

## 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](mailto:christianbaun@fb2.fra-uas.de)

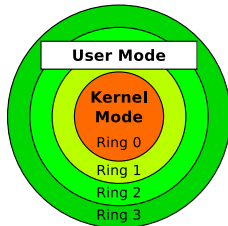
# Learning Objectives of this Slide Set

- At the end of this slide set You know/understand. . .
  - what **System calls** are and how they work
  - what a **process** is from 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**
  - what steps operating systems perform when **processes are created** (via fork or exec) or **erased**
  - what **fork bombs** are
  - the **structure of UNIX processes in memory**

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

# User Mode and Kernel Mode

- x86-compatible CPUs contain 4 privilege levels
  - Objective: Improve stability and security
  - Each process is assigned to a ring permanently and can not 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
  - Processes in this ring have full access to the hardware
  - The kernel can also address physical memory ( $\implies$  Real Mode)
- In ring 3 (= **user mode**) run the applications
  - Processes in this ring can only access virtual memory ( $\implies$  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)

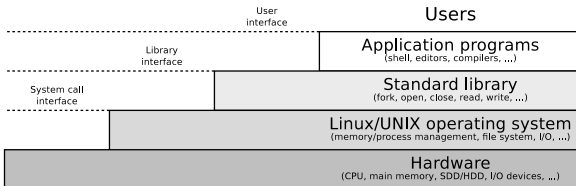
- If a user-mode process must carry out a higher privileged task (e.g. access hardware), it can tell this the kernel via a **system call**
  - A system call is a function call in the operating system, which triggers a switch from user mode to kernel mode ( $\implies$  **context switch**)

## 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, passes the kernel, the control over the CPU back to the user-mode process
  - The process continues its execution at the location, 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

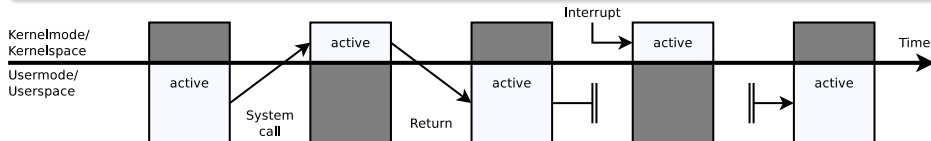


Simply stated...

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 ( $\Rightarrow$  slide set 3) are triggered by events outside user-mode processes



## Example of a System Call: `ioctl()`

- `ioctl()` enables processes to communicate with and control:
  - This way, Linux programs call device-specific instructions
    - 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

Helpful overview about the Linux system calls

*LINUX System Call Quick Reference* from Jialong He

<http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf>

Helpful overview about the Windows system calls

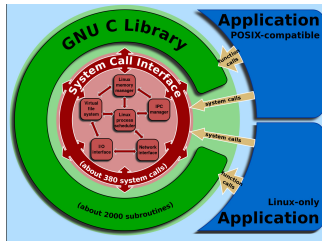
*Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8)* from Mateusz Jurczyk

<http://j00ru.vexillium.org/ntapi>

# System Calls and Libraries

Image Source: Wikipedia

- 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
- 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 can not trigger the context switch to kernel mode for themselves

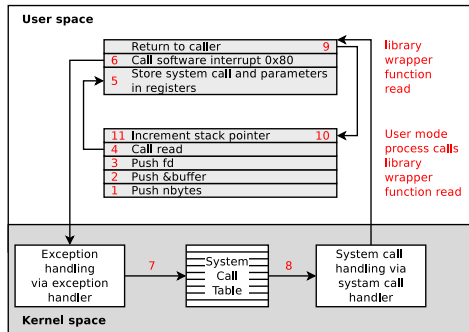


## Examples of such libraries

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

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

- In step **1-3** stores the user mode process the parameters on the stack
- In step **4** calls the user mode process the **library wrapper function** for read ( $\Rightarrow$  read nbytes from the file fd and store it inside buffer)
- In step **5** stores the library wrapper function the system call number in the *accumulator register* EAX
  - The library wrapper function stores the parameters of the system call in the registers EBX, ECX and EDX



Source of this example

Moderne Betriebssysteme, Andrew S. Tanenbaum, 3<sup>rd</sup> edition, Pearson (2009), P.84-89

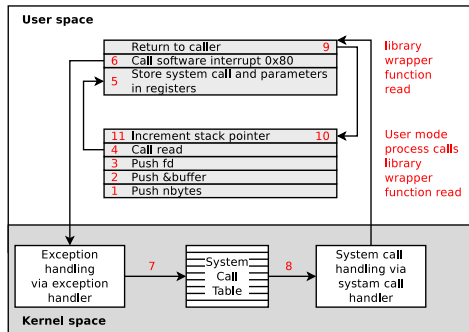


## Step by Step (2/4) – read(fd, buffer, nbytes);

- In step **6**, the software interrupt (exception) 0x80 (decimal: 128) is triggered to switch from user mode to kernel mode

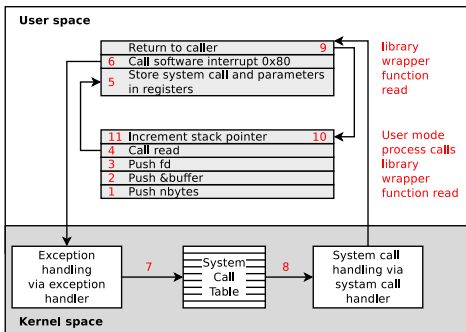
- The software interrupt interrupts the program execution in user mode and enforces the execution of an exception handler in kernel mode

- The kernel maintains a list (system call table) with all system calls
  - In this list, each system call is assigned to a unique number and an internal kernel function



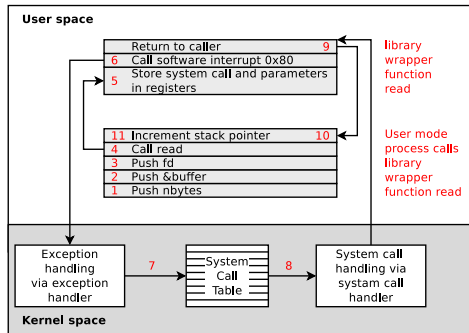
## Step by Step (3/4) – read(fd, buffer, nbytes);

- 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 step **7**, the corresponding kernel function from the system call table with the arguments, which are stored in the registers EBX, ECX and EDX
- In step **8**, the system call is executed



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

- In step **9**, the exception handler returns control back to the library, which triggered the software interrupt
- Next, this function returns in step **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 step **11** just like after every function call
- The user process can now continue to operate

# 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

```
1 #include <syscall.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <sys/types.h>
5
6 int main(void) {
7     unsigned int ID1, ID2;
8
9     // System call
10    ID1 = syscall(SYS_getpid);
11    printf ("Result of the system call: %d\n", ID1);
12
13    // Wrapper function of the glibc, which calls the system call
14    ID2 = getpid();
15    printf ("Result of the wrapper function: %d\n", ID2);
16
17    return(0);
18 }
```

```
$ 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 overall 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	Time since January 1st, 1970

# 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 and 4.x in these files:  
arch/x86/syscalls/syscall\_64.tbl  
arch/x86/syscalls/syscall\_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 for a 2.6.x kernel

<http://tldp.org/HOWTO/Implement-Sys-Call-Linux-2.6-i386/index.html>  
<http://www.ibm.com/developerworks/library/l-system-calls/>

# Process and Process Context

We already know...

- A **process** (lat. *procedere* = proceed, move forward) is an instance of a program, which is executed
  - Processes are dynamic objects and represent sequential activities in a computer system
  - On computers, all the time, multiple processes are executed
  - The CPU is switched back and forth between the processes
- 
- A process includes in addition to the source code its **context**
  - 3 types of context information stores the operating system:
    - **User context**
      - Content of the allocated address space (virtual memory)  $\implies$  slide set 5
    - **Hardware context**
      - CPU registers
    - **System context**
      - Information, which stores the operating system about a process
  - The operating system stores the information in the hardware context and system context in the **process control block**

# Hardware Context

- The **hardware context** includes the content of the CPU registers at the time of process execution
- Registers whose content need to be backed up in case of a context switch:
  - Program Counter (Instruction Pointer) – stores the address of the next instruction to be executed
  - Stackpointer – stores the address at the current end of the stack
  - Basepointer – points to an address in the stack
  - Instruction register – stores the instruction, which is currently executed
  - Akkumulator – stores operands for the ALU and their results
  - Page-table base Register – stores the address of the page table of the current process
  - Page-table length register – stores the length of the page table of the current process

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

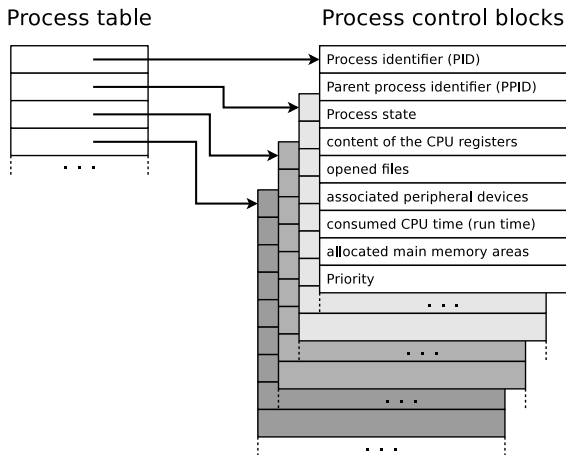


# System Context

- The **system context** are the information, which stores the operating system about a process
- Examples:
  - Entry in the process table
  - Process ID (PID)
  - Process state
  - Information about parent or child processes
  - Priority
  - Identifiers - access credentials to resources
  - Quotas - available quantity of the individual resources
  - Run time
  - Opened files
  - Assigned devices

# Process Table and Process Control Blocks

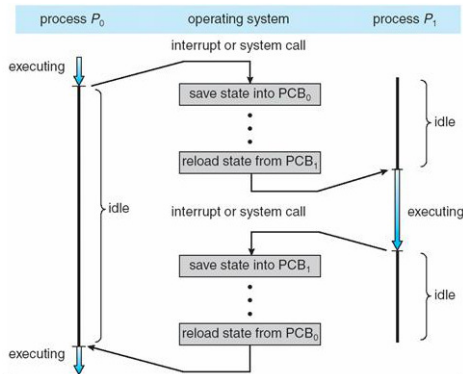
- Each process has its own process context, which is independent of the contexts of other processes



# Process Switching

Image Source: <http://www.cs.odu.edu/~cs471w/spring10/lectures/Processes.htm>

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



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

# Process States

We already know...

Every process is at any moment in a state

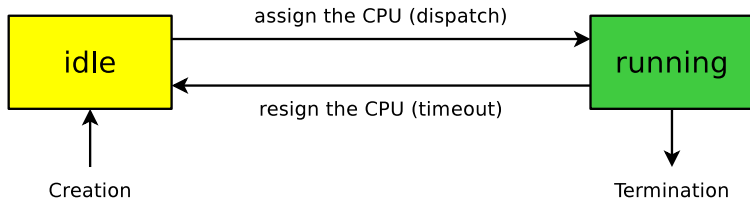
- The number of different states, depends of the process state model of the operating system

## Question

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

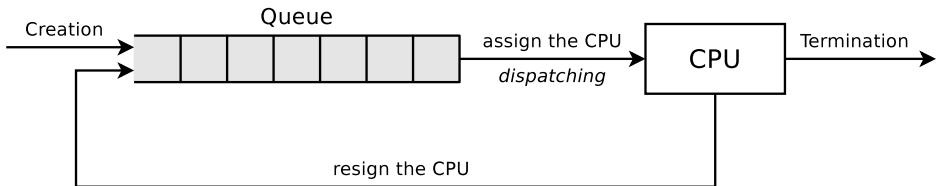
# Process State Model with 2 States

- 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, where they wait for their 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.

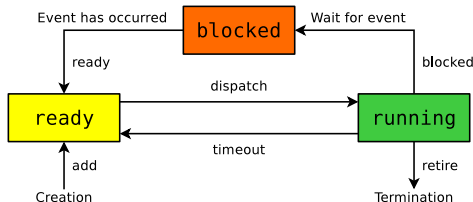
- This model also shows the working method of the **dispatcher**
  - The job of the dispatcher is to carry out the state transitions
- The execution order of the processes is specified by the **scheduler** which uses a **scheduling algorithm** (see slide set 8)

# Conceptual Error of the Process State Model with 2 States

- The process state model with 2 states assumes that every process is always ready for execution
    - 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
  - Solution: The idle processes be distinguished into 2 groups
    - Processes, which are **ready**
    - Processes, which are **blocked**
- ⇒ Process state model with 3 states

# Process State Model with 3 States (1/2)

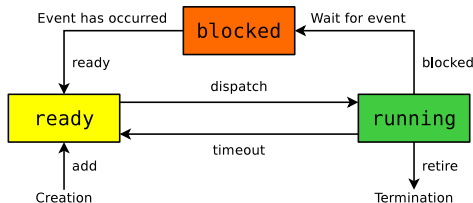
- Each process is in one of the following states:
- **running:**
  - The CPU is assigned to the process and executes its instructions
- **ready:**
  - The process could immediately execute its instructions on the CPU and it is currently waiting for the allocation of the CPU
- **blocked:**
  - The process can currently not be executed and is waiting for the occurrence of an event or the satisfaction of a condition
  - This may be e.g. a message of another process or of an input/output device or the occurrence of a synchronization event





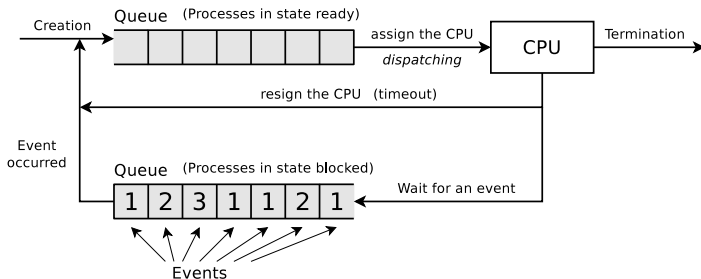
## Process State Model with 3 States (2/2)

- **add**: Process creation and insertion into the list of processes in state **ready**
- **retire**: The process, which has the CPU allocated, terminates
  - Resources, which are occupied by the process, are released
- **dispatch**: The CPU is allocated to a process in state **ready**, which now switches to state **running**
- **blocked**: The running process waits for a message or a synchronization event and switches to the state **blocked**
- **timeout**: The running process is removed from the CPU due to a decision of the scheduler and it switches to state **ready**
- **ready**: The reason, which blocked the process, does not exist anymore and the process switches into the state **ready**



# Process State Model with 3 States – Implementation (1/2)

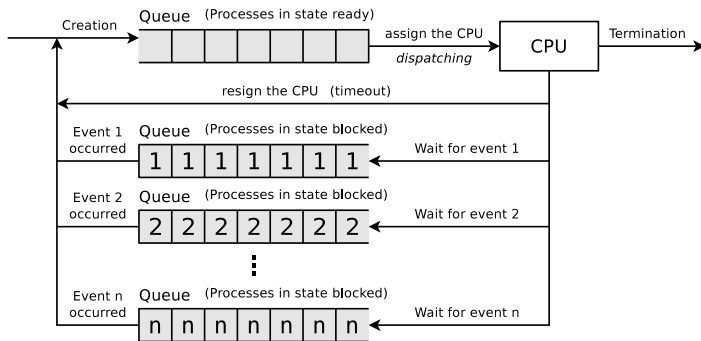
- An implementation approach could use 2 queues
  - A queue for processes in **ready** state
  - A queue for processes in **blocked** state



- Multiple queues for the blocked processes are useful

# Process State Model with 3 States – Implementation (2/2)

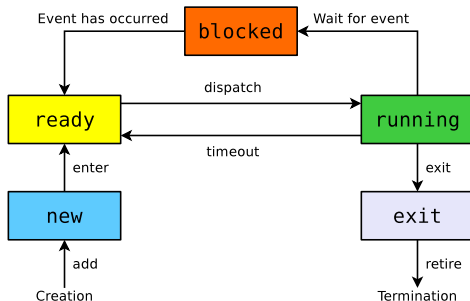
- Multiple queues for blocked processes
  - This way it is implemented by Linux



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

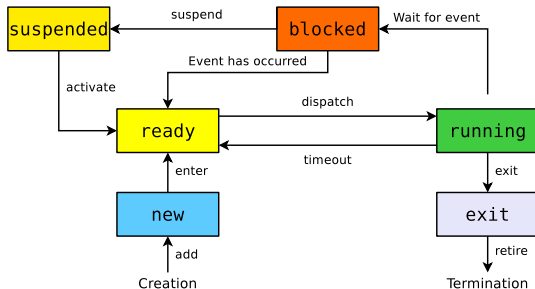
# Process State Model with 5 States

- It makes sense to expand the process state model with 3 states with 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 was halted, 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 multiprogramming



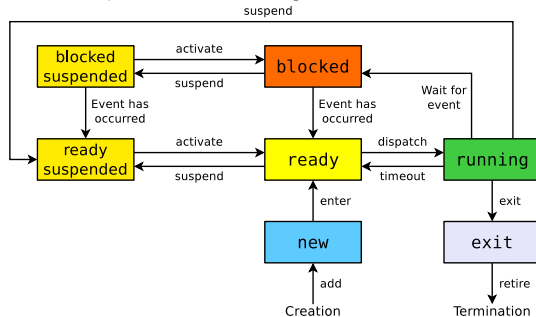
# Process State Model with 6 States

- If not enough physical main memory capacity exists for all processes, parts of processes must be swapped out  $\Rightarrow$  **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**



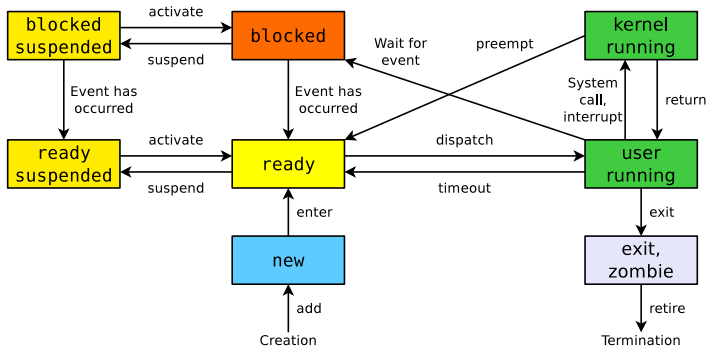
# Process State Model with 7 States

- If a process was suspended, it is better to use the free main memory capacity 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 distinguish the suspended processes in:
  - 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 after creation the same source code
  - 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



# 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()`

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int return_value = fork();
7
8     if (return_value < 0) {
9         // If fork() returns -1, an error happened.
10        // Memory or processes table have no more free capacity.
11        ...
12    }
13    if (return_value > 0) {
14        // If fork() returns a positive number, we are in the parent process.
15        // The return value is the PID of the newly created child process.
16        ...
17    }
18    if (return_value == 0) {
19        // If fork() returns 0, we are in the child process.
20        ...
21    }
22 }
```

# Process Tree

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

```
$ ps tree
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 -aF
UID      PID  PPID  C   SZ    RSS  PSR  STIME  TTY          TIME CMD
root      1      0  0   517    700    0 Nov10  ?           00:00:00 init
user    4311      1  0  9012  20832    0 Nov10  ?           00:00:05 xfce4-terminal
user    4321   4311  0   951   1984    0 Nov10 pts/0       00:00:00 bash
root    4380   4347  0   753   1128    0 Nov10 pts/3       00:00:00 su
root    4381   4380  0   920   1972    0 Nov10 pts/3       00:00:00 bash
user   20920      1  0  1127   2548    0 09:45 pts/1       00:00:00 gv SYS_WS0708.ps
user   20923  20920  0  4587   9116    0 09:45 pts/1       00:00:10 gs -sDEVICE=x11
user   21478   4321  0  1570   2996    0 10:28 pts/0       00:00:00 xterm
user   21479  21478  0   950   1936    0 10:28 pts/4       00:00:00 bash
user   21484  21479  0  1993   5036    0 10:28 pts/4       00:00:00 xterm
user   21485  21484  0   949   1936    0 10:28 pts/5       00:00:00 bash
user   21491  21479  0  1569   2872    0 10:29 pts/4       00:00:00 xterm
user   21492  21491  0   949   1924    0 10:29 pts/6       00:00:00 bash
user   21497  21479  0  1570   2880    0 10:29 pts/4       00:00:00 xterm
user   21498  21497  0   949   1924    0 10:29 pts/7       00:00:00 bash
user   21556  21485  0   672    976    0 10:31 pts/5       00:00:00 ps -aF
```

- RSS (Resident Set Size) = Occupied physical memory (without swap) in kB
- SZ = Size in physical pages of the core image of the process. This includes text segment, heap, and stack
- TIME = Computing time on the CPU
- PSR = CPU, which is allocated to the processes

# Independent of Parent and Child Processes

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

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int i;
7     if (fork())
8         // Parent process source code
9         for (i = 0; i < 5000000; i++)
10             printf("\n Parent: %i", i);
11     else
12         // Child process source code
13         for (i = 0; i < 5000000; i++)
14             printf("\n Child : %i", i);
15 }
```

```
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 switches 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 can not be reproduced

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

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int pid_of_child;
7
8     pid_of_child = fork();
9
10    // An error occurred --> program abort
11    if (pid_of_child < 0) {
12        perror("\n fork() caused an error!");
13        exit(1);
14    }
15
16    // Parent process
17    if (pid_of_child > 0) {
18        printf("\n Parent: PID: %i", getpid());
19        printf("\n Parent: PPID %i", getppid());
20    }
21
22    // Child process
23    if (pid_of_child == 0) {
24        printf("\n Child:  PID: %i", getpid());
25        printf("\n Child:  PPID: %i", getppid());
26    }
27 }
```

- This example creates a child process
- Child process and parent process both print out:
  - 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
```

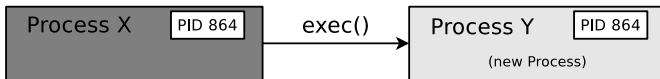
- The parent process was terminated before the child process
  - If a parent process terminates before the child process, it gets `init` as the new parent process assigned
  - Orphaned processes are always adopted by `init`

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

All running processes originate from `init`  $\implies$  `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 the objective is to start a new process out 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

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     int pid;
6     pid = fork();
7
8     // If PID!=0 --> Parent process
9     if (pid) {
10         printf("...Parent process...\n");
11         printf("[Parent] Own PID:          %d\n", getpid());
12         printf("[Parent] PID of the child:  %d\n", pid);
13     }
14     // If PID=0 --> Child process
15     else {
16         printf("...Child process...\n");
17         printf("[Child] Own PID:          %d\n", getpid());
18         printf("[Child] PID of the parent: %d\n", getppid());
19
20         // Corrent program is replaced by "date"
21         // "date" will be the process name in the process table
22         execl("/bin/date", "date", "-u", NULL);
23     }
24     printf("[%d ]Program abort\n", getpid());
25     return 0;
26 }
```

- 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

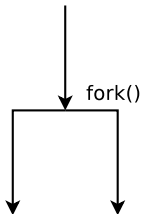
## !!! Update !!!

Since Linux Kernel 3.4 (2012) and Dragonfly BSD 4.2 (2015) it is also possible that other processes than `PID=1` can 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>

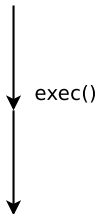
## 3 possible Ways to create a new Process

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

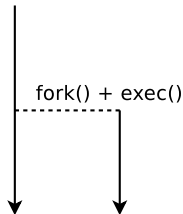
Process forking



Process chaining



Process creation



# 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 no more memory is free
  - The system becomes unusable

## Python fork bomb

```
1 import os
2
3 while True:
4     os.fork()
```

## C fork bomb

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5     while(1)
6         fork();
7 }
```

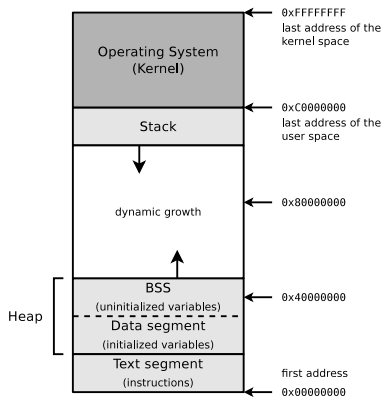
## PHP fork bomb

```
1 <?php
2 while(true)
3     pcntl_fork();
4 ?>
```

- 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 32-bit CPU
  - 1 GB for the system (kernel)
  - 3 GB for the running process
- The **text segment** contains the executable program code (machine code)
- Can be shared by multiple processes
  - Must be kept for this reason only once in physical memory
  - Is therefore usually read-only
- `exec()` reads the text segment from the program file

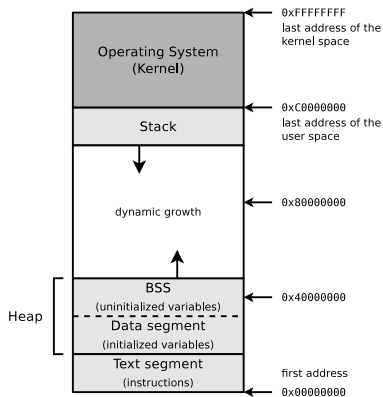


## Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001)  
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

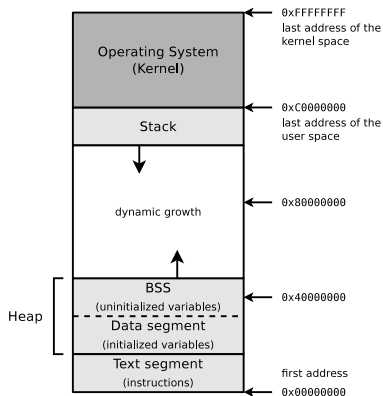


## Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001)  
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 inside the BSS with 0

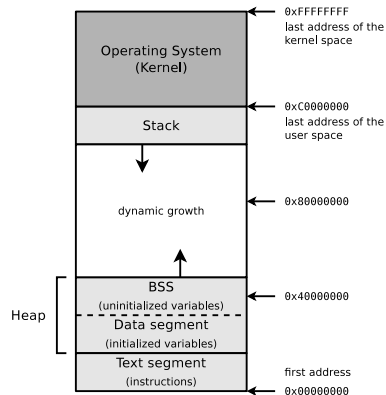


## Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001)  
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 and environment variables of the program call
- Operates according to the LIFO (Last In First Out) principle



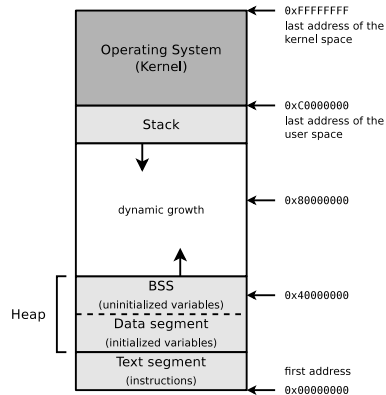
## Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347  
 Betriebssysteme, *Carsten Vogt*, Spektrum (2001)  
 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 on top on the stack:
  - Call parameters
  - Return address
  - Pointer to the calling function inside the stack
- The functions also add (*push*) their local variables on top of the stack
- During a return from a function, the data structure of the function is removed (*pop*) from the stack



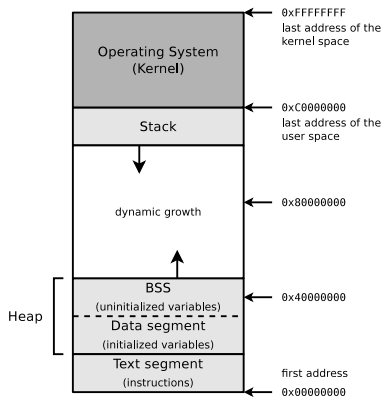
## Sources

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

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

- The command `size` prints out 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

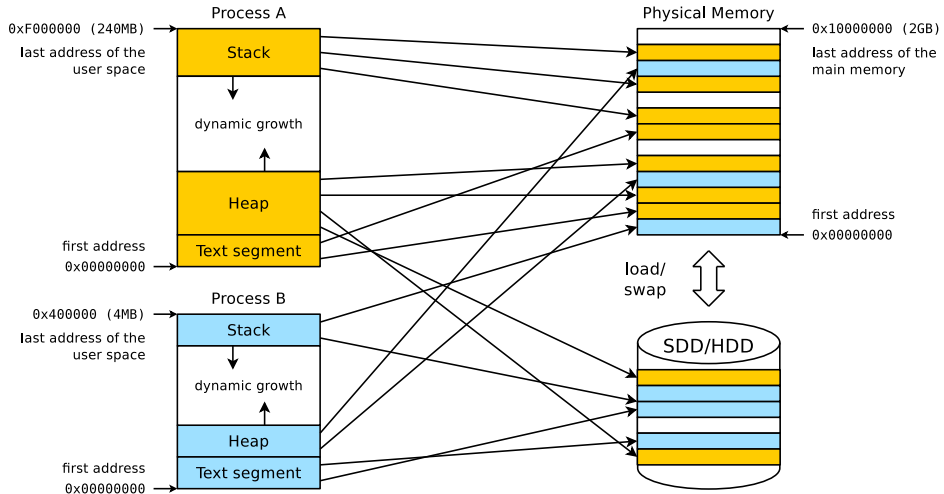
\$	size	/bin/c*				
	text	data	bss	dec	hex	filename
	46480	620	1480	48580	bdc4	/bin/cat
	7619	420	32	8071	1f87	/bin/chac1
	55211	592	464	56267	dbcb	/bin/chgrp
	51614	568	464	52646	cda6	/bin/chmod
	57349	600	464	58413	e42d	/bin/chown
	120319	868	2696	123883	1e3eb	/bin/cp
	131911	2672	1736	136319	2147f	/bin/cpio



## Sources

UNIX-Systemprogrammierung, *Helmut Herold*, Addison-Wesley (1996), P.345-347  
Betriebssysteme, *Carsten Vogt*, Spektrum (2001)  
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](http://cseweb.ucsd.edu/classes/wi11/cse141/Slides/19_VirtualMemory.key.pdf)