Operating systems

Fundamental concepts

Lecture 1.1

Nicola Dall'Ora

nicola.dallora@univr.it

Nicola Drago

nicola.drago@univr.it

Co-author: Florenc Demrozi

University of Verona
Department of Computer Science

2023/2024

Table of Contents

Processes and Programs

Processes and Programs

Process

A *process* is an instance of an executing program.

Program

A *program* is a binary file containing a set of information that describes how to construct a process at run time

From the Kernel's point of view, a process consists of:

- user-space memory containing program code,
- the variables used by that code, and
- a set of kernel data structures that maintain information about the process's state (e.g. page tables, table of open files, signals to be delivered, process resource usage and limits, ...)

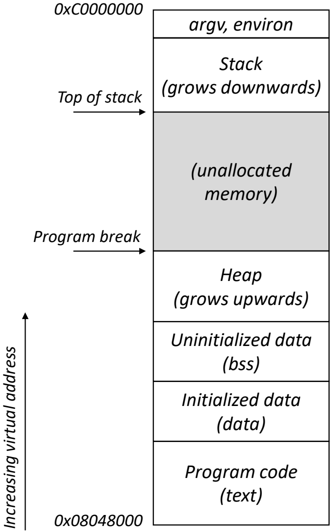
Processes and Programs

Memory Layout of a Process

Memory Layout of a Process (1/5)

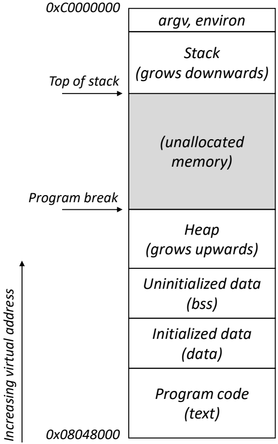
Typical memory (RAM) layout of a process on Linux/x86-32

- The **program code** read-only segment containing machine-language instructions (text)
- The **initialized data** segment containing initialized global and static variables (data)
- The **uninitialized data** segment containing not initialized global and static variables (bss)
- The **heap** segment containing dynamically allocated variables ↑
- The **stack** segment containing for each called function its arguments and locally declared variables ↓



Memory Layout of a Process (2/5)

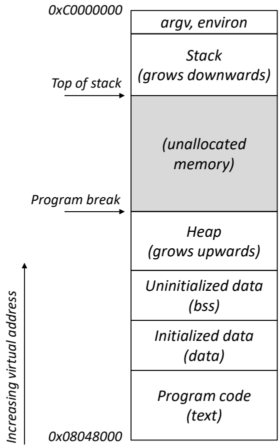
```c
#include <stdlib.h>
// Declared global variables
char buffer[10];                        // <- (bss)
int primes [] = {2, 3, 5, 7};           // <- (data)
// Function implementation
void method(int *a) {                   // <- (stack)
    int i;                              // <- (stack)
    for (i = 0; i < 10; ++i)
        a[i] = i;
}
// Program entry point
int main (int argc, char *argv[]) {     // <- (stack)
    static int key = 123;               // <- (data)
    int *p;                             // <- (stack)
    p = malloc(10 * sizeof(int));       // <- (heap)
    method(p);
    free(p);
    return 0;
}
```



Memory Layout of a Process (3/5)

You can query segments size of the previous code, by means of the `size` command:

```
user@localhost[~]$ size main
     text    data     bss     dec     hex   filename
     1695     628      24    2347     92b       main
```



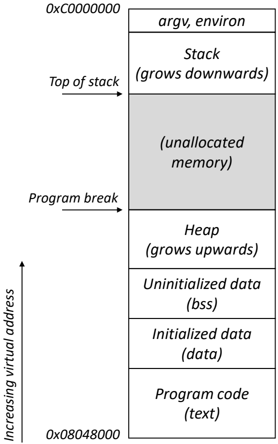
Memory Layout of a Process (4/5)

- Code-1 Example

```c
int main (int argc, char *argv[]) {
    char *string = "ciao";
    string[0] = ‘C‘;
    printf("%s\n", string);
    return 0;
}
```

- Code-2 Example

```c
int main (int argc, char *argv[]) {
    char string[] = "ciao";
    string[0] = ‘C‘;
    printf("%s\n", string);
    return 0;
}
```



Why do we have a Segmentation fault error?

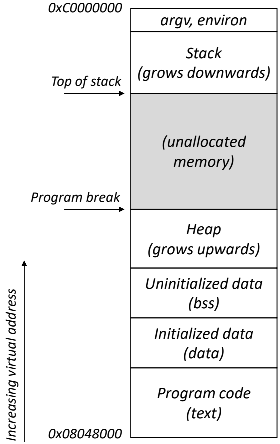
Memory Layout of a Process (5/5)

- Code-1 Example

```
int main (int argc, char *argv[]) {
    char *string = "ciao";
    string[0] = ‘C‘;
    printf("%s\n", string);
    return 0;
}
```

- Code-2 Example

```
int main (int argc, char *argv[]) {
    char string[] = "ciao";
    string[0] = ‘C‘;
    printf("%s\n", string);
    return 0;
}
```



Why do we have a Segmentation fault error with Code-1? (advice: text segment)

Processes and Programs

File descriptor table (overview)

File descriptor table (overview)

For each generated process the Kernel maintains a *file descriptor table*. Each entry of the table is a *file descriptor*, namely a positive number representing an input/output resource opened by the process (*e.g.* files, pipes, sockets, . . .).

By convention, three file descriptors are always present in a new process:

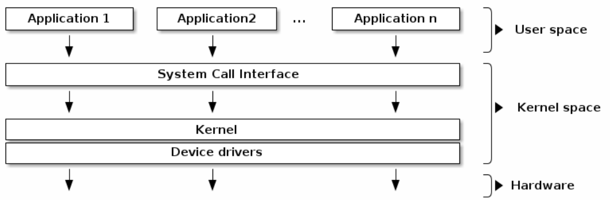
File descriptor	Purpose	POSIX name
0	standard **input**	STDIN_FILENO
1	standard **output**	STDOUT_FILENO
2	standard **error**	STDERR_FILENO

Further details about *file descriptor table* are reported in File system chapter.

System calls

System calls (1/2)

Typical operating system architecture



A *system call* is a controlled entry point into the Kernel, allowing a process to request a service. For example, the services provided by Kernel include: creation of a new process, execution of I/O operation, creation of a pipe for interprocess communication . . . .

System calls (2/2)

The *syscalls(2)* manual page lists the available Linux system calls. Technical details are available for each *system call* through the man(2) command (*e.g. man 2 open*)

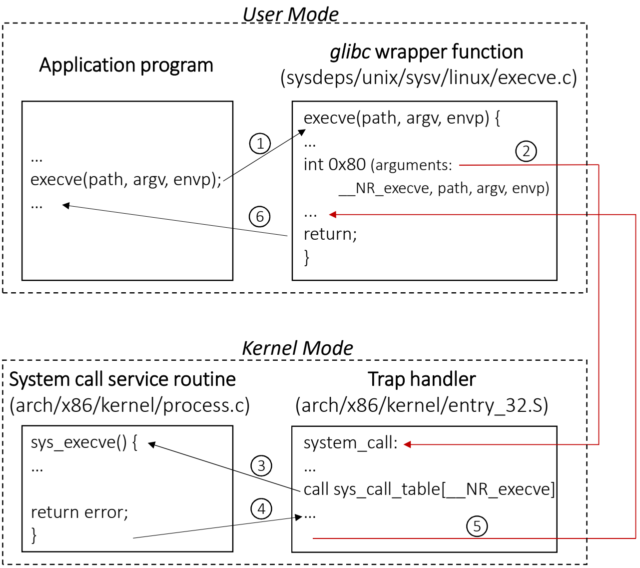
From a programming point of view, invoking a system call looks much like calling a C function. However, the following steps are performed behind a system call execution.

(For more information read The Linux Kernel)

System calls

System call execution

System call execution (1/4)



System call execution (2/4)

1. The application makes a system call by calling a wrapper function in the C library.
2. The wrapper function: copies the system call arguments from the stack to specific CPU registers, copies the system call number into the %eax CPU register [1]. Finally, the wrapper makes the CPU switch from *user mode* to *kernel mode* (*e.g. int 0x80* software interrupt).

[1] The set of system call is fixed. Each system call is identified by a name in C library, and by a unique number in the Kernel! The `execve()` system call has the number 11 (`__NR_execve`) in Linux/x86-32.

System call execution (3/4)

3. the Kernel executes `system_call()` routine which: saves the register values onto the kernel stack, checks the validity of the system call number, and invokes the system call service routine[2]
4. The service routine performs the required task. Finally, a *result status* is returned to the `system_call()`.

[2] The `sys_call_table` vector contains a pointer to the system call service routine. The 11-th entry of `sys_call_table` contains a function pointer to the `sys_execve` service routine.

System call execution (4/4)

5. The `system_call()` routine restores the CPU register values from the kernel stack and place the *result status* of the executed service routine on the stack. Simultaneously it switches the CPU from *kernel mode* to *user mode* and returns to the C wrapper function.
6. If the return value of the system call service routine indicated an error, then the wrapper function sets the global variable `errno` using this value. Finally, the wrapper function returns to the caller an integer value indicating the success or failure of the system call.

By convention, the negative number -1 (or a NULL pointer), indicate an error to the calling application program.

System calls

Handling system call errors

Handling system call errors (1/7)

The section *ERRORS* in the manual page of each *system call* documents the possible return value(s) indicating an error. Usually, a *system call* notifies an error by returning –1 (or a NULL pointer) as a result.

When a system call fails, the global integer variable `errno` is set to a positive value that identifies the occurred error. Including the <errno.h> header file provides a declaration of `errno`, as well as a set of constants for the various error numbers.

Handling system call errors (2/7)

Simple example of the use of `errno`[3] to diagnose a system call error

```c
#include <errno.h>
...
// system call to open a file
fd = open(pathname, flags, mode);
// BEGIN code handling errors.
if (fd == -1) {
    if (errno == EACCES) {
        // Handling not allowed access to the file
    } else {
        // Some other error occurred
    }
}
// END code handling errors
...
```

[3] Linux system errors (link)

Handling system call errors (3/7)

A few *system calls* (e.g., getpriority()) can return –1 on success. To determine whether an error occurs with such calls, we set `errno` to 0 before calling the *system call*. If the call returns –1 and `errno` is nonzero, then an error occurred.

```c
#include <sys/resource.h>
...
// Reset the errno variable to 0
errno = 0;
// System call getpriority gets the nice value of a process
nice = getpriority(which, who);
if ( (nice == -1) && (errno != 0) ) {
    // Handling getpriority errors
}
...
```

Handling system call errors (4/7)

The `perror()` function prints on standard error the string `msg` followed by a message that describes last error encountered during the last *system call*.

```
#include <stdio.h>

void perror(const char *msg);
```

Handling system call errors (5/7)

Simple example of the use of `perror` to print a message describing the occurred error.

```c
#include <stdio.h>
...
// System call to open a file.
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("<Open>");
    // System call to kill the current process.
    exit(EXIT_FAILURE);
}
...
```

Example output:

```
<Open>: No such file or directory
```

Handling system call errors (6/7)

The strerror() function returns the error string corresponding to the error number given in its errnum argument.

```
#include <string.h>

char *strerror(int errnum);
```

The string returned by strerror() could be overwritten by subsequent calls to strerror(). If errnum is a unrecognized error number, strerror() returns a string of the form Unknown error nun.[4].

[4] On some other implementations, strerror() returns NULL when errnum is unrecognized

Handling system call errors (7/7)

Simple example of the use of `strerror` to print a message describing the occurred error.

```c
#include <stdio.h>
...
// System call to open a file
fd = open(path, flags, mode);
if (fd == -1) {
    printf("Error opening (%s):\n\t%s\n", path, strerror(errno));
    // System call to kill the current process
    exit(EXIT_FAILURE);
}
...
```

Example output:

```
Error opening (myFile.txt):
    No such file or directory
```

Function errExit

Throughout these slides the function `errExit` is used as a short cut to print a message and terminate a process. The following is its C implementation:

```
void errExit(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

N.B.:
The function `errExit` is not a default/standardized C function. In order to replicated the examples of these slides you must first define it!

System calls

strace command

strace command

The `strace` system command let us find out what system calls a process is using. In its simplest form, `strace` is used as follows:

```
user@localhost[~]$ strace command arg...
```

```
alessandro@DESKTOP-CDEI5N7:/tmp/test$ strace ls /tmp
execve("/bin/ls", ["ls", "/tmp"], [/* 25 vars */]) = 0
brk(NULL)                               = 0x1461000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=67920, ...}) = 0
mmap(NULL, 67920, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f0ef7acf000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260Z\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0ef7ac0000
mmap(NULL, 2234080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f0ef75d0000
mprotect(0x7f0ef75ef000, 2093056, PROT_NONE) = 0
mmap(0x7f0ef77ee000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e000) = 0x7f0ef77ee000
mmap(0x7f0ef77f0000, 5856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f0ef77f0000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
```

Kernel data types

Kernel data types (1/2)

Even on a single Linux implementation, the data types used to represent information may differ between kernel releases. Example:
On Linux $\leq$2.2, user and group IDs were represented by 16 bits, meanwhile on Linux $\geq$2.4 and later, they are represented by 32 bits.

To avoid portability problems various standard system data types were defined. Each of these types is defined using the *C typedef* feature. Most of the standard system data types have names ending in _t. Many of them are declared in the header file <sys/types.h>

Example:
`pid_t` data type is intended for representing process IDs. On Linux/x86-32 this type is defined as `typedef int pid_t;`

Kernel data types (2/2)

The following table lists some of the system data types that we'll encounter in this course.

Data type	Type requirement	Description
ssize_t	signed integer	byte count or error indication
size_t	unsigned integer	byte count
off_t	signed integer	file offset
mode_t	integer	file permission and type
pid_t	signed integer	process, or process group, or session ID
uid_t	integer	numeric user identifier
gid_t	integer	numeric group identifier
key_t	arithmetic type	System V IPC type
time_t	integer or real floating	time in seconds since Epoch
msgqnum_t	unsigned integer	counts of messages in a queue
msglen_t	unsigned integer	number of allowed byte for a msg
shmatt_t	unsigned integer	counts attaches fo a shared mem.

Manual pages

Manual pages (1/2)

The manual pages are a set of pages that explain every command available on your system including what they do, the specifics of how you run them and what command line arguments they accept. Manual pages are accessible via the *man* command. Example:

```
man <command>
```

A manual page is usually divided into numbered sections:

1. User commands
2. System calls documentation
3. Library functions documentation provided by the standard C library
4. Devices documents details
5. File Formats and Conventions

Manual pages (2/2)

How to get the documentation of...

- cd bash command: man cd (or man 1 cd)
- open system call: man 2 open
- strlen C function: man 3 strlen
- hard disk devices: man 4 hd
- file format fstab: man 5 fstab

Utility:
The command man -k <str> search the short descriptions and manual page names for the keyword str as regular expression.