Operating Systems

Processes

Prof. Dr. Oliver Hahm

Frankfurt University of Applied Sciences
Faculty 2: Computer Science and Engineering
oliver.hahm@fb2.fra-uas.de
https://teaching.dahahm.de

November 15, 2022

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

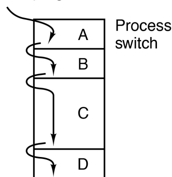
Process

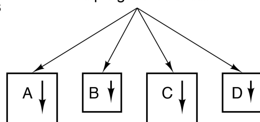
Definition: Process

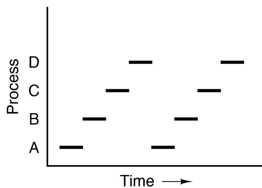
- A process (lat. *procedere* = proceed, move forward) is an instance of a *program*

⇒ A program in execution

- Dynamic objects which represent sequential activities in a computer system
- While running every computer always run (at least) one process
- Each process has assigned resources
- A process can run in *user* or *kernel mode*







Source: Tanenbaum, Modern Operating Systems 4e, (c) 2014 Prentice-Hall, Inc. All rights reserved.

Process Context

- These elements are called the process context
- The operating system manages three types of context information:
  - User context
    - Content of the allocated address space ($\longrightarrow$ virtual memory)
  - Hardware context ($\longrightarrow$ slide 7)
    - CPU registers
  - System context ($\longrightarrow$ slide 8)
    - Information, which stores the operating system about a process
- Typically information about the hardware and system context are stored in the process control block ($\longrightarrow$ slide 9)

Recap: Registers

What is a register?
Which registers do you remember?

Hardware Context

Definition: Hardware Context

The hardware context describes the content of the CPU registers during process execution.

Hardware Context

Definition: Hardware Context

The hardware context describes the content of the CPU registers during process execution.

- The following registers may need to be backed up when switching to another process ($\longrightarrow$ context 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

System Context

Definition: System Context

The information the operating system stores about a process is called the system context. Each process can be uniquely identified by a subset of this information.

System Context

Definition: System Context

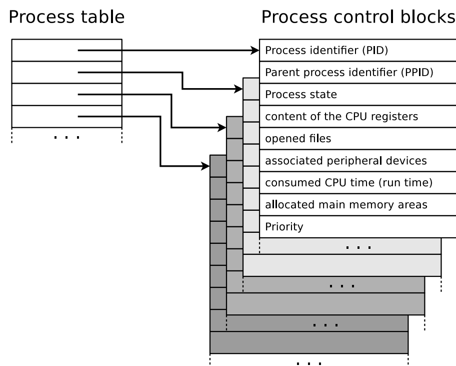
The information the operating system stores about a process is called the system context. Each process can be uniquely identified by a subset of this information.

- **Examples**:
  - Record in the process table,
  - Identifier (→ Process ID (PID)),
  - ⟶ State,
  - Information about parent or child processes,
  - Priority,
  - Identifiers - access credentials to resources,
  - Quotas (allowed usage quantity of individual resources),
  - Runtime,
  - Opened files, or
  - Assigned devices.

Process Table and Process Control Blocks

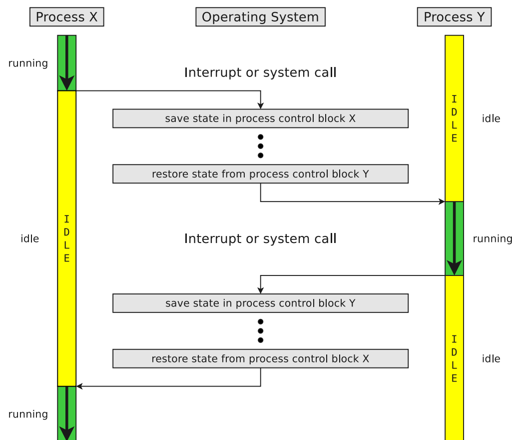
- Each process has its own process context, which is independent of the contexts of other processes

- 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



Context Switching

- If the OS switches from one process to another, the context ($\longrightarrow$ CPU register content) of the former one is stored in the process control block
  - $\Rightarrow$ The content of the process control block of the latter is loaded to restore its context



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

Agenda

- Process Management
- **Process State Models**
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

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

Process States

We already know. . .

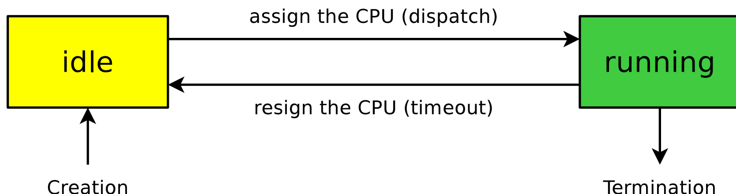
every process is at any moment in a state

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

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

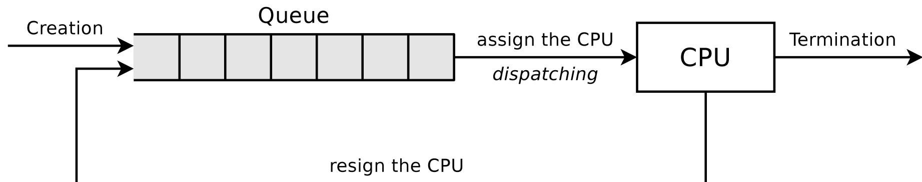
Process State Model with 2 States

- In principle two 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)

- Processes in state idle are stored in a queue ($\rightarrow$ the runqueue), in which they wait for execution
  - The list can be sorted according to the process priority or waiting time



- 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

Process Priorities

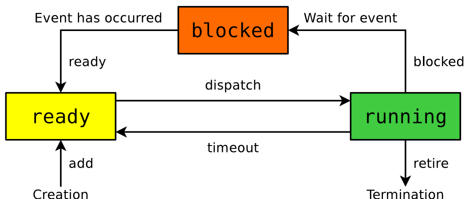
- The priority of a process is proportional to its CPU time
- The process priority is typically expressed as an integer value
  - → A *lower value* represents a *higher priority*
- For **Linux** systems:
  - Priorities between -20 and +19 are available
  - ⇒ -20 is the highest priority and +19 is the lowest priority.
  - The default priority is 0
  - Normal users can assign priorities from 0 to 19
  - The super user (`root`) can assign negative values too
- For **RIOT** systems:
  - Priorities between 0 and 15 are available
  - ⇒ 0 is the highest priority and 15 is the lowest priority.
  - The default priority is 7
  - Priorities are typically fixed at process creation

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 be categorized into 2 groups
  - Processes, which are ready
  - Processes, which are blocked

$\Longrightarrow$ Process state model with 3 states

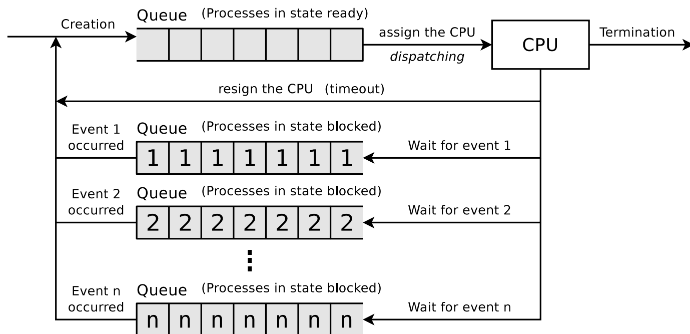
Process State Model with 3 States



- Each process is in one of the following states:
- running:
  - The CPU is assigned to the process and executes its instructions
- ready:
  - The process is ready to run and is currently waiting for the allocation of the CPU
  - This state is sometimes also called pending
- 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 – Implementation

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

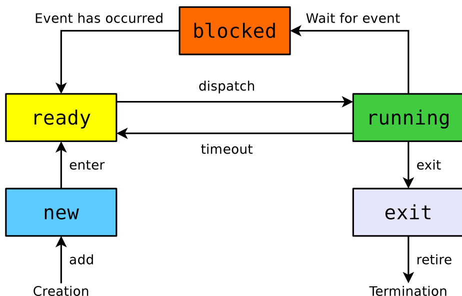


- During state transition, the process control block of the affected process is removed from the old status list and inserted into the new status list
- No separate list exists for processes in running state

Process State Model with 5 States

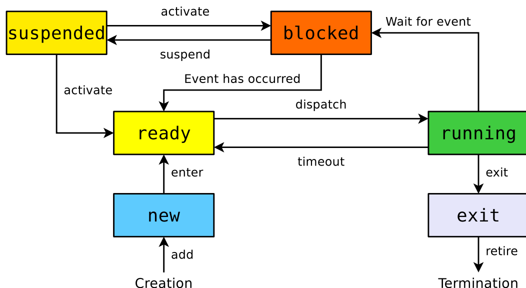
- It makes sense to expand the process state model with 3 states by 2 further process states
  - new: The process (process control block) has been created by the operating system but the process is not yet added to the queue of processes in ready state
  - exit: The execution of the process has finished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

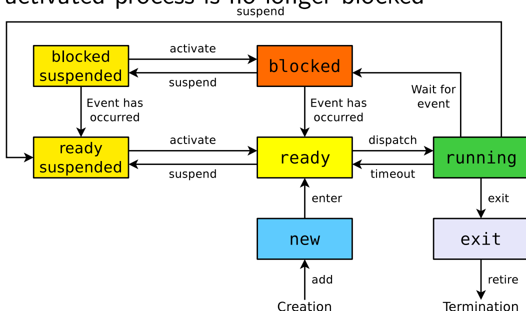
- If not enough physical main memory capacity exists for all processes, parts of processes must be swapped out $\Longrightarrow$ swapping
- The operating system outsources processes, which are in blocked state
- As a result, more main memory capacity is available for the processes in the states running 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

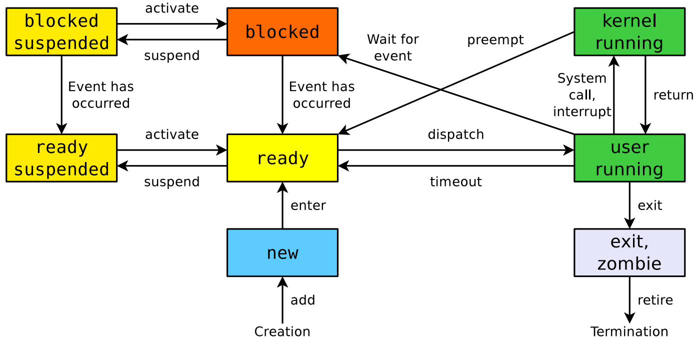
- If a process has been suspended, it is better to use the freed up space in main memory to activate an outsourced process instead of assigning it to a new process
  - This is only useful if the activated process is no longer blocked

- 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)

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

Writing Portable Code

What does one need to do in order to implement an application that can be run on a variety of computers?

POSIX

- POSIX (Portable Operating System Interface) is a family of IEEE standards for operating systems
- Aims for portability and compatibility of applications between different operating systems
- Defines user and system level APIs (application programming interfaces)
- Additionally it defines command line shells and utility interfaces
- It is based on UNIX
- There are few POSIX-certified OS (e.g., macOS, VxWorks, or AIX)
- Many OS (like Linux, FreeBSD, or Minix) are mostly POSIX compliant

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

POSIX Process Creation via `fork`

- In a POSIX system 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 after creation runs the exactly same code
  - Since the program counters are identical as well both processes refer to the same line of code
- All assigned resources (like 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.

Code example for fork on Linux

- 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

- Processes in a system form a tree of processes (($\longrightarrow$ process hierarchy) based on the parent-child relationship
- The commands `pstree` and `ps faux` returns an overview about 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     2471  2389  0   5374   5360   2 Apr11 pts/0    00:00:00 bash
bnc     3105  2284  0 597319 257520   0 Apr11 ?        00:05:22 kate -b
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 48)
- 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)

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

Independent of Parent and Child Processes

- The example demonstrates 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  int main(void) {
5      int i;
6      if (fork())
7      // Parent process source code
8          for (i = 0; i < 5000000; i++)
9              printf("\n Parent: %i", i);
10     else
11     // Child process source code
12         for (i = 0; i < 5000000; i++)
13             printf("\n Child : %i", i);
14 }
15 return 0;
```

```
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  void main() {
5      int pid_of_child;
6      pid_of_child = fork();
7      // An error occured --> program abort
8      if (pid_of_child < 0) {
9          perror("\n fork() caused an error!");
10         exit(1);
11     }
12     // Parent process
13     if (pid_of_child > 0) {
14         printf("\n Parent: PID: %i", getpid());
15         printf("\n Parent: PPID: %i", getppid());
16     }
17     // Child process
18     if (pid_of_child == 0) {
19         printf("\n Child:  PID: %i", getpid());
20         printf("\n Child:  PPID: %i", getppid());
21     }
22 }
```

- 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
```

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 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` or `systemd` (PID 1) is the first process in Linux/UNIX

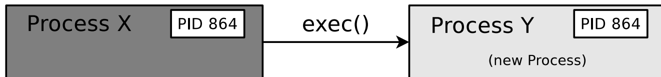
All running processes originate from `init` → init (or systemd) = parent of all processes

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

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 is replaced
- 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 stared 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)

Another 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

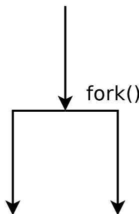
```
$ ./exec_example
...Parent process...
[Parent] Own PID:           25646
[Parent] PID of the child:  25647
[25646 ]Program abort
...Child process...
[Child]  Own PID:           25647
[Child]  PID of the parent: 25646
Di 24. Mai 17:25:31 CEST 2016
$ ./exec_example
...Parent process...
[Parent] Own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

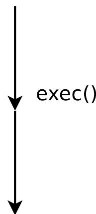
- After printing its PID via `getpid()` and the PID of its parent process via `getppid()`, the child process is replaced via `date`
- If the parent process of a process terminates before the child process, the child process gets `init` as new parent process assigned

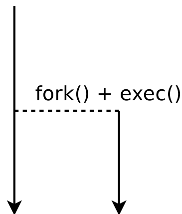
Since Linux Kernel 3.4 (2012) and Dragonfly BSD 4.2 (2015), it is also possible that other processes than PID=1 become the new parent process of an orphaned process
`http://unix.stackexchange.com/questions/149319/new-parent-process-when-the-parent-process-dies/177361#177361`

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







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

Agenda

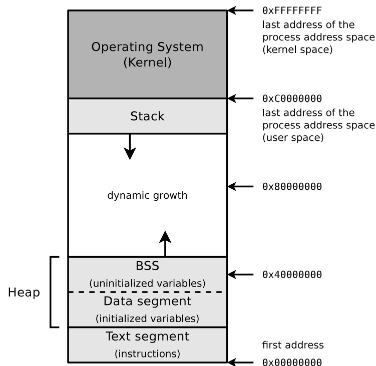
- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

Text Segment

- 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 segment contains the program code (machine code) and other read-only code (e.g., strings literals)
- Can be shared by multiple processes
  - Must be stored for this reason only once in physical memory
  - Is therefore usually read-only
- exec() reads the text segment from the program file



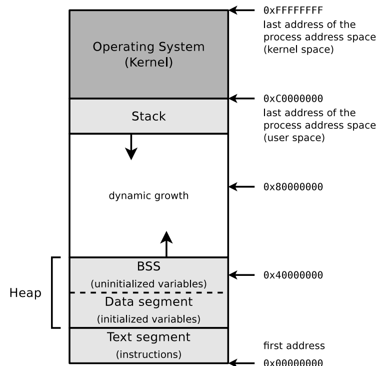
Sources

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

Heap: Data and BSS

- 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 assigned to initialized global variables
    - **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 5). It is the virtual address space (virtual memory) allocated by the operating system

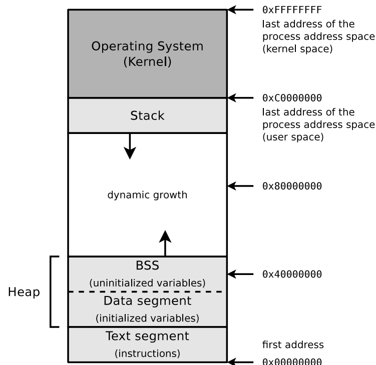


Sources

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

BSS

- The area BSS (*block started by symbol*) contains uninitialized variables
- Contains unitialized global variables
  - **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

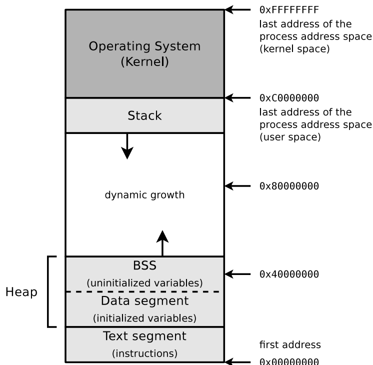


Sources

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

Stack (1/2)

- 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

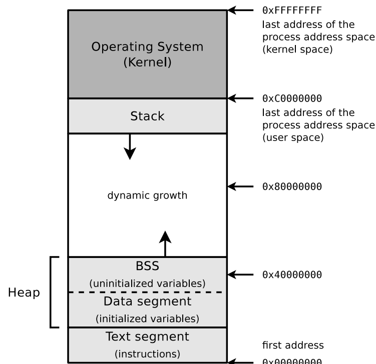


Sources

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

Stack (2/2)

- 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



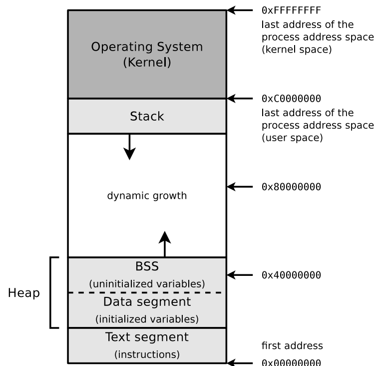
Sources

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

Assessing the Memory Consumption of a Program

- 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

Agenda

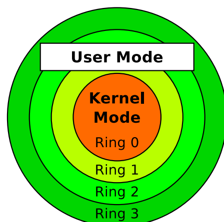
- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

User Mode and Kernel Mode

- x86-compatible CPUs implement four privilege levels
  - **Objective**: Improve stability and security
  - Each process is assigned to a ring permanently



x86 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

- Ring 0 (= kernel mode) runs the kernel
  - ⇒ processes have full access to the hardware
    - The kernel can also address physical memory (⟶ Real Mode)
- Ring 3 (= user mode) run the applications
  - ⇒ processes can only access virtual memory (⟶ Protected Mode)

Modern operating systems use only two privilege levels (rings)

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

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

System Calls (1/2)

How can a process from user space access the hardware?

System Calls (1/2)

How can a process from user space access the hardware?

- 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

System Calls (1/2)

How can a process from user space access the hardware?

- 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

System Calls (1/2)

How can a process from user space access the hardware?

- 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)

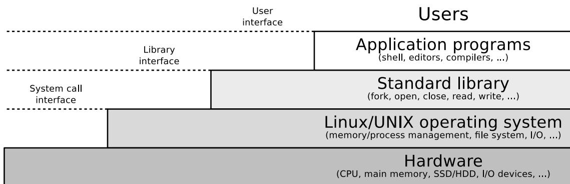
Context switch

- 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

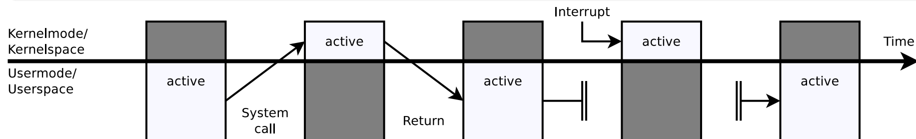


Simply stated. . .

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 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

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
  - Process Creation via fork()
  - Parent and Child Processes
  - Replacing Processes via exec()
  - Structure of a UNIX Process in Memory
- System Calls
  - User Mode and Kernel Mode
  - System Calls
  - System Calls and Libraries

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

Examples of such libraries

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

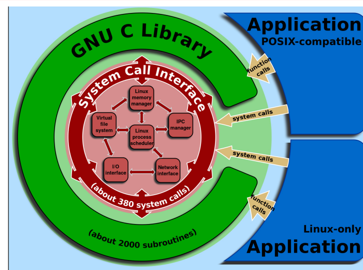
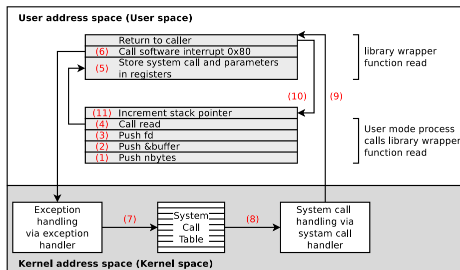


Image Source: Wikipedia (Shmuel Csaba Otto Traian, CC-BY-SA-3.0)

- 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

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

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



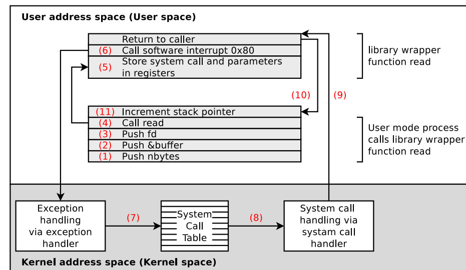
- In 5 stores the library wrapper function the system call number in the *accumulator register* EAX (32 bit) or RAX (64 bit)
  - The library wrapper function stores the parameters of the system call in the registers EBX, ECX and EDX (or for 64 bit: RBX, RCX and RDX)

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

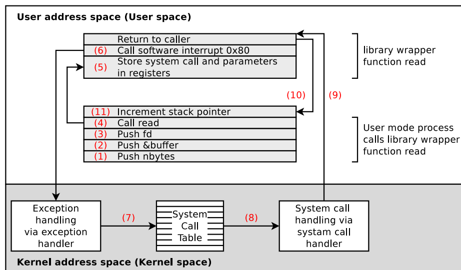


The kernel maintains the *System Call Table*, a list of all system calls

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

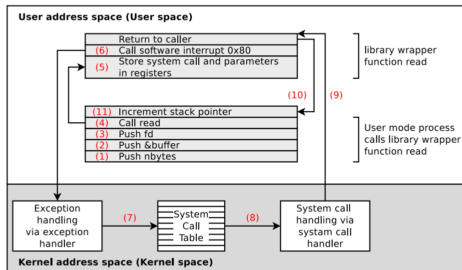
Step by Step (3/4) – `read(fd, buffer, nbytes);`

- 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



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

- In 9, the exception handler returns control back to the library, which triggered the software interrupt
- 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

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

Category	System call	Description
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

You should now be able to answer the following questions:

- What is a process?
- Which information does the hardware and the system context provide?
- What happens when the OS switches from one process to another?
- Which states can a process have?
- How can a new process be started?
- How can a user mode process execute a higher privileged task?