7th Slide Set
Operating Systems

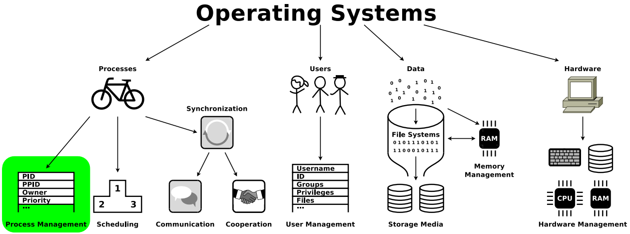
Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Faculty of Computer Science and Engineering
christianbaun@fb2.fra-uas.de

Learning Objectives of this Slide Set

- At the end of this slide set, you know/understand...
  - what a **process** is from an operating system perspective
  - what information the **process context** contains in detail
    - **User context**, **Hardware context**, **System context**
  - the different process states by discussing **process state models**
  - how **process management works** in detail with **process tables**, **process control blocks** and **status lists**
  - how **processes are created** and **erased**
  - the **structure of UNIX processes in memory**
  - what **system calls** are and how they work

Exercise sheet 7 repeats the contents of this slide set which are relevant for these learning objectives



Process and Process Context

We already know. . .

- A **process** (lat. *procedere* = proceed, move forward) is an instance of a program that is running
- Processes are dynamic objects and represent sequential activities in a computer system
- On computers multiple processes are executed all the time
- In multitasking mode, the CPU switches back and forth between the processes

- A process includes, in addition to the program code, its **context**
- 3 types of contextual information manages the operating system:
  - **User context**
    - Content of the allocated address space (virtual memory) $\Longrightarrow$ slide set 5
  - **Hardware context** ($\Longrightarrow$ slide 4)
    - CPU registers
  - **System context** ($\Longrightarrow$ slide 5)
    - Information, which is stored by the operating system about a process
- The operating system stores the information of the hardware context and system context in the **process control block** ($\Longrightarrow$ slide 6)

Hardware Context

- The **hardware context** is the content of the CPU registers during process execution
- Registers whose content needs to be backed up in the event of a process switch:
  - Program Counter (Instruction Pointer) – stores the memory address of the next instruction to be executed
  - Stack pointer – stores the address at the current end of the stack
  - Base pointer – points to an address in the stack
  - Instruction register – stores the instruction, which is currently executed
  - Accumulator – stores operands for the ALU and their results
  - Page-table base Register – stores the address of the page table of the running process
  - Page-table length register – stores the length of the page table of the running process

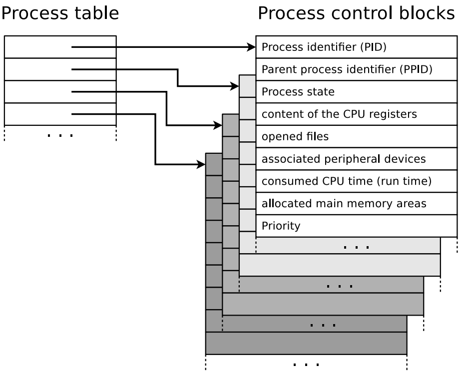
Some of these registers have been discussed in slide set 3 and slide set 5

System Context

- The **system context** is the information the operating system stores about a process
- Examples:
  - Record in the process table
  - Process ID (PID)
  - Process state
  - Information about parent or child processes
  - Priority
  - Identifiers - access credentials to resources
  - Quotas - allowed usage quantity of individual resources
  - Runtime
  - Opened files
  - Assigned devices

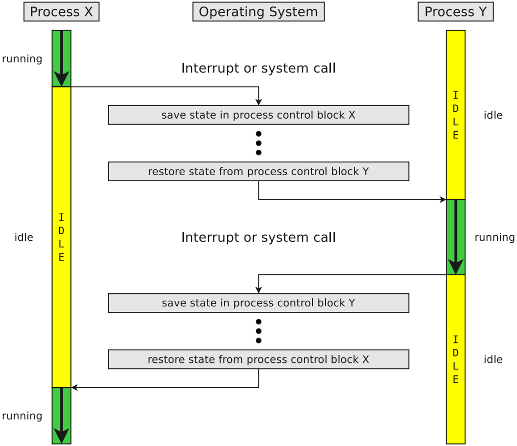
Process Table and Process Control Blocks

- Each process has its own process context, which is independent of the contexts of other processes
- For managing the processes, the operating system implements the **process table**
  - It is a list of all existing processes.
  - It contains for each process a record which is called **process control block**



Process Switching

- If the CPU is switched from one process to another one, the context ($\Longrightarrow$ CPU register content) of the running process is stored in the process control block
  - If a process gains access to the CPU, its context is restored by using the content of the process control block



- Each process is at any moment in a particular **state** $\Longrightarrow$ Process state models

Process States

We already know...

Every process is at any moment in a state

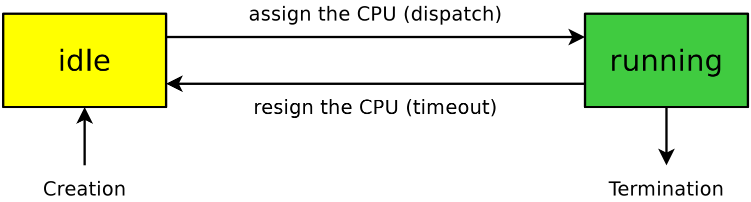
- The number of different states depends on the process state model of the operating system used

Question

How many process states must a process model contain at least?

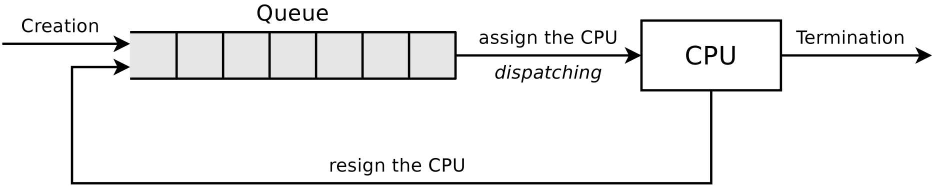
Process State Model with 2 States

- In principle, 2 process states are enough
    - **running**: The CPU is allocated to a process
    - **idle**: The processes waits for the allocation of CPU



Process State Model with 2 States (Implementation)

- The processes in **idle** state must be stored in a queue, in which they wait for execution
  - The list is sorted according to the process priority or waiting time



The priority (proportional computing power) in Linux has a value from -20 to +19 (in integer steps). -20 Is the highest priority and 19 is the lowest priority. The default priority is 0 Normal users can assign priorities from 0 to 19. The system administrator (`root`) can assign negative values too.

- This model also shows the working method of the **dispatcher**
  - The job of the dispatcher is to carry out the state transitions
- The execution order of the processes is specified by the **scheduler**, which uses a **scheduling algorithm** (see slide set 8)

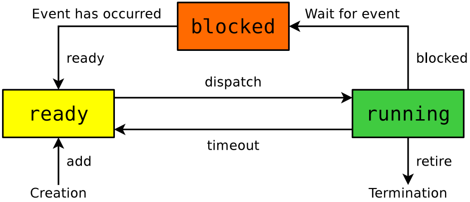
Conceptual Error of the Process State Model with 2 States

- The process state model with 2 states assumes that all processes are ready to run at any time
  - This is unrealistic!
- Almost always do processes exist, which are **blocked**
  - Possible reasons:
    - They wait for the input or output of an I/O device
    - They wait for the result of another process
    - They wait for a user reaction (interaction)
- Solution: The idle processes can be categorized into 2 groups
  - Processes, which are **ready**
  - Processes, which are **blocked**

$\Longrightarrow$ Process state model with 3 states

Process State Model with 3 States

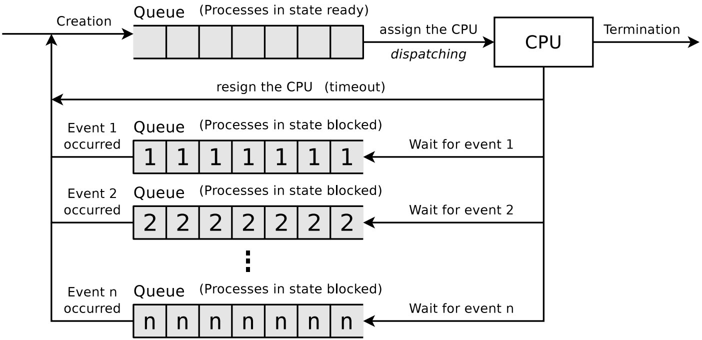
- Each process is in one of the following states:
- **running**:
  - The CPU is assigned to the process and executes its instructions



- **ready**:
  - The process could immediately execute its instructions on the CPU and it is currently waiting for the allocation of the CPU
- **blocked**:
  - The process cannot currently be executed and is waiting for the occurrence of an event or the satisfaction of a condition
  - This may be e.g. a message from another process or an input/output device or the occurrence of a synchronization event

Process State Model with 3 States – Implementation

- In practice, operating systems (e.g. Linux) implement multiple queues for processes in **blocked** state

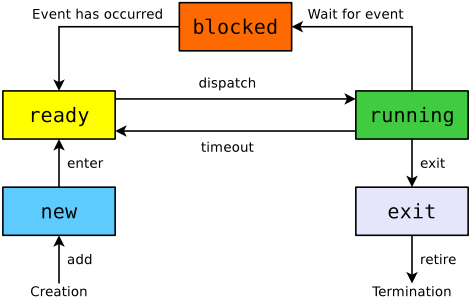


- During state transition, the process control block of the affected process is removed from the old status list and inserted into the new status list
- No separate list exists for processes in **running** state

Process State Model with 5 States

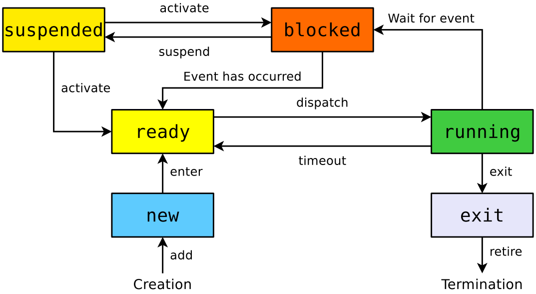
- It makes sense to expand the process state model with 3 states by 2 further process states
  - **new**: The process (process control block) has been created by the operating system but the process is not yet added to the queue of processes in **ready** state
  - **exit**: The execution of the process has finished or was terminated, but for various reasons the process control block still exists

- Reason for the existence of the process states **new** and **exit**:
  - On some systems, the number of executable processes is limited in order to save memory and to specify the degree of multitasking



Process State Model with 6 States

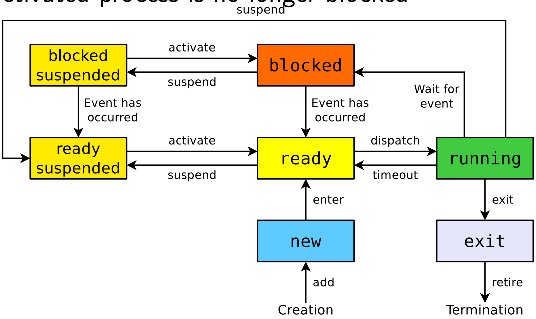
- If not enough physical main memory capacity exists for all processes, parts of processes must be swapped out $\Longrightarrow$ **swapping**
- The operating system outsources processes, which are in **blocked** state
- As a result, more main memory capacity is available for the processes in the states **running** and **ready**
  - Therefore, it makes sense to extend the process state model with 5 states, with a further process state **suspended**



Process State Model with 7 States

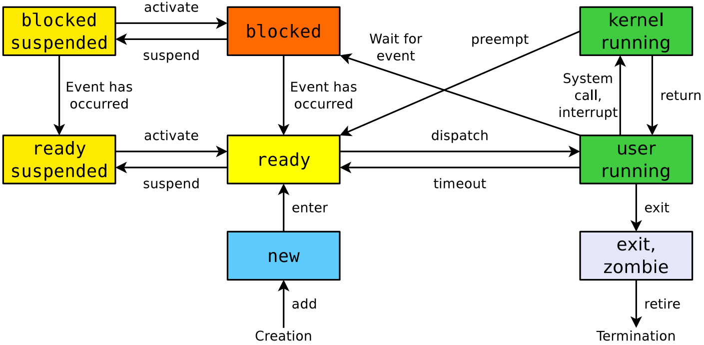
- If a process has been suspended, it is better to use the freed up space in main memory to activate an outsourced process instead of assigning it to a new process
  - This is only useful if the activated process is no longer blocked

- The process state model with 6 states lacks the ability to classify the suspended processes into:
  - blocked suspended processes
  - ready suspended processes



Process State Model of Linux/UNIX (somewhat simplified)

- The state **running** is split into the states...
  - **user running** for user mode processes
  - **kernel running** for kernel mode processes



A **zombie** process has completed execution (via the system call `exit`) but its entry in the process table exists until the parent process has fetched (via the system call `wait`) the exit status (return code)

Process Creation in Linux/UNIX via `fork` (1/2)

- The system call `fork()` is the only way to create a new process
- If a process calls `fork()`, an identical copy is started as a new process
  - The calling process is called **parent process**
  - The new process is called **child process**
- The child process has the same source code after creation
  - Also the program counters have the same value, which means they refer to the same source code line
- Opened files and memory areas of the parent process are copied for the child process and are independent from the parent process
  - Child process and parent process both have their own process context

`vfork` is a variant of `fork`, which does not copy the address space of the parent process, and therefore causes less overhead than `fork`. Using `vfork` is useful if the child process is to be replaced by another process immediately after its creation. In this course `vfork` is not further discussed.

Process Creation in Linux/UNIX via fork (2/2)

- If a process calls fork(), an exact copy is created
  - The processes differ only in the return values of fork()

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

void main() {
  int return_value = fork();

  if (return_value < 0) {
    // If fork() returns -1, an error happened.
    // Memory or processes table have no more free capacity.
    ...
  }
  if (return_value > 0) {
    // If fork() returns a positive number, we are in the parent process.
    // The return value is the PID of the newly created child process.
    ...
  }
  if (return_value == 0) {
    // If fork() returns 0, we are in the child process.
    ...
  }
}
```

Process Tree

- By creating more and more new child processes with `fork()`, a tree of processes ($\Longrightarrow$ process hierarchy) is created
- The command `pstree` returns an overview of the processes running in Linux/UNIX as a tree according to their parent/child relationships

```
$ pstree
init-+-Xprt
     |-acpid
    ...
     |-gnome-terminal-+-4*[bash]
     |                |-bash---su---bash
     |                |-bash-+-gv---gs
     |                |      |-pstree
     |                |      |-xterm---bash---xterm---bash
     |                |      |-xterm---bash---xterm---bash---xterm---bash
     |                |      `-xterm---bash
     |                |-gnome-pty-helpe
     |                `-{gnome-terminal}
     |-4*[gv---gs]
```

Information about processes in Linux/UNIX

```
$ ps -eFw
UID     PID  PPID  C     SZ    RSS PSR STIME TTY        TIME CMD
root      1     0  0  51286   7432   2 Apr11 ?      00:00:03 /sbin/init
root   1073     1  0  90930   6508   0 Apr11 ?      00:00:00 /usr/sbin/lightdm
root   1551  1073  0  60913   6772   2 Apr11 ?      00:00:00 lightdm --session-child 14 23
bnc    2143  1551  0   1069   1560   0 Apr11 ?      00:00:00 /bin/sh /etc/xdg/xfce4/xinitrc
bnc    2235  2143  0  85195  18888   3 Apr11 ?      00:00:11 xfce4-session
bnc    2284  2235  0 110875  45256   3 Apr11 ?      00:06:20 xfce4-panel --display :0.0
bnc    2389  2235  0 129173  47904   0 Apr11 ?      00:00:26 xfce4-terminal --geometry=80x24
bnc    2471  2389  0   5374   5360   2 Apr11 pts/0  00:00:00 bash
bnc    2487     1  5 316370 395892   0 Apr14 ?      00:08:58 /opt/google/chrome/chrome
bnc    2525  2389  0   5895   6620   3 Apr11 pts/5  00:00:00 bash
bnc    3105  2284  0 597319 257520   0 Apr11 ?      00:05:22 kate -b
bnc    3122  3105  0   5364   5156   2 Apr11 pts/6  00:00:00 /bin/bash
bnc   11196  2471  0 269491 181048   0 Apr14 pts/0  00:00:25 okular bsrn_vorlesung_04.pdf
bnc   16325     1  0 346638 146872   3 10:31 ?      00:00:16 evince BA.pdf
bnc   17384  2525  1 223478  61312   2 10:39 pts/5  00:00:49 dia
bnc   19561  2471  0   9576   3340   0 11:20 pts/0  00:00:00 ps -eFw
```

- C (CPU) = CPU utilization of the process in percent
- SZ (Size) = virtual process size = Text segment, heap and stack (see slide 31)
- RSS (Resident Set Size) = Occupied physical memory (without swap) in kB
- PSR = CPU core assigned to the process
- STIME = start time of the process
- TTY (Teletypewriter) = control terminal. Usually a virtual device: pts (pseudo terminal slave)
- TIME = consumed CPU time of the process (HH:MM:SS)

Independent of Parent and Child Processes

- The example demonstrates that parent and child processes operate independently of each other and have different memory areas

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

void main() {
  int i;
    if (fork())
      // Parent process source code
      for (i = 0; i < 5000000; i++)
        printf("\n Parent: %i", i);
    else
      // Child process source code
      for (i = 0; i < 5000000; i++)
        printf("\n Child : %i", i);
}
```

```
Child : 0
Child : 1
...
Child : 21019
Parent: 0
...
Parent: 50148
Child : 21020
...
Child : 129645
Parent: 50149
...
Parent: 855006
Child : 129646
...
```

- The output demonstrates the CPU switching between the processes
- The value of the loop variable `i` proves that parent and child processes are independent of each other
  - The result of execution cannot be reproduced

Execute on a single CPU core only...

```
$ taskset --cpu-list 1 ./fork_beispiel2.c
```

The PID Numbers of Parent and Child Process (1/2)

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

void main() {
  int pid_of_child;

  pid_of_child = fork();

  // An error occured --> program abort
  if (pid_of_child < 0) {
    perror("\n fork() caused an error!");
    exit(1);
  }

  // Parent process
  if (pid_of_child > 0) {
    printf("\n Parent: PID: %i", getpid());
    printf("\n Parent: PPID: %i", getppid());
  }

  // Child process
  if (pid_of_child == 0) {
    printf("\n Child:  PID: %i", getpid());
    printf("\n Child:  PPID: %i", getppid());
  }
}
```

- This example creates a child process
- Child process and parent process both print:
  - Own PID
  - PID of parent process (PPID)

The PID Numbers of Parent and Child Process (2/2)

- The output is usually similar to this one:

```
Parent: PID: 20835
Parent: PPID: 3904
Child:  PID: 20836
Child:  PPID: 20835
```

- This result can be observed sometimes:

```
Parent: PID: 20837
Parent: PPID: 3904
Child:  PID: 20838
Child:  PPID: 1
```

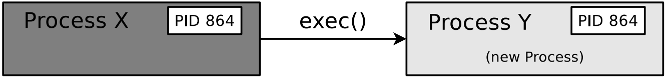
- The parent process was terminated before the child process
  - If a parent process terminates before the child process, it gets `init` as the new parent process assigned
  - Orphaned processes are always adopted by `init`

init (PID 1) is the first process in Linux/UNIX

All running processes originate from `init` $\Longrightarrow$ `init` = father of all processes

Replacing Processes via exec

- The system call `exec()` replaces a process with another one
  - A **concatenation** takes place
  - The new process gets the PID of the calling process
- If one wants to launch a new process from a program, it is necessary, to create a new process with `fork()` and to replace this new process with `exec()`
  - If no new process is created with `fork()` before `exec()` is called, the parent process gets lost
- Steps of a program execution from a shell:
  - The shell creates with `fork()` an identical copy of itself
  - In the new process, the actual program is started with `exec()`



exec Example

```
$ ps -f
UID          PID   PPID   C STIME TTY          TIME CMD
user        1772   1727   0 May18 pts/2    00:00:00 bash
user       12750   1772   0 11:26 pts/2    00:00:00 ps -f
$ bash
$ ps -f
UID          PID   PPID   C STIME TTY          TIME CMD
user        1772   1727   0 May18 pts/2    00:00:00 bash
user       12751   1772  12 11:26 pts/2    00:00:00 bash
user       12769  12751   0 11:26 pts/2    00:00:00 ps -f
$ exec ps -f
UID          PID   PPID   C STIME TTY          TIME CMD
user        1772   1727   0 May18 pts/2    00:00:00 bash
user       12751   1772   4 11:26 pts/2    00:00:00 ps -f
$ ps -f
UID          PID   PPID   C STIME TTY          TIME CMD
user        1772   1727   0 May18 pts/2    00:00:00 bash
user       12770   1772   0 11:27 pts/2    00:00:00 ps -f
```

- Because of the `exec`, the `ps -f` command replaced the bash and got its PID (12751) and PPID (1772)

A further exec Example

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid;
    pid = fork();

    // If PID!=0 --> Parent process
    if (pid) {
        printf("...Parent process...\n");
        printf("[Parent] Own PID:           %d\n", getpid());
        printf("[Parent] PID of the child:  %d\n", pid);
    }
    // If PID=0 --> Child process
    else {
        printf("...Child process...\n");
        printf("[Child]  Own PID:           %d\n", getpid());
        printf("[Child]  PID of the parent: %d\n", getppid());

        // Current program is replaced by "date"
        // "date" will be the process name in the process table
        execl("/bin/date", "date", "-u", NULL);
    }
    printf("[%d ]Program abort\n", getpid());
    return 0;
}
```

- The system call exec() does not exist as wrapper function
- But multiple variants of the exec() function exist
- One of these variants is execl()

Helpful overview about the different variants of the exec() function

http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html

Explanation of the exec Example

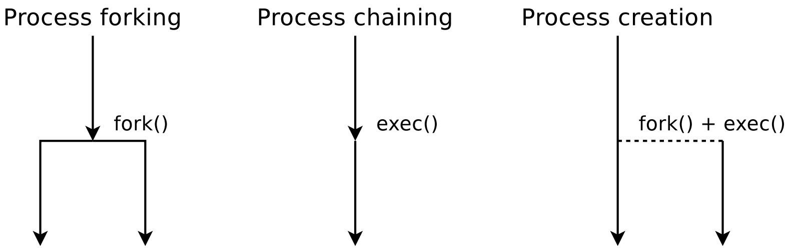
```
$ ./exec_example
...Parent process...
[Parent] Own PID:            25646
[Parent] PID of the child:   25647
[25646 ]Program abort
...Child process...
[Child]  Own PID:            25647
[Child]  PID of the parent:  25646
Di 24. Mai 17:25:31 CEST 2016

$ ./exec_example
...Parent process...
[Parent] Own PID:            25660
[Parent] PID of the child:   25661
[25660 ]Program abort
...Child process...
[Child]  Own PID:            25661
[Child]  PID of the parent:  1
Di 24. Mai 17:26:12 CEST 2016
```

- After printing its PID via `getpid()` and the PID of its parent process via `getppid()`, the child process is replaced via `date`
- If the parent process of a process terminates before the child process, the child process gets `init` as new parent process assigned

Since Linux Kernel 3.4 (2012) and Dragonfly BSD 4.2 (2015), it is also possible for processes other than PID=1 to become the new parent process of an orphaned process
`http://unix.stackexchange.com/questions/149319/new-parent-process-when-the-parent-process-dies/177361#177361`

3 possible Ways to create a new Process

- **Process forking**: A running process creates a new, identical process with `fork()`
- **Process chaining**: A running process creates a new process with `exec()` and terminates itself in this way because it gets replaced by the new process
- **Process creation**: A running process creates a new, identical process with `fork()`, which replaces itself by a new process via `exec()`



Have Fun with Fork Bombs

- A fork bomb is a program, which calls the `fork` system call in an infinite loop
- Objective: Create copies of the process until there is no more free memory
  - The system becomes unusable

Python fork bomb

```python
import os

while True:
    os.fork()
```

C fork bomb

```c
#include <unistd.h>

int main(void)
{
    while(1)
        fork();
}
```

PHP fork bomb

```php
<?php
while(true)
    pcntl_fork();
?>
```

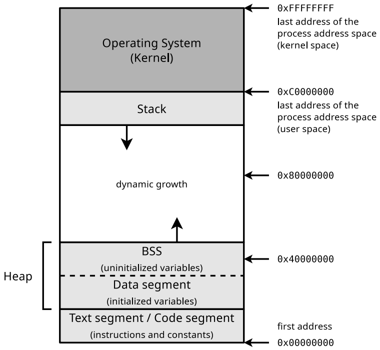
- Only protection option: Limit the maximum number of processes and the maximum memory usage per user

Structure of a UNIX Process in Memory (1/6)

- Default allocation of the virtual memory on a Linux system with a 32-bit CPU
  - 1 GB for the system (kernel)
  - 3 GB for the running process

The structure of processes on 64 bit systems is not different from 32 bit systems. Only the address space is larger and thus the possible extension of the processes in the memory.

- The **text** or **code segment** contains the program code (machine code)
  - Contains the constants too
    - Example: `const int MAX = 100;`
  - Is read-only
  - Can be shared by multiple processes
  - `exec()` reads the text/code segment from the program file

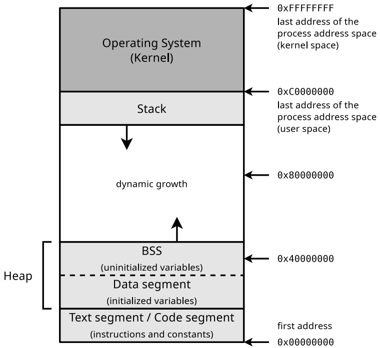


Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), P.58-60
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), P.874-877

Structure of a UNIX Process in Memory (2/6)

- The **heap** grows dynamically and consists of 2 parts:
  1. **data segment**
  2. **BSS**
- The **data segment** contains **initialized** variables and constants
  - Contains all data, which get values assigned in global declarations (outside of functions)
    - Example: `int sum = 0;`
  - `exec()` reads the data segment from the program file



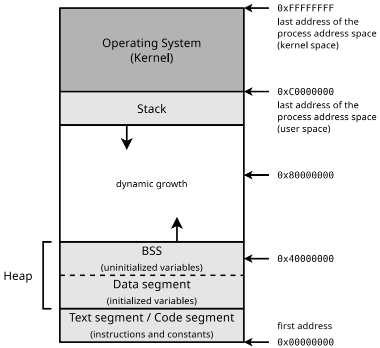
The user space in the memory structure of the processes is the user context (see slide 3). It is the virtual address space (virtual memory) allocated by the operating system ⟹ see slide set 5

Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), P.58-60
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), P.874-877

Structure of a UNIX Process in Memory (3/6)

- The area **BSS** (*block started by symbol*) contains **uninitialized** variables
- Contains global variables (declaration is outside of functions), which get no initial values assigned
  - Example: `int i;`
- Moreover, the process can dynamically allocate memory in this area at runtime
  - In C with the function `malloc()`
- `exec()` initializes all variables in the BSS with 0

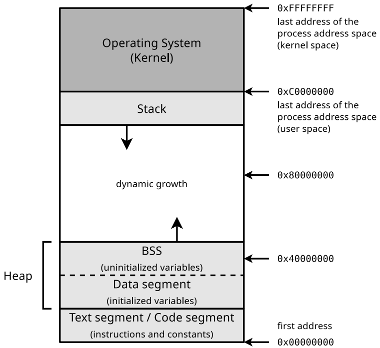


Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), P.58-60
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), P.874-877

Structure of a UNIX Process in Memory (4/6)

- The **stack** is used to implement nested function calls
  - It also contains command line arguments of the program call and environment variables
- Operates according to the LIFO (Last In First Out) principle

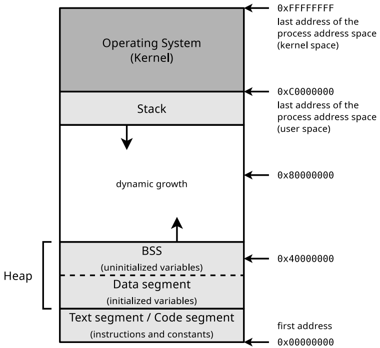


Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), P.58-60
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), P.874-877

Structure of a UNIX Process in Memory (5/6)

- With every function call, a data structure with the following contents is placed onto the stack:
  - Call parameters
  - Return address
  - Pointer to the calling function in the stack
- The functions also add (*push*) their local variables onto the stack
- When returning from from a function, the data structure of the function is removed (*pop*) from the stack



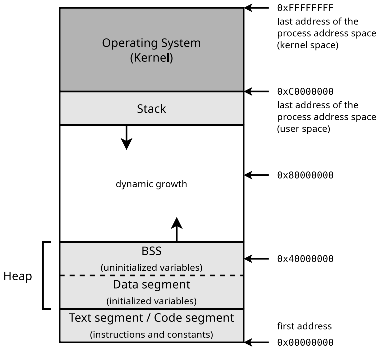
Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), P.58-60
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), P.874-877

Structure of a UNIX Process in Memory (6/6)

- The command `size` returns the size (in Bytes) of the text segment, data segment and BSS of program files
  - The contents of the text segment and data segment are included in the program files
  - All contents in the BSS are set to value 0 at process creation

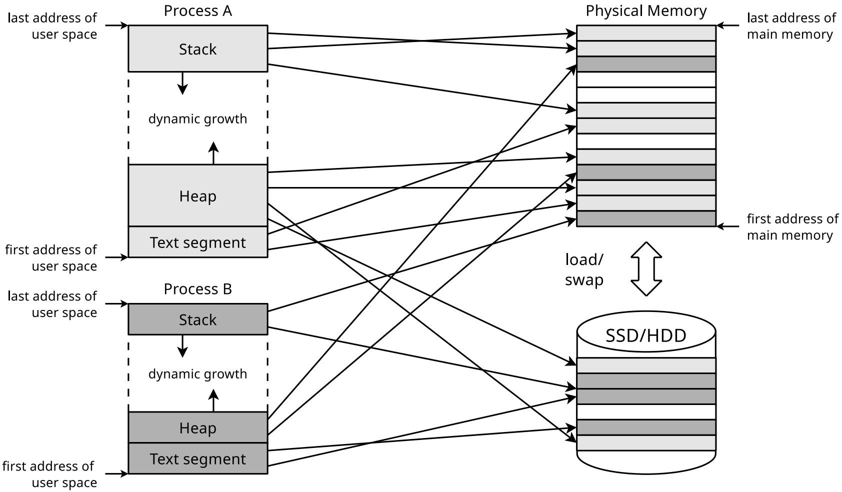
```
$ size /bin/c*
   text    data     bss     dec     hex filename
  46480     620    1480   48580    bdc4 /bin/cat
   7619     420      32    8071    1f87 /bin/chacl
  55211     592     464   56267    dbcb /bin/chgrp
  51614     568     464   52646    cda6 /bin/chmod
  57349     600     464   58413    e42d /bin/chown
 120319     868    2696  123883   1e3eb /bin/cp
 131911    2672    1736  136319   2147f /bin/cpio
```



Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347
Betriebssysteme, *Carsten Vogt*, Spektrum (2001), P.58-60
Moderne Betriebssysteme, *Andrew S. Tanenbaum*, Pearson (2009), P.874-877

Remember: Virtual Memory (Slide Set 5)

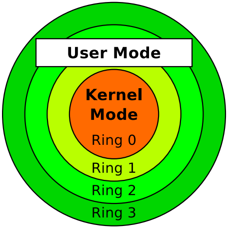


Source: http://cseweb.ucsd.edu/classes/wi11/cse141/Slides/19_VirtualMemory.key.pdf

Processes are stored in physical memory by virtual memory, not in a continuous manner and not always in main memory

User Mode and Kernel Mode

- x86-compatible CPUs implement 4 privilege levels
  - Objective: Improve stability and security
  - Each process is assigned to a ring permanently and cannot free itself from this ring



Implementation of the privilege levels

- The register CPL (Current Privilege Level) stores the current privilege level
- Source: Intel 80386 Programmer's Reference Manual 1986
  http://css.csail.mit.edu/6.858/2012/readings/i386.pdf

- In ring 0 (= **kernel mode**) runs the kernel
  - Here, processes have full access to the hardware
  - The kernel can also address physical memory ($\Longrightarrow$ Real Mode)
- In ring 3 (= **user mode**) run the applications
  - Here, processes can only access virtual memory ($\Longrightarrow$ Protected Mode)

Modern operating systems use only 2 privilege levels (rings)

Reason: Some hardware architectures (e.g. Alpha, PowerPC, MIPS) implement only 2 levels

System Calls (1/2)

We already know...

All processes outside the operating system kernel are only allowed to access their own virtual memory

- If a user-mode process must carry out a higher privileged task (e.g. access hardware), it can tell this the kernel via a **system call**
  - A system call is a function call in the operating system that triggers a switch from user mode to kernel mode ($\Longrightarrow$ **context switch**)

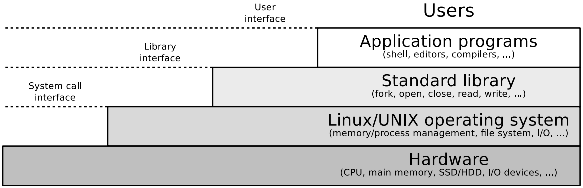
Context switch

- A process passes the control over the CPU to the kernel and is suspended until the request is completely processed
- After the system call, the kernel returns the control over the CPU to the user-mode process
- The process continues its execution at the point, where the context switch was previously requested

- The functionality of a system call is provided in the kernel
  - Thus, outside of the address space of the calling process

System Calls (2/2)

- **System calls** are the interface, which provides the operating system to the user mode processes
  - System calls enable the user mode programs, among others, to create and manage processes and files and to access the hardware

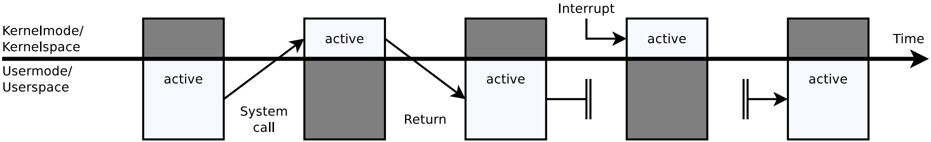


Simply stated...

A system call is a request from a user mode process to the kernel in order to use a service of the kernel

Comparison between System Calls and Interrupts

Interrupts (⟹ slide set 3) are triggered by events outside user-mode processes



Example of a System Call: `ioctl()`

- This way, Linux programs call device-specific instructions
  - `ioctl()` enables processes to communicate with and control of:
    - Character devices (Mouse, keyboard, printer, terminals, ...)
    - Block devices (SSD/HDD, CD/DVD drive, ...)
- Syntax:

```
ioctl (File descriptor, request code number, integer value or pointer to data);
```

- Typical application scenarios of `ioctl()`:
  - Format floppy track
  - Initialize modem or sound card
  - Eject CD
  - Retrieve status and link information of the WLAN interface
  - Access sensors via the Inter-Integrated Circuit (I^2C) data bus

Helpful overviews about system calls

Linux: http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf
Linux: http://syscalls.kernelgrok.com
Linux: http://www.tutorialspoint.com/unix_system_calls/
Windows: http://j00ru.vexillium.org/ntapi

System Calls and Libraries

- Working directly with system calls is **unsafe** and the **portability is poor**
- Modern operating systems provide a library, which is logically located between the user mode processes and the kernel

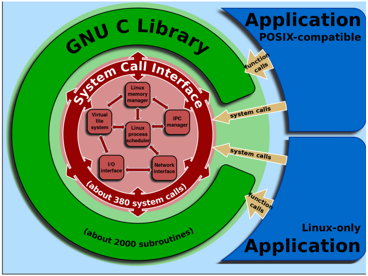


Image Source: Wikipedia
(Shmuel Csaba Otto Traian, CC-BY-SA-3.0)

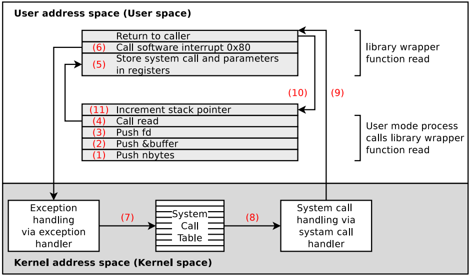
Examples of such libraries

C Standard Library (UNIX), GNU C library glibc (Linux), C Library Implementationen (BSD), Native API `ntdll.dll` (Windows)

- The library is responsible for:
  - Handling the communication between user mode processes and kernel
  - Context switching between user mode and kernel mode
- Advantages which result in using a library:
  - Increased **portability**, because there is no or very little need for the user mode processes to communicate directly with the kernel
  - Increased **security**, because the user mode processes cannot trigger the context switch to kernel mode for themselves

Step by Step (1/4) – `read(fd, buffer, nbytes);`

- In step **1-3**, the user mode process stores the parameters on the stack
- In **4**, the user mode process calls the **library wrapper function** for `read` (⟹ read `nbytes` from the file `fd` and store it inside `buffer`)



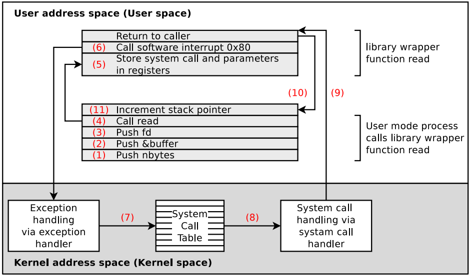
- In **5**, the library wrapper function stores the system call number in the *accumulator register* EAX (32 bit) or RAX (64 bit)
  - The library wrapper function stores the parameters of the system call in the registers EBX, ECX and EDX (or for 64 bit: RBX, RCX and RDX)

Source of this example

Moderne Betriebssysteme, *Andrew S. Tanenbaum*, 3rd edition, Pearson (2009), P.84-89

Step by Step (2/4) – `read(fd, buffer, nbytes);`

- In **6**, the software interrupt (exception) 0x80 (decimal: 128) is triggered to switch from user mode to kernel mode
  - The software interrupt interrupts the program execution in user mode and enforces the execution of an exception handler in kernel mode

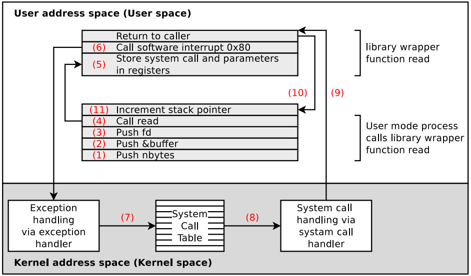


The kernel maintains the *System Call Table*, a list of all system calls

In this list, each system call is assigned to a unique number and an internal kernel function

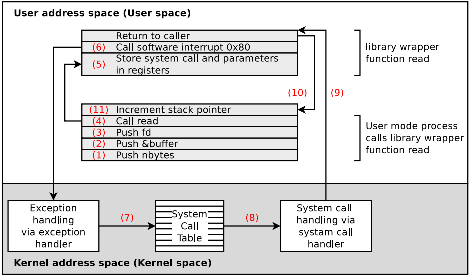
Step by Step (3/4) – `read(fd, buffer, nbytes);`

- The called exception handler is a function in the kernel, which reads out the content of the EAX register
- The exception handler function calls in **7**, the corresponding kernel function from the system call table with the arguments, which are stored in the registers EBX, ECX and EDX
- In **8**, the system call is executed
- In **9**, the exception handler returns control back to the library, which triggered the software interrupt



Step by Step (4/4) – `read(fd, buffer, nbytes);`

- Next, this function returns in **10** back to the user mode process, in the way a normal function would have done it
- To complete the system call, the user mode process must clean up the stack in **11** just like after every function call
- The user process can now continue to operate



The described method with software interrupt 0x80 works under 32-bit and, in most cases, also under 64-bit operating systems. In 64-bit operating systems, however, this working method is considered outdated and slow. Therefore, the more modern way of working is to use the instruction `syscall` (`unistd.h`) and the registers RAX for the syscall number and RDI, RSI, and RDX for the parameters.
More Information:
`https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/`
`https://stackoverflow.com/questions/2535989/what-are-the-calling-conventions-for-unix-linux-system-calls-and-user-space-f`

Example of a System Call in Linux

- System calls are called like library wrapper functions
  - The mechanism is similar for all operating systems
  - In a C program, no difference is visible

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

int main(void) {
  unsigned int ID1, ID2;

  // System call
  ID1 = syscall(SYS_getpid);
  printf ("Result of the system call: %d\n", ID1);

  // Wrapper function of the glibc, which calls the system call
  ID2 = getpid();
  printf ("Result of the wrapper function: %d\n", ID2);

  return(0);
}
```

```
$ gcc SysCallBeispiel.c -o SysCallBeispiel
$ ./SysCallBeispiel
Result of the system call: 3452
Result of the wrapper function: 3452
```

Selection of System Calls

Process management	`fork`	Create a new child process
	`waitpid`	Wait for the termination of a child process
	`execve`	Replace a process by another one. The PID is kept
	`exit`	Terminate a process
File management	`open`	Open file for reading/writing
	`close`	Close an open file
	`read`	Read data from a file into the buffer
	`write`	Write data from the buffer into a file
	`lseek`	Reposition read/write file offset
	`stat`	Determine the status of a file
Directory management	`mkdir`	Create a new directory
	`rmdir`	Remove an empty directory
	`link`	Create a directory entry (link) to a file
	`unlink`	Erase a directory entry
	`mount`	Attach a file system to the file system hierarchy
	`umount`	Detatch a file system
Miscellaneous	`chdir`	Change current directory
	`chmod`	Change file permissions of a file
	`kill`	Send signal to a process
	`time`	Seconds since January 1st, 1970 (“UNIX time”)

Linux System Calls

- The list with the names of the system calls in the Linux kernel...
  - is located in the source code of kernel 2.6.x in the file: `arch/x86/kernel/syscall_table_32.S`
  - is located in the source code of kernel 3.x, 4.x and 5.x in these files: `arch/x86/syscalls/syscall_[64|32].tbl` or `arch/x86/entry/syscalls/syscall_[64|32].tbl`

arch/x86/syscalls/syscall_32.tbl

```
...
1       i386    exit                    sys_exit
2       i386    fork                    sys_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open
6       i386    close                   sys_close
...
```

Tutorials how to implement own system calls

https://www.kernel.org/doc/html/v4.14/process/adding-syscalls.html
https://brennan.io/2016/11/14/kernel-dev-ep3/
https://medium.com/@jeremyphilemon/adding-a-quick-system-call-to-the-linux-kernel-cad55b421a7b
https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38
http://tldp.org/HOWTO/Implement-Sys-Call-Linux-2.6-i386/index.html
http://www.ibm.com/developerworks/library/l-system-calls/