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 14, 2023

What is a process?

Agenda

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
- Structure of a UNIX Process in Memory

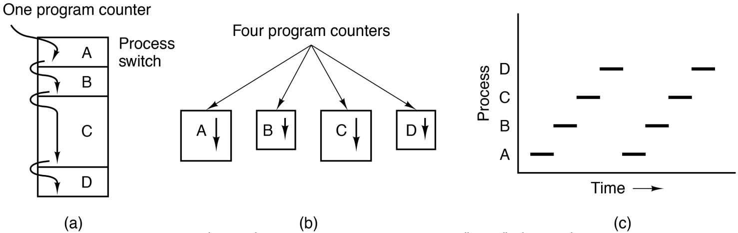
Process

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*



Process Resources

Which resources are associated to a process?

Process Context

- The **resources** associated with a process managed by the OS are called the process context
- The operating system manages three types of context information:

Process Context

- The **resources** associated with a process managed by the OS 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**)

Process Context

- The **resources** associated with a process managed by the OS are called the process context
- The operating system manages three types of context information:
  - User context
    - Content of the allocated address space (⟶ **virtual memory**)
  - Hardware context (⟶ slide 9)
    - CPU registers

Process Context

- The **resources** associated with a process managed by the OS 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 9)
    - CPU registers
  - System context ($\longrightarrow$ slide 10)
    - Information, which stores the operating system about a process

Process Context

- The **resources** associated with a process managed by the OS are called the process context
- The operating system manages three types of context information:
  - User context
    - Content of the allocated address space (⟶ **virtual memory**)
  - Hardware context (⟶ slide 9)
    - CPU registers
  - System context (⟶ slide 10)
    - Information, which stores the operating system about a process
- Typically information about the **hardware** and **system context** are stored in the process control block (PCB) (⟶ slide 11)

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

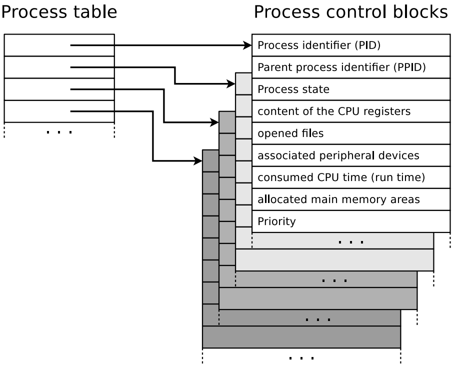
The information the operating system stores about a process is called the system context. Each process can be uniquely identified by a subset of this information.

- **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

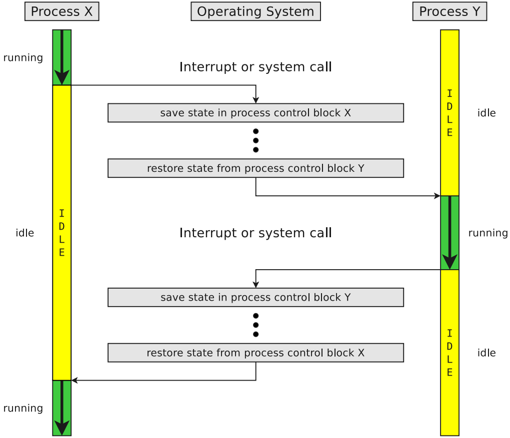
- For managing the processes, the operating system implements the process table
  - It is a list of all existing processes.
  - It contains for each process a record which is called **process control block (PCB)**



Context Switching

- In order to switch from one process to another, the OS stores the context (⟶ CPU register content) of the former one in the **process control block**

⇒ The context of the latter one is restored from the content of its **process control block**



- Each process is at any moment in a particular state
  ⟶ Process state models

Agenda

- Process Management
- Process State Models
- Create and Erase Processes
- Structure of a UNIX Process in Memory

Process States

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

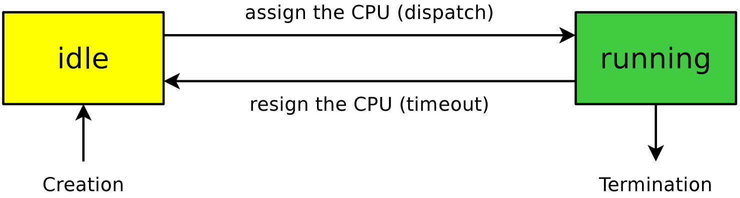
Process States

- 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?

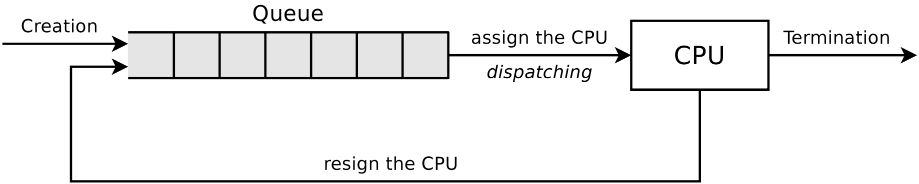
Process State Model with 2 States

- 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 (→ 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

Two States do not suffice in Practice

- The process state model with 2 states assumes that all processes are ready to run at any time
  - This is unrealistic!

Two States do not suffice in Practice

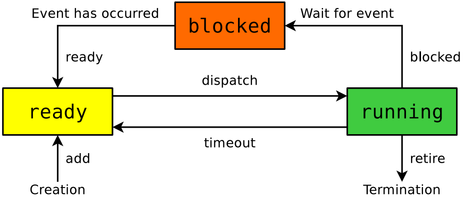
- The process state model with 2 states assumes that all processes are ready to run at any time
  - This is unrealistic!
- In almost any system processes become blocked at some point
  - Possible reasons:
    - They wait for an I/O device
    - They wait for the result of another process
    - They wait for a user input

Two States do not suffice in Practice

- The process state model with 2 states assumes that all processes are ready to run at any time
  - This is unrealistic!
- In almost any system processes become blocked at some point
  - Possible reasons:
    - They wait for an I/O device
    - They wait for the result of another process
    - They wait for a user input
- **Solution:** Split the idle state into two:
  - ready state
  - blocked state

$\Longrightarrow$ Process state model with **3 states**

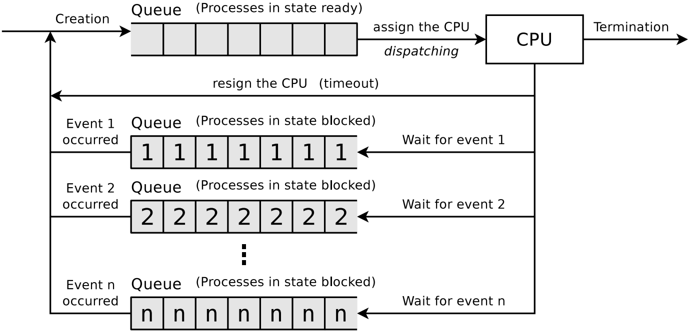
Process State Model with 3 States



- 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 I/O 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

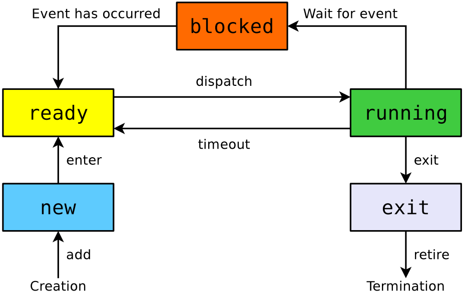


- State transition: When a process state is changed, the corresponding entry is removed from one queue and inserted into another one
- No separate list exists for processes in running state

Process State Model with 5 States

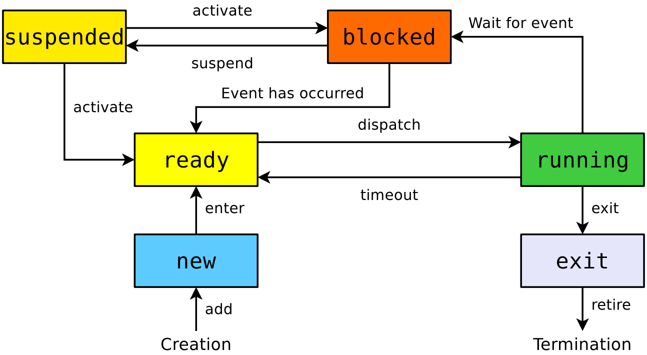
- For many implementations the introduction of two additional states is useful:
  - new: The process (process control block) has been created by the OS but not yet in **ready** state
  - exit: The execution of the process has finished or was terminated but the process control block still exists

- Reason for the existence of the process states **new** and **exit**:
  - The number of executable processes may be limited in order to save memory and to specify the degree of multitasking



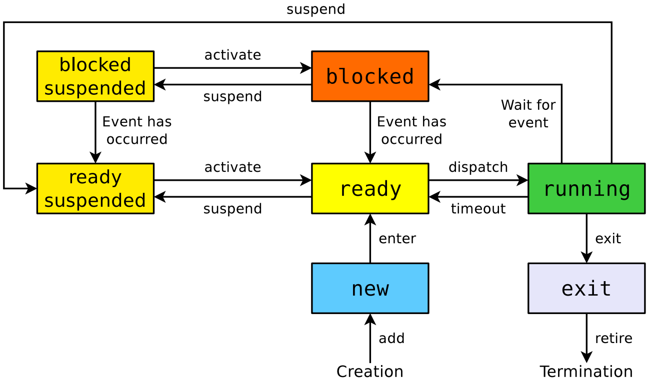
Process State Model with 6 States

- The sum of all processes may exceed the amount of physical main memory ⇒ memory belonging to currently not running processes is swapped out ⟹ swapping
- The OS outsources processes which are in blocked state



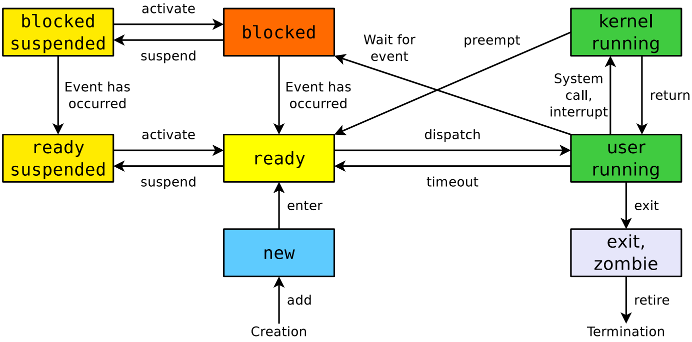
Process State Model with 7 States

- For more efficient use of available memory or in order to reduce waiting time, processes in suspended state may be distinguished into
  - blocked suspended state
  - ready suspended state



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
- Structure of a UNIX Process in Memory

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

What do you already know?

Let's go to the survey again:
`https://pingo.coactum.de/977183`

What do you already know?

Let's go to the survey again:
https://pingo.coactum.de/977183

- Which cache write policy yields the best performance?

What do you already know?

Let's go to the survey again:
https://pingo.coactum.de/977183

- Which cache write policy yields the best performance?
- Which types of context information does the OS store per process?

What do you already know?

Let's go to the survey again:
https://pingo.coactum.de/977183

- Which cache write policy yields the best performance?
- Which types of context information does the OS store per process?
- To which states are there valid transitions from the ready state in a Linux system?

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**
- Child process and parent process both have their own process context, but . . .
- 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
- 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

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(void) {
    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 Hierarchy of a POSIX System

- All processes on a POSIX system are spawned via `fork()`

⇒ All processes are part of the same hierarchy

But which process forms the root of this hierarchy?

Process Hierarchy of a POSIX System

- All processes on a POSIX system are spawned via `fork()`

⇒ All processes are part of the same hierarchy

But which process forms the root of this hierarchy?

`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

Process Tree

- Processes in a system form a tree of processes (⟶ process hierarchy) based on the parent-child relationship

The commands pstree and ps f return an overview about the processes, running in Linux/UNIX, as a tree according to their parent/child relationships

```
 $ ps fax
 1    ?         Ss     0:01 /usr/lib/systemd/systemd --switched-root --system
...
1211 ?         Ss     0:00 dhcpcd: [manager] [ip4] [ip6]
1214 ?         S      0:00  \_ dhcpcd: [privileged proxy]
7775 ?         S      0:00  |   \_ dhcpcd: [BPF ARP] enp0s31f6 10.2.0.190
7778 ?         S      0:00  |   \_ dhcpcd: [BPF ARP] wlan0 10.51.134.219
1215 ?         S      0:00  \_ dhcpcd: [network proxy]
1216 ?         S      0:00  \_ dhcpcd: [control proxy]
1339 ?         Ss     0:00 /usr/lib/systemd/systemd --user
1340 ?         S      0:00  \_ (sd-pam)
1465 ?         Ss     0:00  \_ /usr/bin/dbus-daemon --session --nofork
1511 ?         Ssl    0:00  \_ /usr/lib/at-spi-bus-launcher
1519 ?         S      0:00  |   \_ /usr/bin/dbus-daemon --address=unix:path=/run/user/1000/
     at-spi/bus
```

Information about processes in Linux/UNIX

```
$ ps -eFw
UID     PID  PPID  C     SZ    RSS PSR STIME TTY          TIME CMD
root      1     0  0   5456  12860   2 12:06 ?        00:00:01 /usr/lib/systemd/systemd
root   1311     1  0   1998   4992   4 12:06 ?        00:00:00 login -- oleg
oleg   1339     1  0   5110  11828   4 12:07 ?        00:00:00 /usr/lib/systemd/systemd --user
oleg   1347  1311  0 1122763 171300 0 12:07 tty1  00:00:51 sway
oleg   8031     1  0 285131  31908   3 13:16 ?        00:00:02 foot
oleg   8033  8031  0   4948  15160   7 13:16 pts/2 00:00:02 /usr/bin/zsh
oleg  14043     1  3 949647 569960 4 13:26 ?        00:01:33 /usr/lib/firefox/firefox
oleg  14077     1  0 261432 165640 2 13:26 tty1  00:00:06 Xwayland :0 -rootless -core
oleg  22367     1  0 285340  35712   3 13:54 ?        00:00:01 foot
oleg  22369 22367  0   3710   9548   2 13:54 pts/1 00:00:00 /usr/bin/zsh
root  25003     2  0      0      0   6 14:05 ?        00:00:00 [kworker/6:2-events]
root  25097     2  0      0      0   0 14:05 ?        00:00:00 [kworker/0:2-i915-unordered]
oleg  25202 22369  0   3187   4564   3 14:05 pts/1 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 63)
- 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>
int main(void) {
    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);
    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 switches between the processes
- The value of the loop variable `i` proves that parent and child processes are independent of each other

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>
int main(void) {
    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);
    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 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)

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void main(void) {
    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
```

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`

Replacing Processes via exec

- The system call `exec()` replaces a process with another one
  - The new process gets the PID of the calling process

⇒ To start a new process, one need to ...
  - call `fork()`, and then
  - call `exec()`

If no new process is created with `fork()` before `exec()` is called, the parent process is replaced

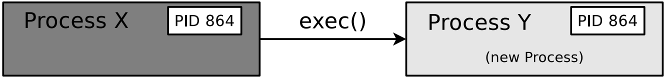
Replacing Processes via exec

- The system call exec() replaces a process with another one
  - The new process gets the PID of the calling process

⇒ To start a new process, one need to …
  - call fork(), and then
  - call 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(void) {
    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

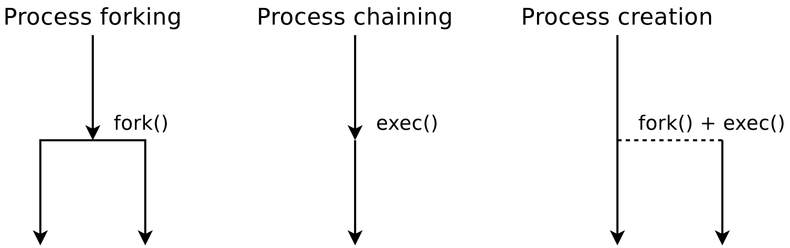
```
$ ./exec_example
...Parent process...
[Parent] Own PID:              25646
[Parent] PID of the child:     25647
[25646 ]Program abort
...Child process...
[Child]  Own PID:              25647
[Child]  PID of the parent:    25646
Di 24. Mai 17:25:31 CEST 2016
$ ./exec_example
...Parent process...
[Parent]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

Python code

```python
import os

while True:
    os.fork()
```

C code

```c
#include <unistd.h>

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

PHP code

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

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 code

```python
import os

while True:
    os.fork()
```

C code

```c
#include <unistd.h>

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

PHP code

```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
- Structure of a UNIX Process in Memory

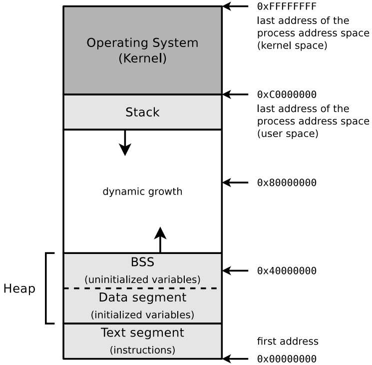
Process' Data

What types of data are being accessed by a process?

Memory Layout of a Unix Process

- 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.

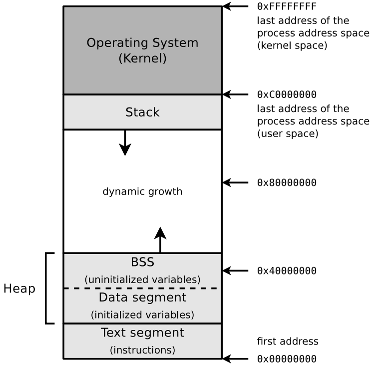


Sources

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

Text Segment

- The text segment contains the program code (machine instructions) and other read-only data (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

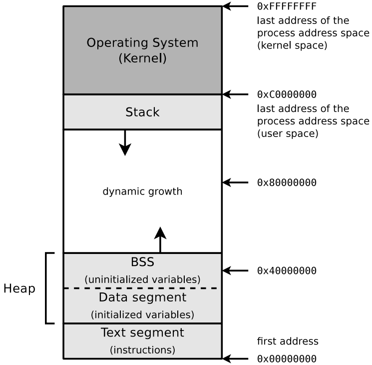


Sources

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



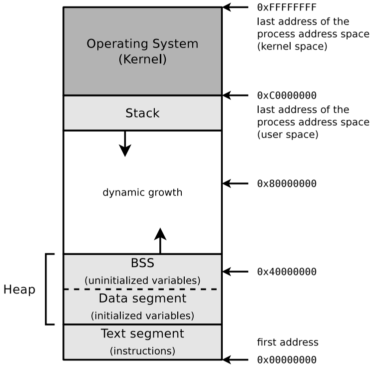
The user space in the memory structure of the processes is the user context (see slide 7). It is the virtual address space (virtual memory) allocated by the operating system

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 uninitialized global variables
  - **Example**: `int i;`
- Moreover, the process can dynamically allocate memory in this area at runtime
  - In C with the function `malloc()`
- The operating system loader typically initializes all variables in the BSS with 0 on start



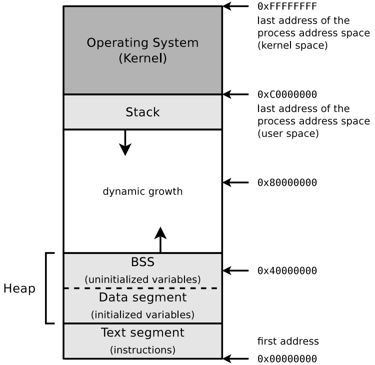
Sources

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 uninitialized global variables
  - **Example**: `int i;`
- Moreover, the process can dynamically allocate memory in this area at runtime
  - In C with the function `malloc()`
- The operating system loader typically initializes all variables in the BSS with 0 on start

Why not simply initialize variables with zero in data?

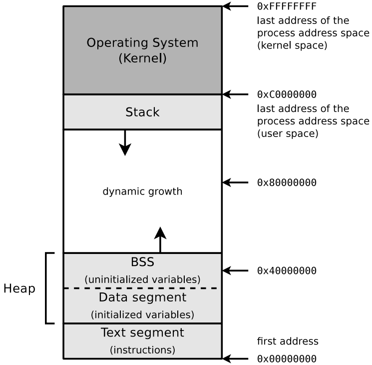


Sources

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

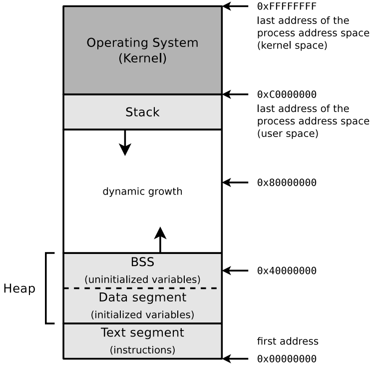


Sources

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



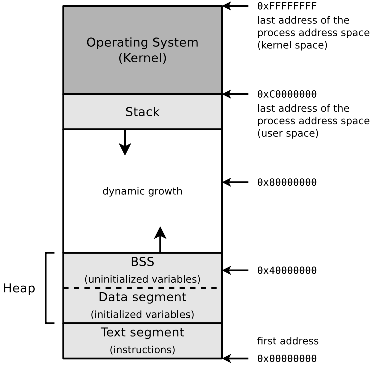
Sources

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

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?