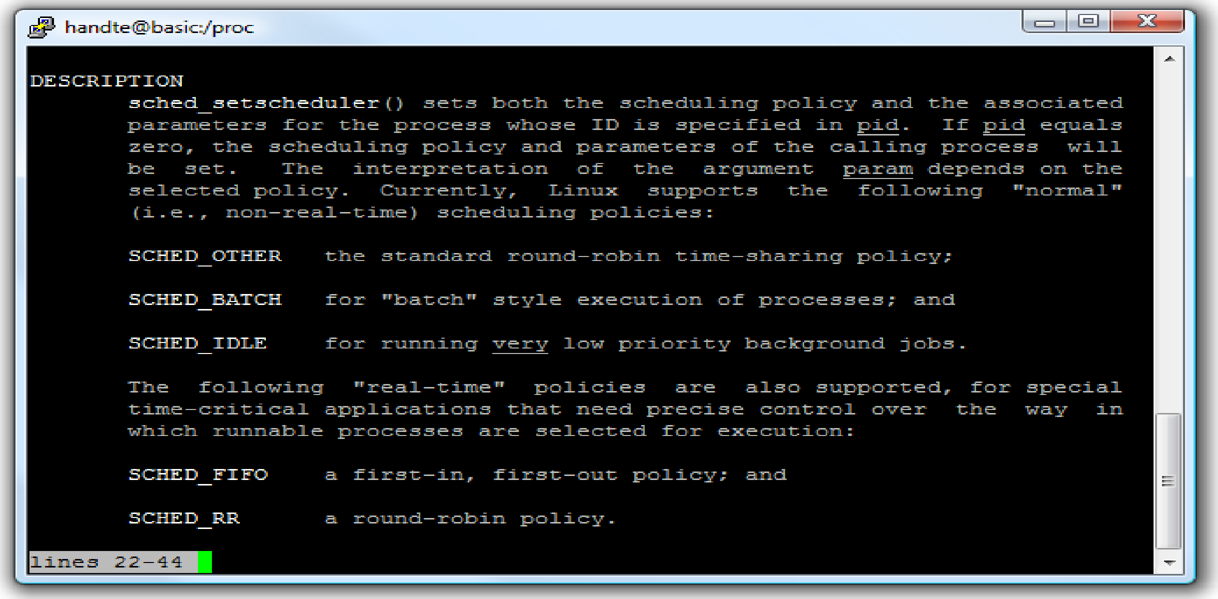
- As a process executes, it changes *state*
  - **new:** The process is being created
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **ready:** The process is waiting to be assigned to a processor
  - **terminated:** The process has finished execution




[Silberschatz04]

# Scheduling

see „man sched_setscheduler“




*Linux Scheduling Policies*

- **Before context switch OS decides which process to run (scheduling)**
  - Waiting processes cannot run but multiple processes might be ready

- **Scheduling goals may differ**
  - Batch system – high throughput
  - Interactive system – low latency
  - Embedded system – meet real-time deadlines

  => Different scheduling algorithms/policies to achieve different goals

- **Scheduler models**
  - Cooperative – Processes give up control by system call (yield)
  - Preemptive – Process may be stopped by system (e.g. upon timer interrupt)

# Observations

- Code of a process is guaranteed to be executed **sequentially**
- Memory used by processes is **isolated** by OS
- **Feeling of parallel** execution is achieved by (fast) switching
- Depending on scheduler model, processes may be switched at any point in time during the execution
  - Execution duration of programs differs across multiple runs
  - Programmer cannot count on the switching time
  - Process implementation cannot count on real world timing of individual instructions
- Simplifies development of most types of applications but may complicate development of programs with real-time constraints

# Process Creation in UNIX

- In most systems **processes are created dynamically**
    - Possible exceptions are embedded systems with fixed processes

- Process identified and **managed via a process identifier** (`pid`)

- Exemplary reasons for process creation
    - System initialization
    - User requests to execute a program
    - Batch job started by operating system
    - Running process starts another one

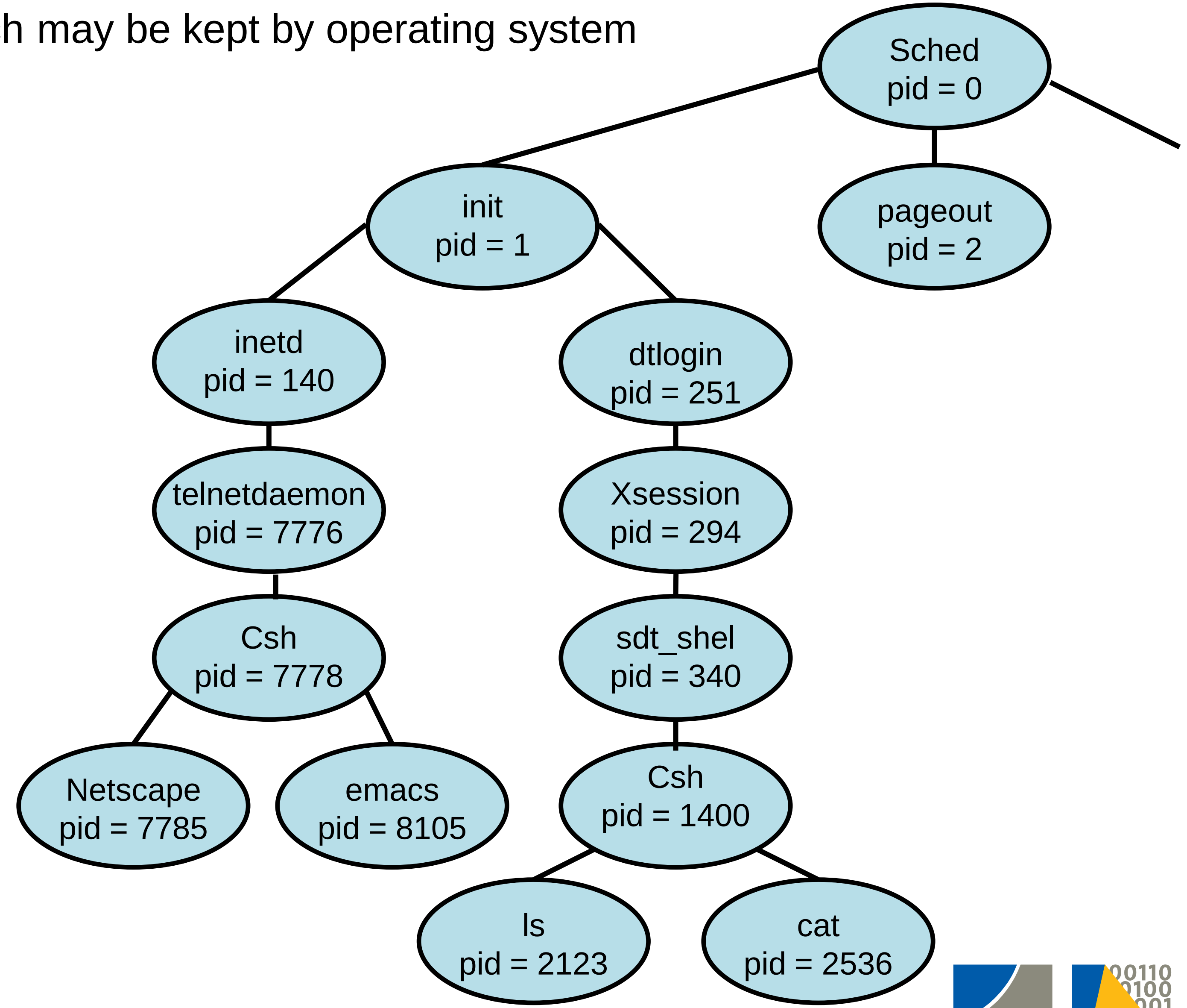
=> Technically, **all processes are created by another process**

- Parent processes create children processes
- Conceptually forming a tree of processes

# Process Hierarchies

[Silberschatz04]

- Parent/child relationship forms a **hierarchy** which may be kept by operating system
- **UNIX:**
  - Keeps the relationship
  - Referred to as "process group“
  - Child may get `pid` of parent (`ppid`)
- **Windows:**
  - No concept of hierarchy
  - All processes are created equal



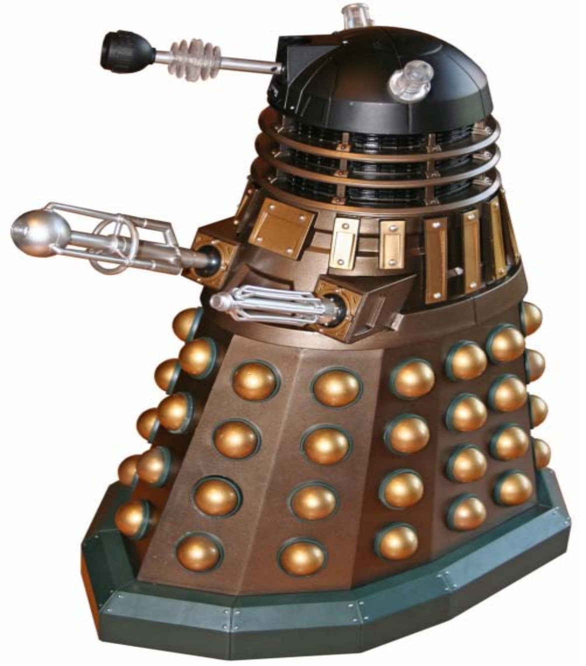

***Exemplary process hierarchy on Solaris***

# Process Creation Options

- Operating system may support various options with respect to:
  - **Resource sharing**
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources
  - **Execution**
    - Parent and children execute concurrently
    - Parent waits until children terminate
  - **Address space**
    - Child duplicate of parent
    - Child has a program loaded into it

# Process Termination

- Processes may terminate for several reasons
  - **Process executes last statement and asks operating system to delete it** `(exit)`
    - Process indicates `success/failure` via return value
    - Output data transmitted from child to parent (via `wait`)
    - Process' resources are deallocated by operating system
  - **Parent process may terminate execution of child** `(abort)`
    - If child has exceeded allocated resources
    - If task assigned to child is no longer needed
    - If parent is exiting: some operating systems terminate all children upon parent termination (cascading termination)





UNIVERSITÄT BONN

# Process Termination (cont'd)

- Processes may also be terminated by other entities

- **Terminated involuntarily by operating system** `(fatal exit)`
  - Process exceeded memory limitations
  - Program code executes illegal instruction
  - Program code performs division by 0

  => Some conditions might be handled by process without exit

- **Terminated upon request by some other process**
  - E.g. terminated by a task manager program
  - Requires adequate privileges

# Outline

- **2.1.1. Overview**
  - Motivation
  - Concept
  - Implementation
  - Scheduling
  - Basic operations
- **2.1.2. Utilization**
  - Examples (Linux)
- 2.1.3. Summary

# 2.1.2. Utilization – Processes on Linux

- Status
- Creation
- Signals
- Termination
- Tools

# Process Status in C

- Processes in Linux are arranged as a tree rooted at the `init` process

- Each process in Linux has
  - a process identifier (`pid`) to identify the process and
  - a parent process identifier (`ppid`) to identify its parent

- To obtain a process's `pid` and `ppid`, use the following system calls:

```c
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

# C Code Example (getpid(), getppid())

[Mitchell01]

- Exemplary program code that prints the process and parent process id:

```
int main ()
{
    printf(“The process id is %d\n“, (int) getpid());
    printf(“The parent process id is %d\n”, (int) getppid());
    return 0;
}
```

*(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)*

Anm.:
Alle Code-Ausschnitte mit Kennzeichen [Mitchell01] sind online verfügbar, s. Folie 3

# Process Creation

- Two classic ways to create a process in Linux:
  - Using the `system()` function to execute a command
  - Using `fork()` and `exec()` system calls

- Using the `system()` function to execute a command
  - `system()` creates a sub process to run the standard shell (`/bin/sh`), which then executes the command
  - `system()` returns the exit status of the shell command
    - 127, if the shell fails to run
    - -1, other errors

# Code Example (system)

[Mitchell01]

- Exemplary program code that invokes “`ls -l`“ on the root directory

```
#include <stdlib.h>

int main()
{
    int return_value;
    return_value = system(“ls -1 /”);
    return return_value;
}
```

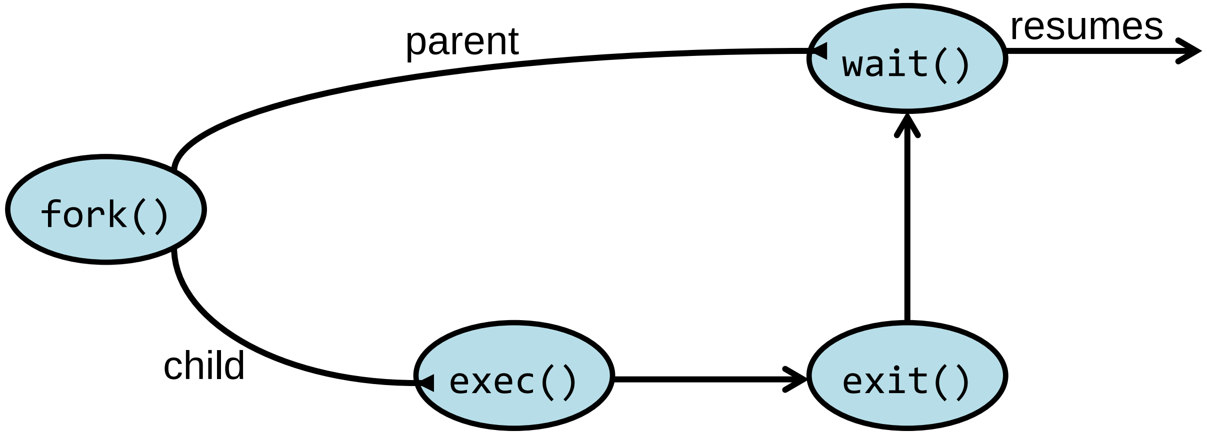
*(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)*

# Process Creation (2)

- Using `fork()` and `exec()` system calls
  - `fork()` creates a child process with the returned values
    - child process's pid in the parent process, and
    - zero in the child process
    - In the past: child process basically a clone of parent
    - Nowadays: Copy-on-write for performance

[Silberschatz04]

- `exec()` executes the sub program
  - current process is replaced with the new process
- `exit()` signals normal completion
- `abort()` signals abnormal completion
- `wait()` awaits the process completion



# Process Creation (3)

- Several variants of `exec()`
  - List of arguments specified as null-terminated strings, last argument must be null pointer
    - `int execl(const char *path, const char *arg, ...);`
    - `int execlp(const char *file, const char *arg, ...);`
    - `int execle(const char *path, const char *arg, ..., char * const envp[]);`

- List of arguments specified as array of null-terminated strings
  - `int execv(const char *path, char *const argv[]);`
  - `int execvp(const char *file, char *const argv[]);`

- `execlp` and `execvp` will search for executable in `PATH` if leading / not specified

- `execle` enables specification of environment

„l" = list, „v" = vector (either one to be used)

„e" = environment (to be spec.)

„p" = search in PATH

# Code Example (fork, exec, exit)

[Silberschatz04]

- Exemplary program code that invokes “ls“ and waits for child

```
int main()
{
    pid_t pid;

    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error ocurred*/
    fprintf(stderr, “Fork Failed”);
    exit(-1);
    }
    else if (pid == 0) { /* child process */
    execlp(“/bin/ls”, “ls”, NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf(“Child Complete”);
    exit(0);
    }
}
```

# C Code Example (fork, exec, abort)

- Exemplary program code that invokes “ls -l“ on the root directory

[Mitchell01]

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

/*  Spawn a child process running a new program. PROGRAM is the name of the
    program to run; the path will be searched for this program. ARG_LIST is
    a NULL-terminated list of character strings to be passed as the
    program’s argument list. Returns the process ID of the spawned process.
    */

int spawn (char *program, char **arg_list)
{
    pid_t child_pid;

    /* Duplicate this process */
    child_pid = fork();
    if (child_pid != 0)
    /* This is the parent process. */
    return child_pid;
    else {
    /* Now execute PROGRAM, searching for it in the path. */
    execvp(program, arg_list);
    /* The execvp function returns only if an error occurs. */
    fprintf(stderr, “an error occurred in execvp\n”);
    abort();
    }
}
```

```
int main ()
{
    /* The argument list to pass to the “ls” command. */
    char *arg_list[] = {
    “ls”,  /* argv[0], the name of the program. */
    “-l”,
    “/”,
    NULL   /* The argument list must end with a NULL. */
    };

    /* Spawn a child process running the “ls” command.
    Ignore returned child process ID. */
    spawn(“ls”, arg_list);

    printf(“done with main program\n”);

    return 0;
}
```

```
int spawn (char* program, char** arg_list)
```

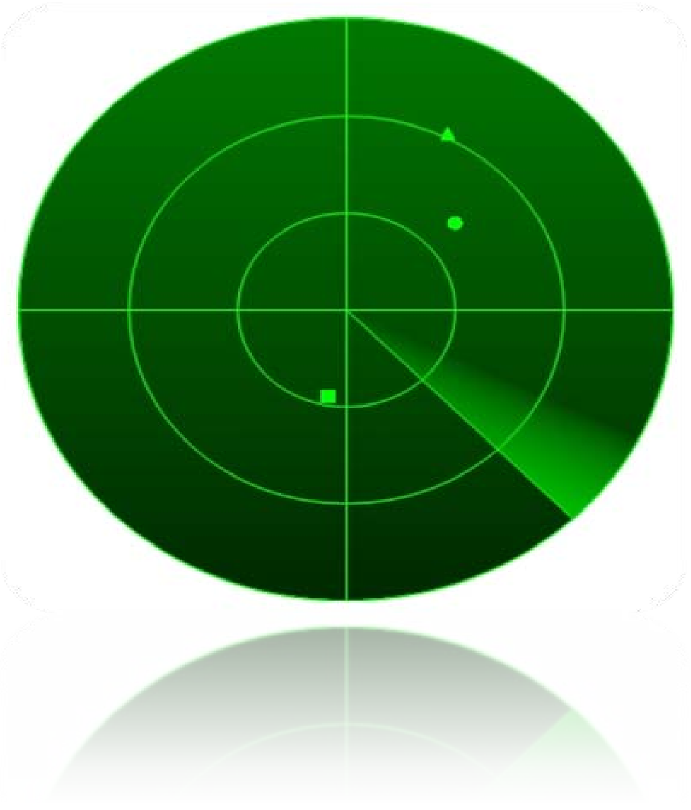
*(Source: Advanced Linux Programming by CodeSourcery LLC, published by  New Riders Publishing)*

# Signals

- Signals are:
  - mechanisms from communicating and manipulating processes
  - special messages sent to a process
  - asynchronous

- Normally, when a process receives a signal, it
  - first stops the current execution,
  - handles the signal, and then
  - resumes the original execution.

- Each signal type is specified by a signal number
- For most signal types, a program can respond to the signal by registering a signal-handler function



# Signal Types

- Linux defines several types of signals and default actions
  - Some examples are (according to POSIX.1-1990 standard)

| Signal | Value | Action | Comment |
|---|---|---|---|
| SIGHUP | 1 | Term | Death of controlling process |
| SIGINT | 2 | Term | Interrupt from Keyboard |
| SIGQUIT | 3 | Core | Quit from Keyboard |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort |
| SIGFPE | 8 | Core | Floating point exception |
| SIGKILL | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm |
| SIGTERM | 15 | Term | Termination signal |
| SIGUSR1 | 30, 10, 16 | Term | User-defined signal 1 |
| SIGUSR2 | 31, 12, 17 | Term | User-defined signal 2 |
| SIGCHLD | 20, 17, 18 | Ign | Child stopped or terminated |
| SIGCONT | 19, 18, 25 | Cont | Continue if stopped |
| SIGSTOP | 17, 19, 23 | Stop | Stop process |
| SIGTSTP | 18, 20, 24 | Stop | Stop typed at tty |
| SIGTTIN | 21, 21, 26 | Stop | tty input for background process |
| SIGTTOU | 22, 22, 27 | Stop | tty output for background process |

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

**Signal SIGIO**
(cf. chapter 1)
has value 29

(special case, not POSIX, may vary with different OSs)

# Code Example (signal handler)

[Mitchell01]

- Exemplary program code that registers handler to count the number of received SIGUSR1 signals

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

sig_atomic_t sigusrl_count = 0;                 (ATOMIC)
void handler(int signal_number)
{
    ++sigusrl_count;
}

int()
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &handler;
    sigaction(SIGUSR1, &sa, NULL);

    /* Do some lengthy stuff here. */
    /* ... */

    printf("SIGUSR1 was raised %d times\n", sigusr1_count);
    return 0;
}
```

*(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)*

# Process Termination

- Wait for process termination in C
  - Parent process may want to retrieve the termination status of its child processes
  - `wait()` function is used to obtain the exit code (integer)
  - `WIFEXITED` macro is used to determine a child process's exit status
  - `WEXITSTATUS` macro can be used to extract the exit code



UNIVERSITÄT BONN

# Code Example (wait)

- Exemplary program code that uses macros to wait for child

[Mitchell01]

```
int main()
{
    int child_status;
    /* The argument list to pass to the "ls" command. */
    char *arg_list[] = {
    "ls",                     /* argv[0], the name of the program. */
    "-1",
    "/",
    NULL             /* The argument list must end with a NULL. */
    };

    /* Spawn a child process running the "ls" command. Ignore the returned child process ID. */
    spawn("ls", arg_list);
    /* Wait for the child process to complete. */
    wait(&child_status);
    if (WIFEXITED(child_status))
    printf("The child process exited normally, with exit code %d\n",
    WEXITSTATUS(child_status));
    else
    printf("The child process exited abnormally\n");
    return 0;
}
```

*(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)*

# Zombies

[Mitchell01]

- A zombie process is a child process that has terminated but its parent process has not called wait()
- Exemplary code for creating a zombie process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t child_pid;

    /* Create a child process. */
    child_pid = fork();
    if(child_pid > 0) {
    /* This is the parent process. Sleep for a minute. */
    sleep(60);
    }
    else {
    /* This is the child process. Exit immediately. */
    exit(0);
    }
    return 0;
}
```

*(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)*

# Zombies (2)

- If parent process calls `wait()` before the child process terminates,
  - the parent process blocks until the child process terminates

- If child process has terminated,
  - it terminates with its exit status code if its parent process has called `wait()`
  - it becomes a zombie process if its parent process has not called `wait()`
  - it is cleaned up by the `init` process if its parent process exits without calling `wait()`

# Asynchronous Cleanup

- A parent process may want to continue to work without being blocked by `wait()`
- One approach is to periodically call `wait3()` or `wait4()` with non-blocking mode
- Another approach is to use the signal **SIGCHLD**
  - Parent process is notified when a child process terminates
  - Parent specifies a signal handler, which stores the child process’s termination status, and then cleans up the child

# Code Example (SIGCHLD)

[Mitchell01]

- Exemplary program code that registers handler to clean up child

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

sig_atomic_t child_exit_status;

void clean_up_child_process(int signal_number)
{
    /* Clean up the child process. */
    int status;
    wait(&status);
    /* Store its exit status in a global variable. */
    child_exit_status = status;
}

int main()
{
    /* Handle SIGCHLD by calling clean_up_child_process. */
    struct sigaction sigchld_action;
    memset(&sigchld_action, 0, sizeof(sigchld_action));
    sigchld_action.sa_handler = &clean_up_child_process;
    sigaction(SIGCHLD, &sigchld_action, NULL);

    /* Now do things, including forking a child process. */
    /* ... */

    return 0;
}
```

*(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders Publishing)*

# Process Tools in Linux

- **View running processes**
  - `ps, pstree, top`

- **Send signals to processes**
  - `kill`

- **Modify process execution**
  - `nice / renice`
  - `& / <ctrl+z> / fg / bg`



UNIVERSITÄT BONN

# Ps

- ps – report a snapshot of the current processes

```
handte@basic:~
handte@basic ~ $ ps
  PID TTY          TIME CMD
11034 pts/0    00:00:00 bash
11105 pts/0    00:00:00 ps
handte@basic ~ $ ps -Al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -   931 -      ?        00:00:01 init
5 S     0     2     0  0  75  -5 -     0 kthrea ?        00:00:00 kthreadd
1 S     0     3     2  0 -40   - -     0 migrat ?        00:00:00 migration/0
1 S     0     4     2  0  75  -5 -     0 ksofti ?        00:00:00 ksoftirqd/0
5 S     0     5     2  0 -40   - -     0 watchd ?        00:00:01 watchdog/0
1 S     0     6     2  0 -40   - -     0 migrat ?        00:00:00 migration/1
1 S     0     7     2  0  75  -5 -     0 ksofti ?        00:00:00 ksoftirqd/1
5 S     0     8     2  0 -40   - -     0 watchd ?        00:00:00 watchdog/1
1 S     0     9     2  0 -40   - -     0 migrat ?        00:00:00 migration/2
1 S     0    10     2  0  75  -5 -     0 ksofti ?        00:00:00 ksoftirqd/2
5 S     0    11     2  0 -40   - -     0 watchd ?        00:00:00 watchdog/2
1 S     0    12     2  0 -40   - -     0 migrat ?        00:00:00 migration/3
1 S     0    13     2  0  75  -5 -     0 ksofti ?        00:00:00 ksoftirqd/3
5 S     0    14     2  0 -40   - -     0 watchd ?        00:00:00 watchdog/3
1 S     0    15     2  0 -40   - -     0 migrat ?        00:00:00 migration/4
1 S     0    16     2  0  75  -5 -     0 ksofti ?        00:00:00 ksoftirqd/4
5 S     0    17     2  0 -40   - -     0 watchd ?        00:00:00 watchdog/4
1 S     0    18     2  0 -40   - -     0 migrat ?        00:00:00 migration/5
1 S     0    19     2  0  75  -5 -     0 ksofti ?        00:00:00 ksoftirqd/5
5 S     0    20     2  0 -40   - -     0 watchd ?        00:00:00 watchdog/5
```

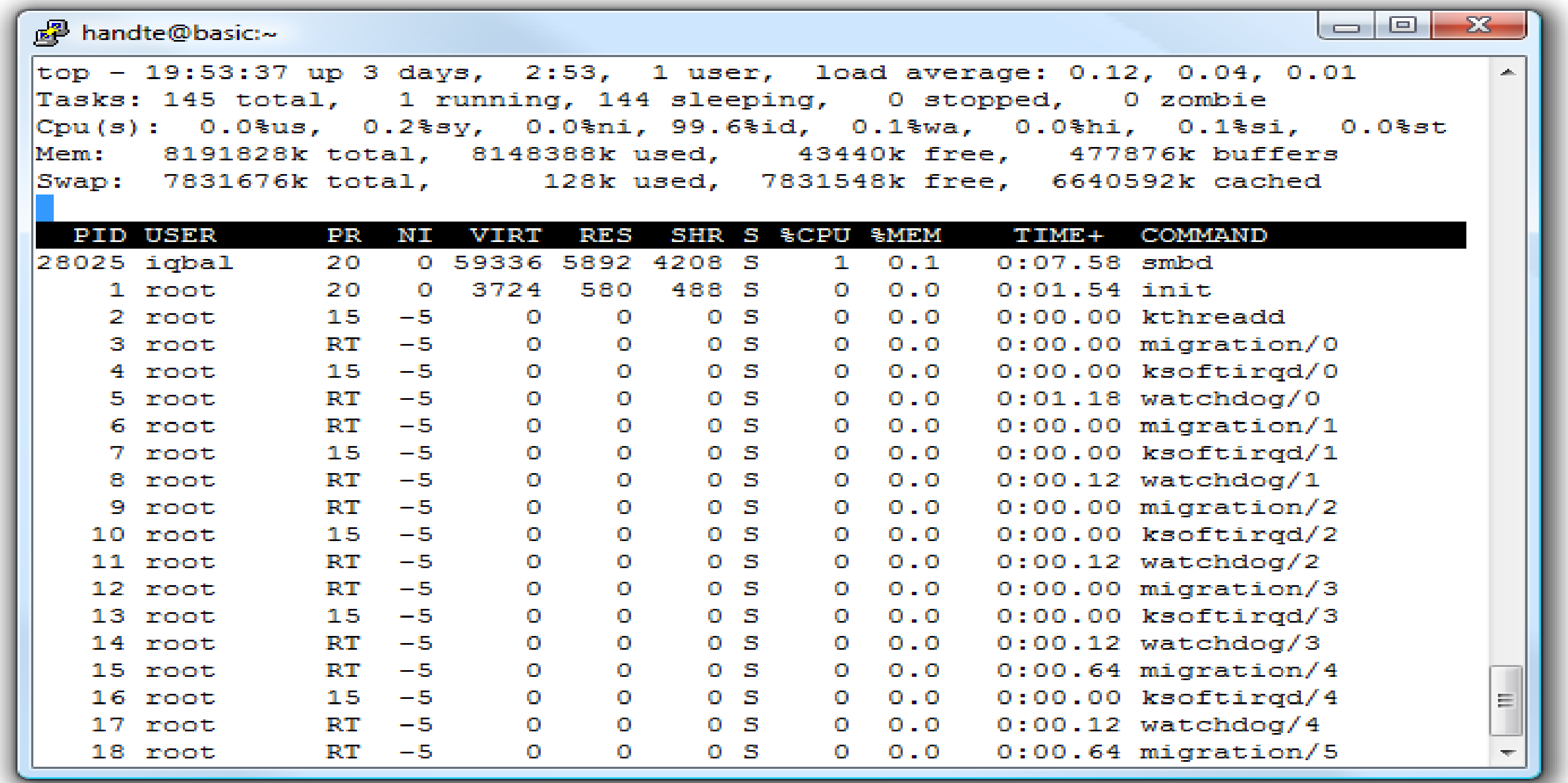
# Pstree

- `pstree` – displays a tree of running processes

```
handte@basic ~ $ pstree
init-+-6*[agetty]
     |-apache2-+-apache2
     |         `-2*[apache2---26*[{apache2}]]
     |-apcupsd
     |-cron
     |-cupsd
     |-dhcpd
     |-dnsmasq
     |-mysqld---53*[{mysqld}]
     |-nmbd
     |-openvpn
     |-pdns_server-+-pdns_server---7*[{pdns_server}]
     |             `-{pdns_server}
     |-portmap
     |-radiusd---5*[{radiusd}]
     |-rpc.mountd
     |-rpc.statd
     |-rsync
     |-runscript.sh---dsmc---{dsmc}
     |-slapd---6*[{slapd}]
     |-smbd---7*[smbd]
     |-sshd---sshd---sshd---bash---pstree
     |-syslog-ng
     `-udevd
handte@basic ~ $
```

# Top

- `top` – displays processes (table can be rearranged interactively)
- `htop` – advanced top



handte@basic:~

```
top - 19:53:37 up 3 days,  2:53,  1 user,  load average: 0.12, 0.04, 0.01
Tasks: 145 total,   1 running, 144 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.2%sy,  0.0%ni, 99.6%id,  0.1%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:   8191828k total,  8148388k used,    43440k free,   477876k buffers
Swap:  7831676k total,      128k used,  7831548k free,  6640592k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28025 iqbal     20   0 59336 5892 4208 S    1  0.1   0:07.58 smbd
    1 root      20   0  3724  580  488 S    0  0.0   0:01.54 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S    0  0.0   0:00.00 migration/0
    4 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    5 root      RT  -5     0    0    0 S    0  0.0   0:01.18 watchdog/0
    6 root      RT  -5     0    0    0 S    0  0.0   0:00.00 migration/1
    7 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    8 root      RT  -5     0    0    0 S    0  0.0   0:00.12 watchdog/1
    9 root      RT  -5     0    0    0 S    0  0.0   0:00.00 migration/2
   10 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksoftirqd/2
   11 root      RT  -5     0    0    0 S    0  0.0   0:00.12 watchdog/2
   12 root      RT  -5     0    0    0 S    0  0.0   0:00.00 migration/3
   13 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksoftirqd/3
   14 root      RT  -5     0    0    0 S    0  0.0   0:00.12 watchdog/3
   15 root      RT  -5     0    0    0 S    0  0.0   0:00.64 migration/4
   16 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksoftirqd/4
   17 root      RT  -5     0    0    0 S    0  0.0   0:00.12 watchdog/4
   18 root      RT  -5     0    0    0 S    0  0.0   0:00.64 migration/5
```

# Kill

- `kill` – send signals to a process (defaults to SIGTERM)

```
handte@basic:~
handte@basic ~ $ cat runforever.sh
#!/bin/bash
for ((;;)) do sleep 1; done

handte@basic ~ $ ./runforever.sh &
[1] 835
handte@basic ~ $ ps
  PID TTY          TIME CMD
  835 pts/0    00:00:00 runforever.sh
  840 pts/0    00:00:00 sleep
  841 pts/0    00:00:00 ps
11034 pts/0    00:00:05 bash
handte@basic ~ $ kill 835
handte@basic ~ $
[1]+  Beendet                 ./runforever.sh
handte@basic ~ $ ps
  PID TTY          TIME CMD
  864 pts/0    00:00:00 ps
11034 pts/0    00:00:05 bash
handte@basic ~ $
```

# Nice / Renice

- `nice` – change scheduling priority upon startup
- `renice` – change scheduling priority of running process

handte@basic:~

```
handte@basic ~ $ nice -n 19 ./runforever.sh & # run lowest priority
[3] 19650
handte@basic ~ $ nice -n -20 ./runforever.sh & # run highest priority (requires
root privledges)
[4] 19822
handte@basic ~ $ nice: kann Priorität nicht setzen: Keine Berechtigung

[4]+  Exit 1                    nice -n -20 ./runforever.sh
handte@basic ~ $
```

root@basic:/home/user/handte

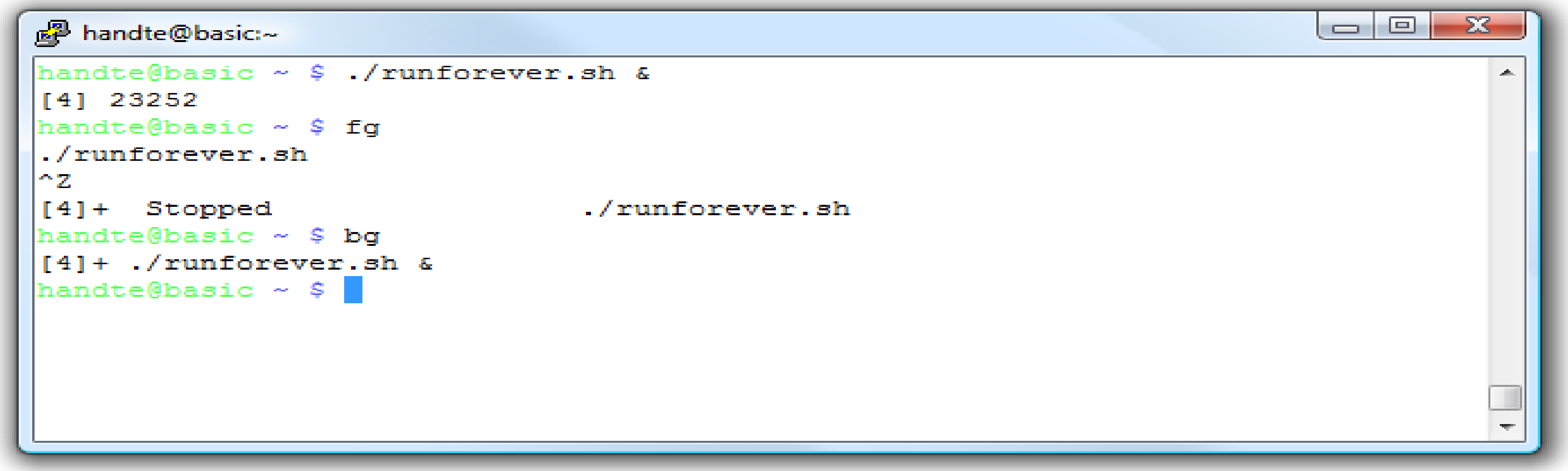
```
basic handte # nice -n -10 ./runforever.sh &
[1] 20614
basic handte # kill 20614
basic handte #
[1]+  Beendet                   nice -n -10 ./runforever.sh
basic handte #
```



UNIVERSITÄT BONN

# & / <Ctrl+Z> / Fg / Bg

- & – start process in background
- `<ctrl+z>` – stop process in foreground
- bg – move process to background
- `fg` - move process to foreground



# Outline

- **2.1.1. Overview**
  - Motivation
  - Concept
  - Implementation
  - Scheduling
  - Basic operations
- **2.1.2. Utilization**
  - Examples (Linux)
- **2.1.3. Summary**



UNIVERSITÄT BONN

# 2.1.3. Summary

- Processes are an abstraction of an executing program instance
  - Simplifies program development and processor usage
  - In C, programmers refrain from assumptions on the execution of
  - Multiple processes (switched in and out arbitrarily)
  - Timing of instructions within a process (blocked / switched out)
  - The OS maps processes on available processors
  - Scheduling can be optimized for system purposes

- Most modern operating systems
  - provide calls for process creation, manipulation and termination
  - hide the low-level details from the programmer
  - Creating space for data (registers, program counter, etc.)
  - Transparent storing and restoring of process context

# Limitations

- Process creation is not a light-weight undertaking

- Parallelism/Concurrency may be required even within single application
  - E.g. a web server processing multiple requests simultaneously

- Memory isolation provided by processes is not always necessary and may be more complicated to deal with
  - E.g. requires explicit data sharing between different processes

… IPC (inter process communication) in subsection 2.3.

- In many cases, threads can provide a more light-weight alternative …
  … but more to come in next subsection

… threads in subsection 2.2.



UNIVERSITÄT BONN