

# Linux System Programming

G. Filosofi 2019

## Concepts and definitions

- *program*: a file containing a range of information that describes how to construct a process at run time. One program may be used to construct many processes, or, put conversely, many processes may be running the same program.
- *process*: an abstract entity, defined by the kernel, to which system resources are allocated in order to execute a program
- *COFF* (Common Object File Format): a UNIX binary file format
- *ELF* (Executable and Linking Format): used by Linux executable files
- Multitasking: multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). The rules governing which processes receive use of the CPU

and for how long are determined by the kernel scheduler (Preemptive multitasking) or by the processes themselves (non-preemptive or collaborative multitasking)

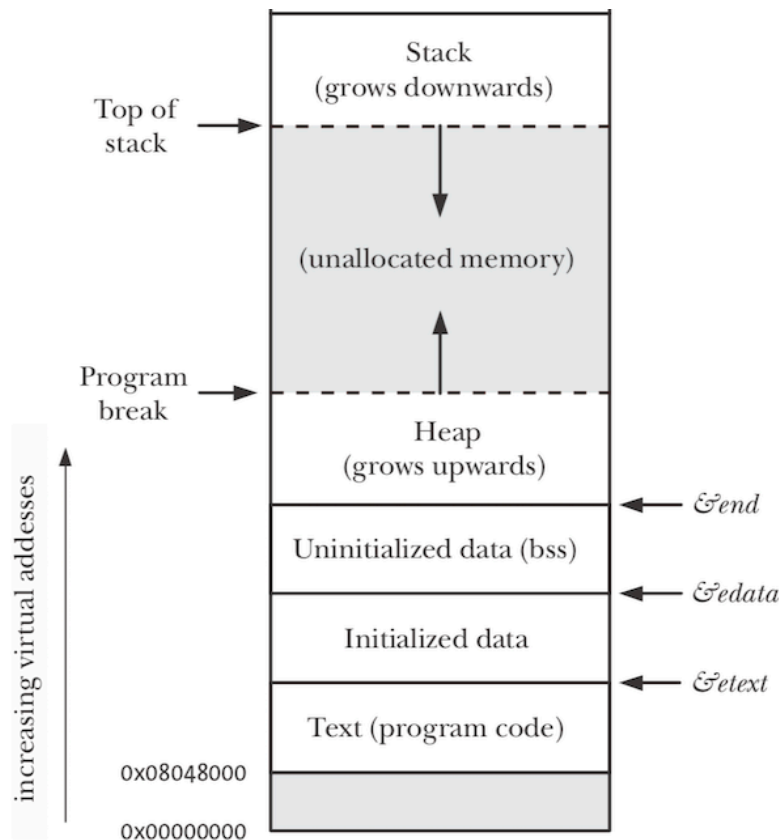
- *Segments*: logical divisions of a process's virtual memory
- Memory segments of a process: TEXT (readonly assembly code), DATA (initialised global and static vars), BSS (uninitialised global and static vars), HEAP (dyn. mem), STACK (containing stack frames, each one allocated for each currently called function). They are partitioned in pages.

- \$ size out

```
__TEXT __DATA__OBJC  others
dec hex
4096 4096 0 4294971392
4294979584 100003000
```

- A *process* can have one or multiple *threads* of execution. All threads of a process share the same virtual memory (DATA and HEAP) and file descriptor table, but each of them has its own

STACK. Some algorithms transpose more naturally to a multithreaded implementation than to a multiprocess implementation



- Swap area: a reserved area of disk space used to supplement the computer's RAM
- On most file systems, file space is allocated in units of blocks. Block size is typically something like 1024, 2048, or 4096 bytes
- Every file has a read-write offset, the location in the file at which the next *read()* or *write()* will commence. This

offset is set to 0 when file is opened, than it can be moved using *lseek()*

- We say things such as "a process can create another process" and things like that. Remember, however, that the kernel mediates all such actions, and these statements are just shorthand for "a process can request that the kernel create another process, and so on"
- *Epoch*: Jan 1, 1970. This date is close to the birth of the UNIX system. *time\_t* is the standard data type to store the number of sec since *Epoch*
- *CPU time*: the total amount of CPU time that a process has used since starting. It is the sum of *system CPU time* (the time spent in *kernel mode*) and *user CPU time* (the time spent in program code)
- *realtime*: said of a OS which guarantees that the response is always delivered within a certain deadline time after the triggering event
- IPC and Synchronization methods: signals, pipes, sockets, file locking, message queues, semaphores, shared

memory

- Each process has a number of associated UIDs and GIDs
  - PID (process ID): identifies the process. *pid\_t* is the system data type to store a PID
  - PPID (parent process identifier): identifies the process that requested the kernel to create this process
  - UID (user ID): identifies the user. UID 0 is reserved to the *superuser*. *uid\_t* is the system data type to store a UID
  - GID (group ID): identifies the group the user belongs to

```
int gidsetsize = 2;
gid_t grouplist[2];
getgroups(gidsetsize, grouplist);

fprintf(stdout, "process ID: %d\n", getpid());
fprintf(stdout, "parent process ID: %d\n", getppid());
fprintf(stdout, "process group ID: %d\n", getpgrp());
```

- *superuser process*: a process with UID 0 (superuser) and login name *root*, which has all capabilities. For example *init* is the superuser process with PID 1 first created at system boots. *init* cannot be killed. All processes will descend from it.

- *daemon*: a long-lived process which runs in the background. Examples are *syslogd* and *httpd*
- GCC (GNU Compiler Collection)
- *race condition*: a situation where the result produced by two processes (or threads) operating on shared resources depends in an unexpected way on the relative order in which the processes gain access to the CPU(s)
- ABI (*Application Binary Interface*): a set of rules specifying how a binary executable should exchange information with some service (e.g., the kernel or a library) at run time. Among other things, an ABI specifies which registers and stack locations are used to exchange this information, and what meaning is attached to the exchanged values. Once compiled for a particular ABI, a binary executable should be able to run on any system presenting the same ABI

## Environment

- environment variable: a name-value pair

stored in user-space memory. Notable examples are USER, GROUPS, HOME, PATH, PID, PPID. Can be created with *export* in most shells, *setenv* in C shell

- environment list: the set of environment variables for a process
- When a new process is created via *fork()*, it inherits a copy of the calling process's environment. When a new process is created via *fork()*, it inherits the calling process's UID and GID
- In this example we read an environment variable

```
void checkvar()
{
    char *s;
    s = getenv("HOME");
    if (s != NULL && *s != '\0')
        fprintf(stdout, "HOME is %s\n", s);
    else
        abort();
}
```

## System Calls

- System calls allow processes to request services from the kernel.
- *glibc* is the Linux C standard library
- Each system call (e.g. *execve*) has a unique number (11). The user mode

application program doesn't call directly the system call. It calls the corresponding *glibc* wrapper function *execve()*. The kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number into a specific CPU register (*%eax* for *x86\_32*), then executes the trap instruction *int 0x80*, which causes the processor to switch to kernel mode with the execution of the architecture dependent *system\_call()* in *arch/i386/entry.S*, which in turn will call the appropriate service routine. In the case of an error, the wrapper function sets the global *int* variable *errno* to a positive value

- Not all of the *glibc* functions are system calls. Some have nothing to do with system call (e.g *strlen*), some others contain a system call, e.g. *brk()* inside *malloc*, or *open()* inside *fopen*
- System calls are executed atomically, i.e. as a single uninterruptible step
- void **perror**(const char \**msg*): prints the



user's msg followed by the standard message associated with *errno*

- *pid\_t getpid(void)*: system call which returns the process ID of the calling process
- *pid\_t getppid(void)*: system call which returns the process ID of the parent
- On 32-bit platforms, the maximum value for PID is 32768, but on 64-bit platforms, it can be adjusted to any value up to  $2^{22}$
- If a child process becomes orphaned because its parent terminates, then the child is adopted by the *init* process, and subsequent calls to *getppid()* in the child return 1
- *pid\_t fork()*: system call used by a process (*parent*) to create another process (*child*). The return value can be <0 (fail), =0 (for the child), >0 (for the parent). Testing the return value of a fork is the correct way to differentiate parent and child source code execution

```

p = fork();
//all statements below this line are executed by parent and child
if (p < 0) {
    fprintf(stderr, "fork Failed");
    return 1;
} else if (p > 0) { // Code for parent process
    ...
} else { // Code for child process
    ...
}

```

- *wait()*: system call used by the parent to know the termination status of one (or any) of its children. The execution is stopped until the signal is received
- *exit(status)*: used by a process to terminate itself with an explicit termination status. By convention, we set status = 0 to indicate that the process succeeded, while a nonzero status indicates that some error occurred. Most shells make the termination status of the last executed program available via a shell variable named `?`
- *kill(pid, sig)*: sends a signal *sig* to a process *pid*. If *sig*=0, the return value tells if *pid* is valid (or the process is still alive)

```

kill(pid, SIGKILL); // killing a process
if (kill(pid, 0) == 0) {
    printf("the process is still alive\n");
}

```

Typically, *sig* will be one of the signals specified in `sigaction(2)`. A value of 0, however, will cause error checking to be performed (with no signal being sent), which can be used to check the validity of *pid*.

- *interpreted file*: a data file beginning with `#! <interpreter>`. For example a shell script begins with `#! /bin/bash`
- `execve()`: transforms the calling process into a new process based on the specified interpreted file or executable object file, loading and running the executable (or the interpreter). The calling process's segments are released. The executable inherits most of the calling process's environment list, like PID and PPID
- `system()`: è una funzione che esegue in cascata `fork()`, `execve()`, `wait()`. Se non stiamo parlando di eseguibili esterni ma di funzioni di glibc o comandi bash, è preferibile usare le system call di base. P.es. invece di fare `system("echo 1 > filepath")` conviene aprire *filepath* con

*open* e *scriverci* con *write*

- *mmap()*: system call used by a process to map a file into an area of the process's virtual memory, either in private or shared mode. Memory Mapping can be used as IPC technique. If a process uses *mmap()* and then *fork()*, the child process inherits the shared memory
- *pipe()*: creates a pipe and returns two file descriptors referring to either end of the pipe
- *socket()*: creates a socket and returns a file descriptor referring to the socket

## Object Libraries

- *object library*: A set of related compiled object modules employed by programs linked against it. Can be *static* or *shared*.
- *static library* (or *archive*): it is linked to the application at build time. At any point a function call is resolved the linker copies the corresponding object module from the library into the executable file. This is memory inefficient (both for disk space and RAM). For example, if

multiple programs use the same library  
then multiple copies of the same code  
are resident in RAM

- *shared library*: the *static linker* just writes a record in the executable file where a library symbol is resolved. At runtime the *dynamic linker* ensures that the required object module is loaded in RAM and executed for that program and for all programs will need the same module.
- *soname*: the name of a shared library which may differ from the real name. It is used by the static linker to record the program executable. It provides a level of abstraction to avoid the static linking against a change of the library that maintains the API compatibility

## Signals

- Signals (SIGxxx) are a kind of software interrupts. Examples:  

#	Name	Default Action	Description
1	SIGHUP	terminate process	terminal

line hangup

2 SIGINT terminate process interrupt program

3 SIGQUIT create core image quit program

6 SIGABRT (abort)

9 SIGKILL terminate process kill program

14 ALRM (alarm clock)

15 SIGTERM terminate process software termination signal

- To see all signal names and numbers

```
for (i=1; i<32; i++)  
    printf("%s\n", strsignal(i));
```

- A received signal can either be blocked, ignored, or handled by a signal handler

```
static void mySigHandler(int sig)  
{  
    printf("adamo: I've dreamt of one of my son killed.\n");  
}  
  
int main()  
{  
    if (signal(SIGCHLD, mySigHandler) == SIG_ERR)  
        perror("Signal");  
}
```

- To send a signal *sig* to a process with *pid* use *kill(pid,sig)* system call
- SIGKILL: it cannot be ignored, blocked nor terminated by a handler. It kills the destination process

- Quando un processo termina, tutte le sue strutture vengono rimosse tranne le informazioni di stato, fino a che un segnale SIGCHLD raggiunge il processo parent che lo intercetta con la *wait()* oppure con un signal handler. A quel punto il processo, in stato di Zombie, è definitivamente distrutto
- *\$ fg* riporta in foreground l'ultimo processo che era stato messo in background
- *interrupt* character: when the user types Ctrl+C, a SIGINT is sent by the kernel to the foreground running command. The process is terminated and the parent is sent a SIGCHLD
- *suspend* character : when the user types Ctrl+Z, a SIGSTOP is sent by the kernel to the foreground running command. The process is suspended and the parent is sent a SIGCHLD
- A *background process group* (*background job*), can be created by terminating a command with &
- *pseudoterminal*: a pair of connected

virtual devices, known as the *master* and *slave*. This device pair provides an IPC channel. An example is ssh

## Example 1

- In this program we use *getpid()*, *getppid()*, *fork()*, *wait()*, *exit()*, *pipe()*  
To compile and execute type  
*\$ gcc -Wall -Wextra -g -o out main.c*  
To execute  
*\$ ./out*



```

main.c  x  get_num.h  x  error_functions.h  x  tlp_i_hdr.h
1  #include "tlpi_hdr.h"
2  int main()
3  {
4      int fd1[2]; // Used to store two ends of first pipe
5      int fd2[2]; // Used to store two ends of second pipe
6      char fixed_str[] = "fromchild";
7      char msg_str[100];
8      char input_str[100];
9      pid_t p;
10
11     if (pipe(fd1)==-1) {
12         fprintf(stderr, "Pipe Failed");
13         return 1;
14     }
15     if (pipe(fd2)==-1) {
16         fprintf(stderr, "Pipe Failed");
17         return 1;
18     }
19     fprintf(stdout, "parent: please enter a message for child (try 'kill')\n");
20     scanf("%s", input_str);
21     p = fork();
22     //all statements below this line are executed by parent and child
23     if (p < 0) {
24         fprintf(stderr, "fork Failed" );
25         return 1;
26     } else if (p > 0) { // Code for parent process
27         fprintf(stdout, "parent: fork return is %d\n", p);
28         fprintf(stdout, "parent: my PID is %d\n", getpid());
29         if (0 == strcmp(input_str, "kill\0")) {
30             fprintf(stdout, "parent: killing child..\n");
31             kill(p, SIGTERM);
32         }
33
34         close(fd1[0]); // Close reading end of first pipe
35         // Write input string and close writing end of first pipe.
36         write(fd1[1], input_str, strlen(input_str)+1);
37         close(fd1[1]);
38         wait(NULL); // Wait for child to exit (or be killed)
39         close(fd2[1]); // Close writing end of second pipe
40         read(fd2[0], msg_str, 100); // Read string from child, print it and close reading end
41         fprintf(stdout, "parent: received from child.. %s\n", msg_str);
42         close(fd2[0]);
43     } else { // Code for child process
44         fprintf(stdout, "child: fork return is %d\n", p);
45         fprintf(stdout, "child: my PPID is %d\n", getppid());
46         fprintf(stdout, "child: my PID is %d\n", getpid());
47         close(fd1[1]); // Close writing end of first pipe
48         read(fd1[0], msg_str, 100); // Read a string using first pipe
49         close(fd1[0]);
50         // Concatenate a fixed string with it
51         int k = strlen(msg_str);
52         int i;
53         for (i=0; i<strlen(fixed_str); i++)
54             msg_str[k++] = fixed_str[i];
55         msg_str[k] = '\0'; // string ends with '\0'
56         close(fd2[0]);
57         // Write concatenated string and close writing end
58         write(fd2[1], msg_str, strlen(msg_str)+1);
59         close(fd2[1]);
60         exit(0);
61     }
62 }

```

- *strace*: command used to trace the system calls made by a program, either for debugging purposes or simply to

investigate what a program is doing.

- *fprintf()*: prints to file, like the standard streams

*fprintf(stderr, "Impossibile continuare!  
|n"); //prints error messages*

*fprintf(stdout, "Operazione completata!  
|n"); //prints a message*

- *fprintf()*: gets string from keyboard  
*fscanf(stdin, "%s", mystring); //gets a  
string from keyboard and stores it in  
mystring*

- To specify the man page for a given command we use to put in parenthesis, e.g. *ldconfig(8)*. The shell command is *\$ man 8 ldconfig*  
page 1 explains shell commands, page 2 explains system calls, etc..

## Shell

- On UNIX systems, the shell is a user process. On other systems it is part of the kernel
- Bourne Shell (sh), C shell (csh), Bourne again shell (bash)

- Shells are designed not merely for interactive use, but also for the interpretation of *shell scripts*
- An interactive shell run under a terminal

## I/O system model

- The file descriptor is the file identifier
- Most programs expect to be able to use the three standard file descriptors *stdin*, *stdout*, *stderr*
- `int open(const char *pathname, int flags, mode_t mode);`

```
int fd;
char path[200] = "testfile";
fd = open(path, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
if (fd == -1) {
    perror("open");
    goto err;
}
```

- `ssize_t read(int fd, void *buffer, size_t count);`

```
ssize_t len;
char content_str[100];
len = read(fd, content_str, 15);
if (len < 0) {
    perror("read");
    goto err;
}
```

- `ssize_t write(int fd, void *buffer, size_t count);`
- `status = close(fd)`
- `off_t lseek(int fd, off_t offset, int`

*whence*) returns the new file internal read-write offset ( $\geq 0$ ) after a move of *offset* ( $\geq < 0$ ) bytes with respect to the start of the file (*whence* = SEEK\_SET), the end of the file (*whence* = SEEK\_END) or the current offset (*whence* = SEEK\_CUR)

```
printf("first byte offset = %ld\n", (long)lseek(fd, 0, SEEK_SET));  
printf("last byte offset = %ld\n", (long)lseek(fd, -1, SEEK_END));
```

- int **fcntl**(int *fd*, int *cmd*, ...) performs a number of operation over the file, depending on *cmd*
- For example, to retrieve the status flag of a file

```
int flags;  
flags = fcntl(fd, F_GETFL);  
if (flags == -1) {  
    perror("fcntl");  
    goto err;  
}  
int accessMode = flags & O_ACCMODE;  
if (accessMode == O_WRONLY || accessMode == O_RDWR)  
    printf("file is writable\n");
```

- The pread() and pwrite() system calls operate just like read() and write(), except that the file I/O is performed at the location specified by offset
- There are three data structures maintained by the kernel:
  - the per-process file descriptor table

- the system-wide table of open file descriptions
- the file system i-node table
- **int ftruncate**(int *fd*, off\_t *length*)  
truncates the file to *length* bytes. *length* can either be lower or greater than the file size. In the first case data is lost, in the second case zero padding occurs. There is also a version **int truncate**(const char \**pathname*, off\_t *length*)
- off\_t is signed long. On 32-bit architectures this would limit the size of files to  $2^{31}-1$  bytes (i.e., 2GB).
- with `#define _FILE_OFFSET_BITS 64`, `open()`, `read()`, etc. can operate on files larger than 2GB
- compiling with flag option - `D_FILE_OFFSET_BITS=64` is equivalent to use `#define _FILE_OFFSET_BITS 64` in the source file
- For each process, the kernel provides the special virtual directory `/dev/fd`. This directory contains filenames of the form `/dev/fd/n`, where *n* is a number

corresponding to one of the open file descriptors for the process. Thus, for example, `/dev/fd/0` is standard input for the process

