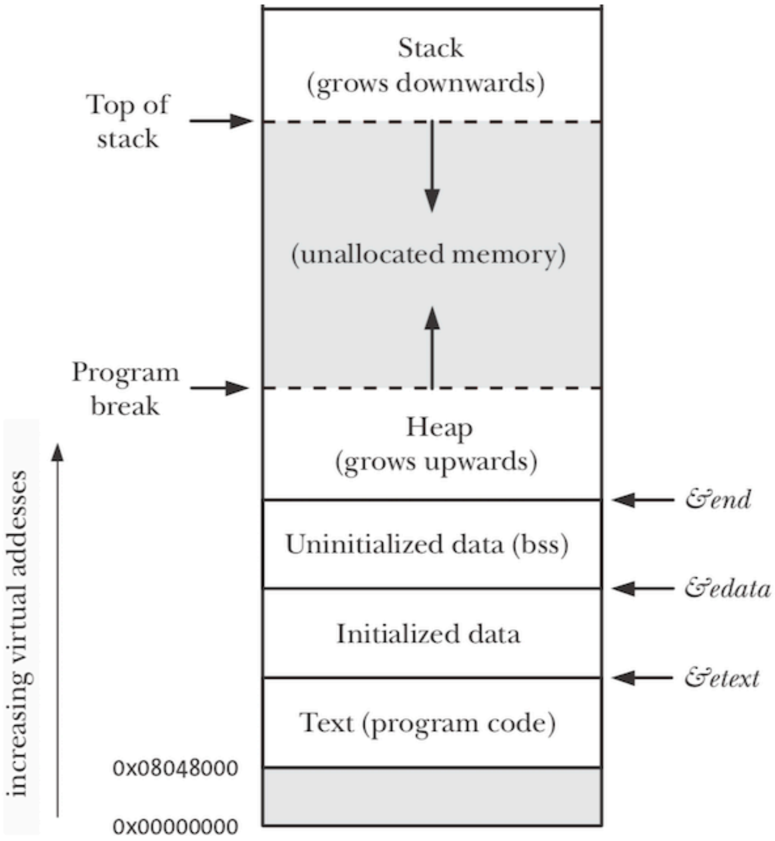
Linux System Programming

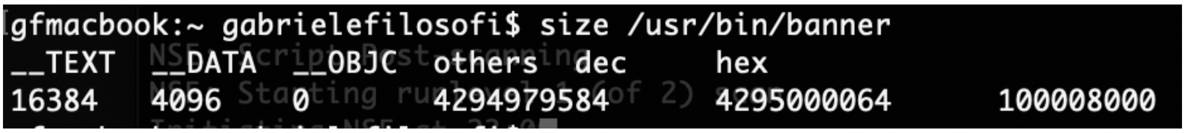
G. Filosofi 2019

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
- 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
- *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, such as 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 executing function). All segments are partitioned in page units

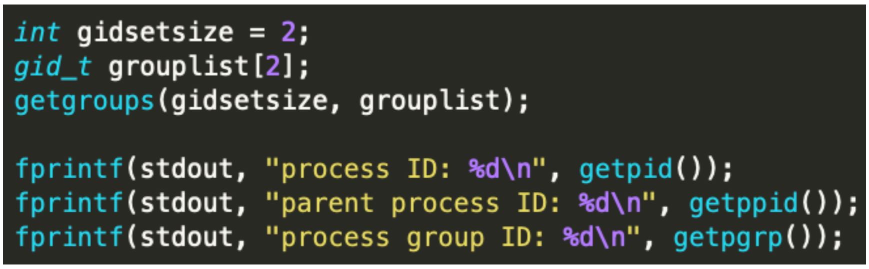


- $ size <prog>



```
gfmacbook:~ gabrielefilosofi$ size /usr/bin/banner
__TEXT  __DATA  __OBJC  others      dec         hex
16384   4096    0       4294979584  4295000064         100008000
```

- 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()*
- *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)
- $ times <PID>
- *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 integers
  - PID (process identifier): identifies the process. *pid_t getpid(void)* returns PID of the calling process
  - PPID (parent PID): identifies the process that requested the kernel to create this process. *pid_t getppid(void)* returns PID of the parent of the calling process
  - UID (user ID): it identifies the user. 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. Getpgrp...



```
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());
```

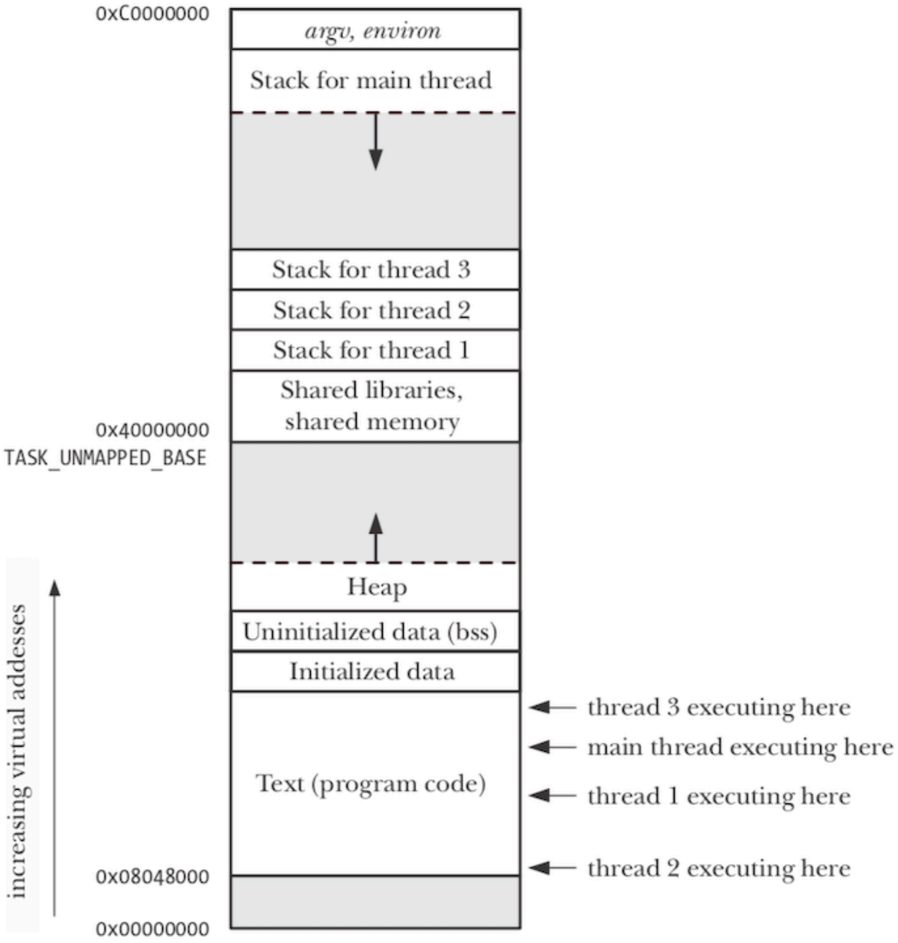
- *superuser process*: a process with UID 0 and login name *root*, which has all capabilities. For example *init* is the superuser process with PID 1, first created at system boot. *init* cannot be killed. All processes will descend from it.
- *deamon*: a long-lived process which runs in the background. 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
- bootstrap: the process of self-starting a computer, specifically with regard to its OS. It is supposed to proceed without external input, an effort analogous to the impossible task of one who would lift himself by his own bootstraps (with same sense of italian saying "sollevarsi da terra tirandosi per i capelli"). The computer term bootstrap began as a

metaphor in the 1950s

- reentrant code, reentrancy: a program or routine that is written so that the same copy in memory can be invoked safely by multiple users concurrently. It differs from thread-safety
- blocking call: control returns only when the call completes
- non-blocking call: control returns immediately. Later OS somehow notifies the process that the call is complete
- Synchronous program: A program which uses blocking calls. In order not to freeze during the call it must have 2 or more threads
- Asynchronous program: A program which uses non-blocking calls. It can have only 1 thread and still remain interactive

Threads

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



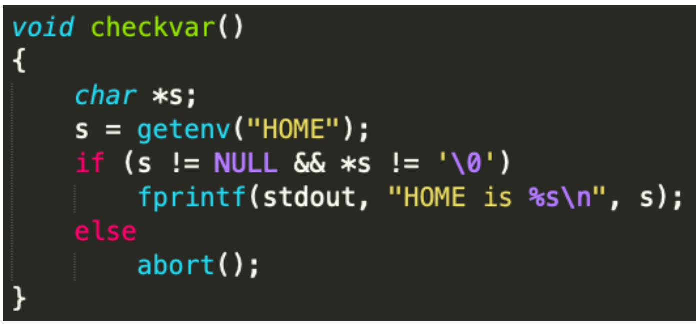
- Pthread: POSIX thread
- A process can create a thread with *clone()*. More commonly we use pthread_create(), which calls *clone()*
- Multiple threads of a process can execute concurrently in memory. On a multiprocessor system, they can execute in parallel over time. In any this will increase efficiency, because if one thread is blocked on I/O, or for a cache miss, other threads are still eligible to execute
- Each thread has a TID (thread ID). *int gettid(void)* returns the thread ID of the calling thread. For a POSIX thread (Pthread) there is a different function, *pthread_self()*.
- In the Linux threading implementations, thread IDs are unique across processes
- To avoid race conditions threads do not use the global *errno* variable. By including

<errno.h> a thread will have its own *errno* variable

- Programs that use the Pthreads API must be compiled with the *cc –pthread* option
- When gets started, the resulting process consists of a single thread, called the *initial* or *main* thread
- To create additional threads use *pthread_create()* which returns the TID
- *pthread_join()*: waits for the thread identified by *thread* to terminate (If that thread has already terminated, *pthread_join* returns immediately) This operation is termed *joining*
- Threads are peers. Any thread in a process can use *pthread_join()* to join with any other thread in the process. For example, if thread A creates thread B, which creates thread C, then it is possible for thread A to join with thread C, or vice versa. This differs from the hierarchical relationship between processes. When a parent process creates a child using *fork()*, it is the only process that can *wait()* on that child.
- Some algorithms transpose more naturally to a multithreaded rather than multiprocess implementation. As an example consider a network server program for social networking. Allocating a new process for any incoming client connection is not a good idea. Client to client communication will require IPC. Data exchange between threads is simple and fast because they already share memory. Furthermore, *clone* in 10 times less expensive then *fork*. In this case multithreaded concurrency is preferable than multiprocess concurrency.

Environment

- environment variable: a name-value pair stored in user-space memory. Notable examples are USER, UID, GROUPS, HOME, PATH, PID, PPID, SHELL. 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();
}
```

- *USER and UID* are the current user login name and user ID; GROUPS is the group ID; HOME is the login folder; SHALL is the name of program that gets control after login
- *$ echo $<varname>* displays the value of *<varname>*

System Calls

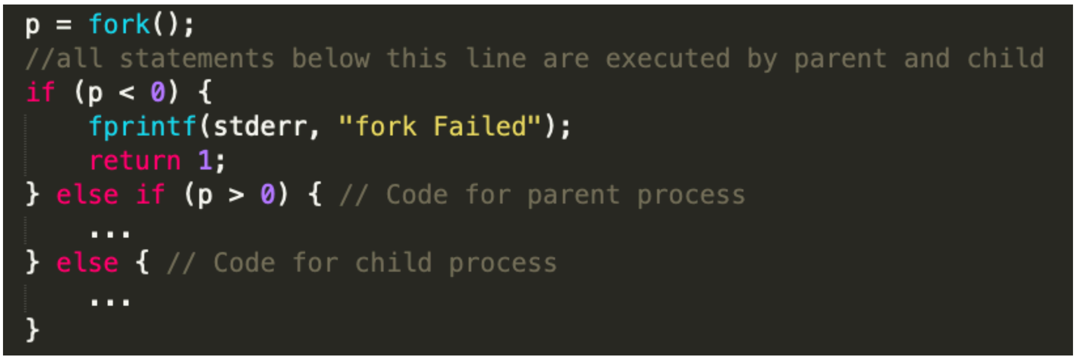
- System calls allow processes to request services from the kernel
- *glibc* is the Linux C standard library
- Each system call (e.g. *execve*) has a unique number (11). The user mode application program doesn't call directly the system call. It calls the corresponding *glibc* wrapper function *execve()*. 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
- The traditional method of returning status from system calls and some library functions

is to return 0 on success and –1 on error, with *errno* being set to indicate the error

- 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

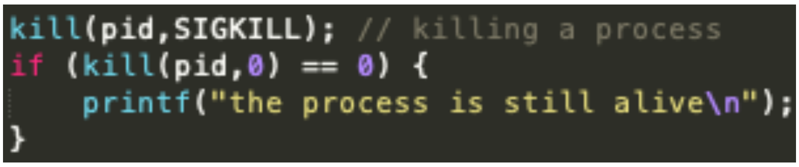
Examples

- *void perror(const char *msg)*: prints the user's msg followed by the standard message associated with current errno value
- 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

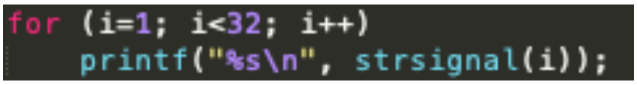
- *object library*: A set of related compiled object modules employed by programs linked against it. Can be *static* or *shared*.
- *static library* (or *archive*): it is linked to the application at build time. At any point a function call is resolved the linker copies the corresponding object module 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:

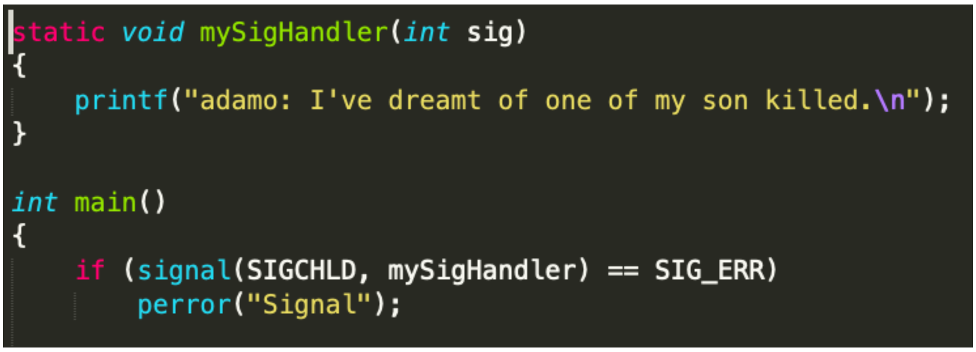
#	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
6	SIGABRT	(abort)	
9	SIGKILL	terminate process	kill program
14	ALRM	(alarm clock)	
15	SIGTERM	terminate process	software termination 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");
```

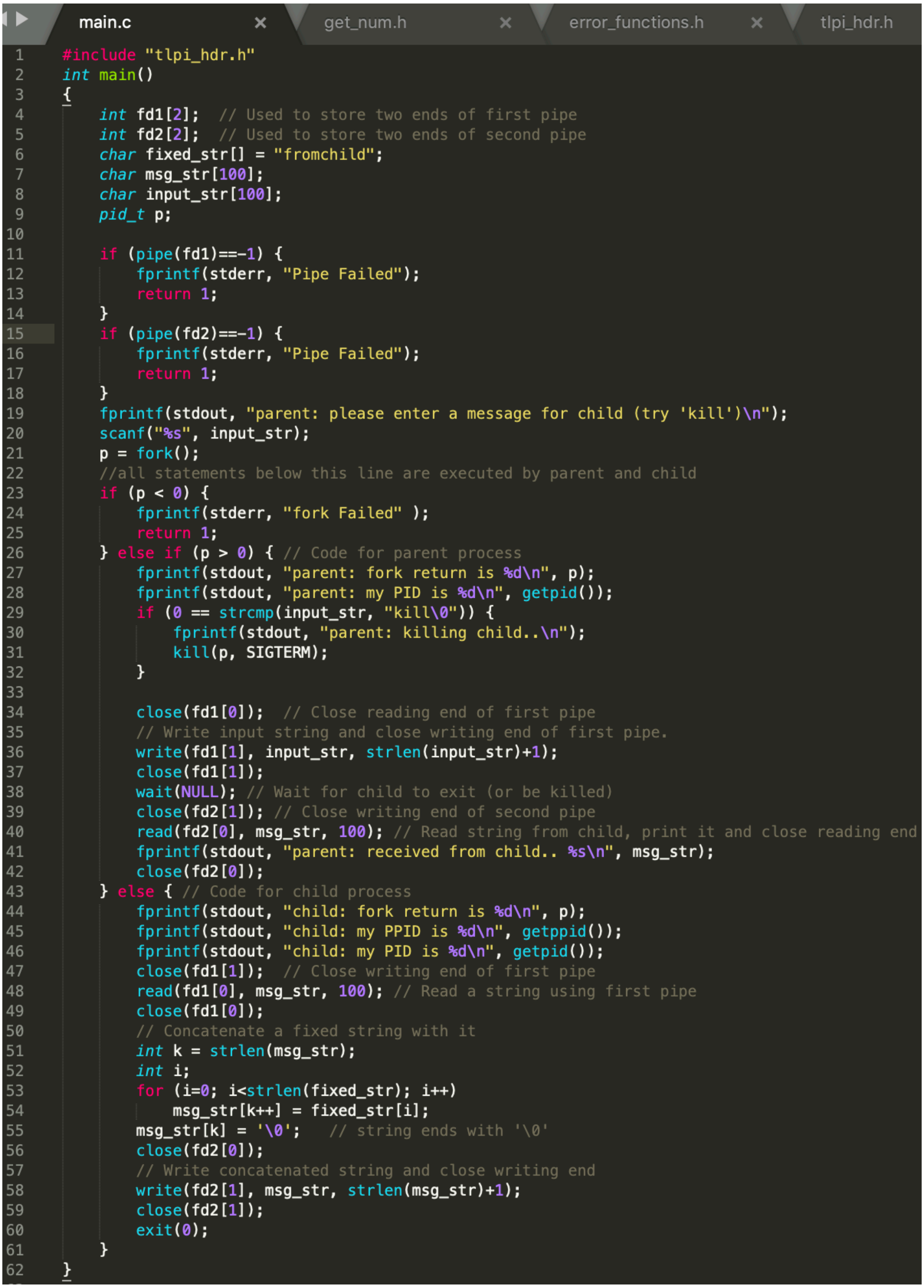
- To send a signal *sig* to a process with *pid* use *kill(pid,sig)* system call
- SIGKILL: it cannot be ignored, blocked nor terminated by a handler. It kills the destination process
- Quando un processo termina, tutte le sue strutture vengono rimosse tranne le informazioni di stato, fino a che un segnale SIGCHILD 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()*



```c
#include "tlpi_hdr.h"
int main()
{
    int fd1[2];  // Used to store two ends of first pipe
    int fd2[2];  // Used to store two ends of second pipe
    char fixed_str[] = "fromchild";
    char msg_str[100];
    char input_str[100];
    pid_t p;

    if (pipe(fd1)==-1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }
    if (pipe(fd2)==-1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }
    fprintf(stdout, "parent: please enter a message for child (try 'kill')\n");
    scanf("%s", input_str);
    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
        fprintf(stdout, "parent: fork return is %d\n", p);
        fprintf(stdout, "parent: my PID is %d\n", getpid());
        if (0 == strcmp(input_str, "kill\0")) {
            fprintf(stdout, "parent: killing child..\n");
            kill(p, SIGTERM);
        }

        close(fd1[0]);  // Close reading end of first pipe
        // Write input string and close writing end of first pipe.
        write(fd1[1], input_str, strlen(input_str)+1);
        close(fd1[1]);
        wait(NULL); // Wait for child to exit (or be killed)
        close(fd2[1]); // Close writing end of second pipe
        read(fd2[0], msg_str, 100); // Read string from child, print it and close reading end
        fprintf(stdout, "parent: received from child.. %s\n", msg_str);
        close(fd2[0]);
    } else { // Code for child process
        fprintf(stdout, "child: fork return is %d\n", p);
        fprintf(stdout, "child: my PPID is %d\n", getppid());
        fprintf(stdout, "child: my PID is %d\n", getpid());
        close(fd1[1]);  // Close writing end of first pipe
        read(fd1[0], msg_str, 100); // Read a string using first pipe
        close(fd1[0]);
        // Concatenate a fixed string with it
        int k = strlen(msg_str);
        int i;
        for (i=0; i<strlen(fixed_str); i++)
            msg_str[k++] = fixed_str[i];
        msg_str[k] = '\0';   // string ends with '\0'
        close(fd2[0]);
        // Write concatenated string and close writing end
        write(fd2[1], msg_str, strlen(msg_str)+1);
        close(fd2[1]);
        exit(0);
    }
}
```

COMPILE: $ gcc -Wall -Wextra -g -o out main.c

- *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

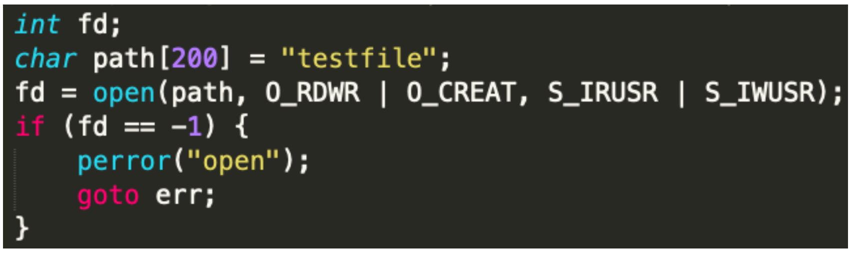
- To specify the man page for a given command we use to put in parenthesis, e.g. ldconfig(8). The shell command is
  $ man 8 ldconfig
  page 1 explains shell commands, page 2 explains system calls, etc..

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

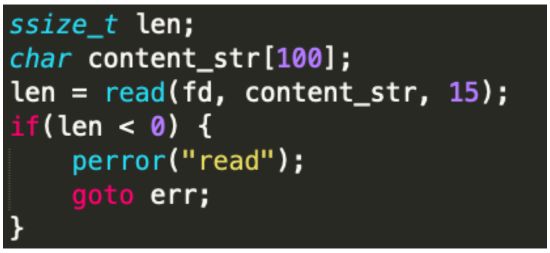
I/O system model

- 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*);



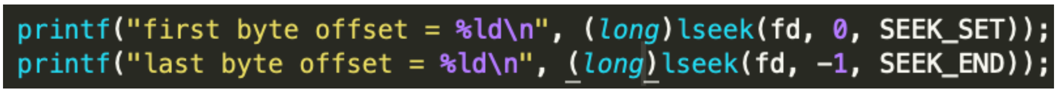
```
int fd;
char path[200] = "testfile";
fd = open(path, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
if (fd == -1) {
    perror("open");
    goto err;
}
```

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



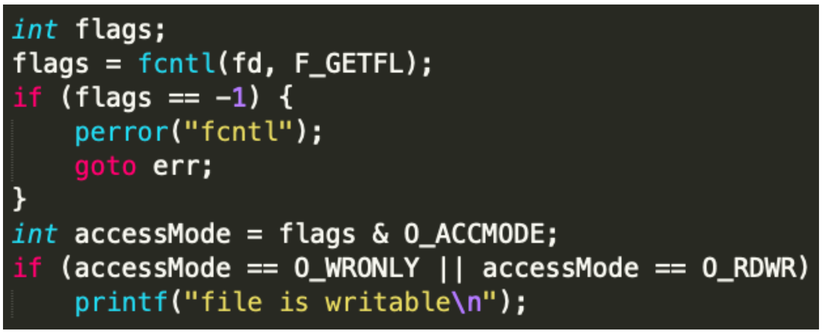
```
ssize_t len;
char content_str[100];
len = read(fd, content_str, 15);
if(len < 0) {
    perror("read");
    goto err;
}
```

- 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 (>=0) after a move of *offset* (>=<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



```c
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");
```

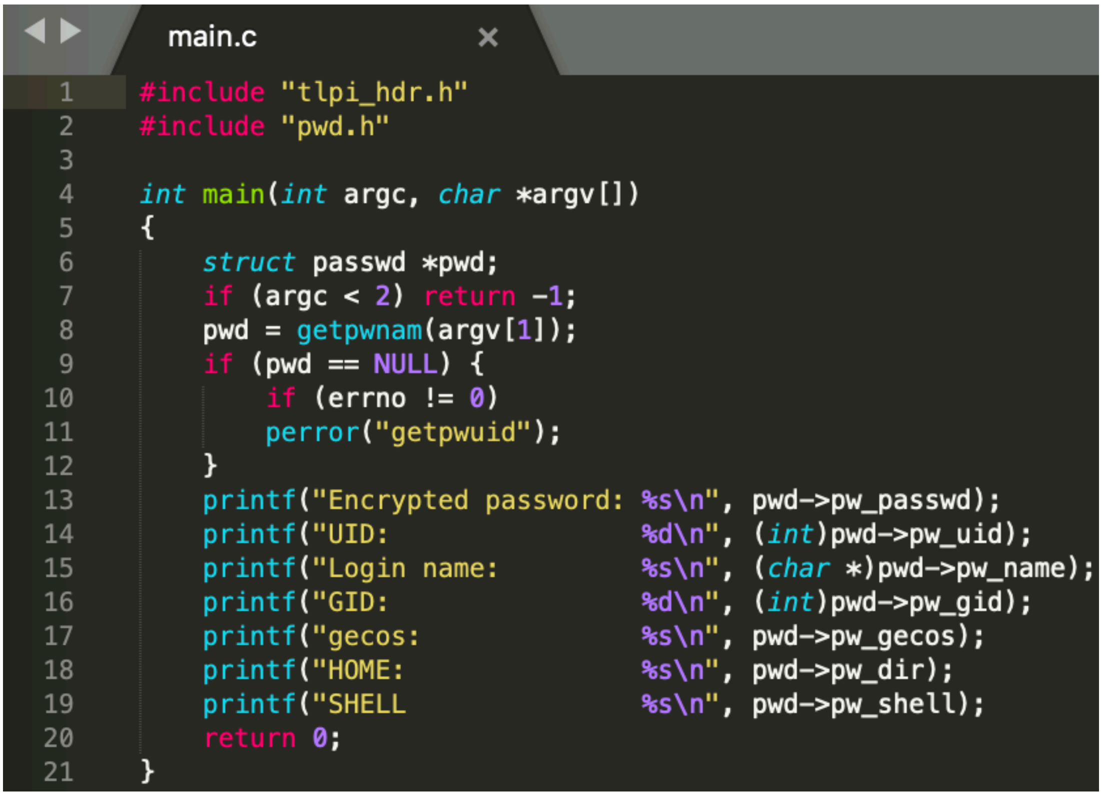
- The pread() and pwrite() system calls operate just like read() and write(), except that the file I/O is performed at the location specified by offset
- There are three data structures maintained by the kernel:
    - the per-process file descriptor table
    - the system-wide table of open file descriptions
    - 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 seconda 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 231–1 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

Users and Groups

- Users are organised in groups. Multi-group membership is allowed
- Users and groups have unique IDs and names
- *$ finger <username>* displays user's information and login time if she is currently logged
- On my macBook I have two users with login name *Guest* (UID: 201) and *gabrielefilosofi* (UID: 501). The initial groups are *_guest* (GID:) and *staff* (GID: 201), respectively
- */etc/group* contains the list of groups, with the UIDs belonging to each of them
- *$ newgrp <groupname>* allows you to get assigned a different initial group. Password may be necessary
- *$ groups <username>* or *$ id -p <username>* give the list of groups *<username>* belongs to

Example 1

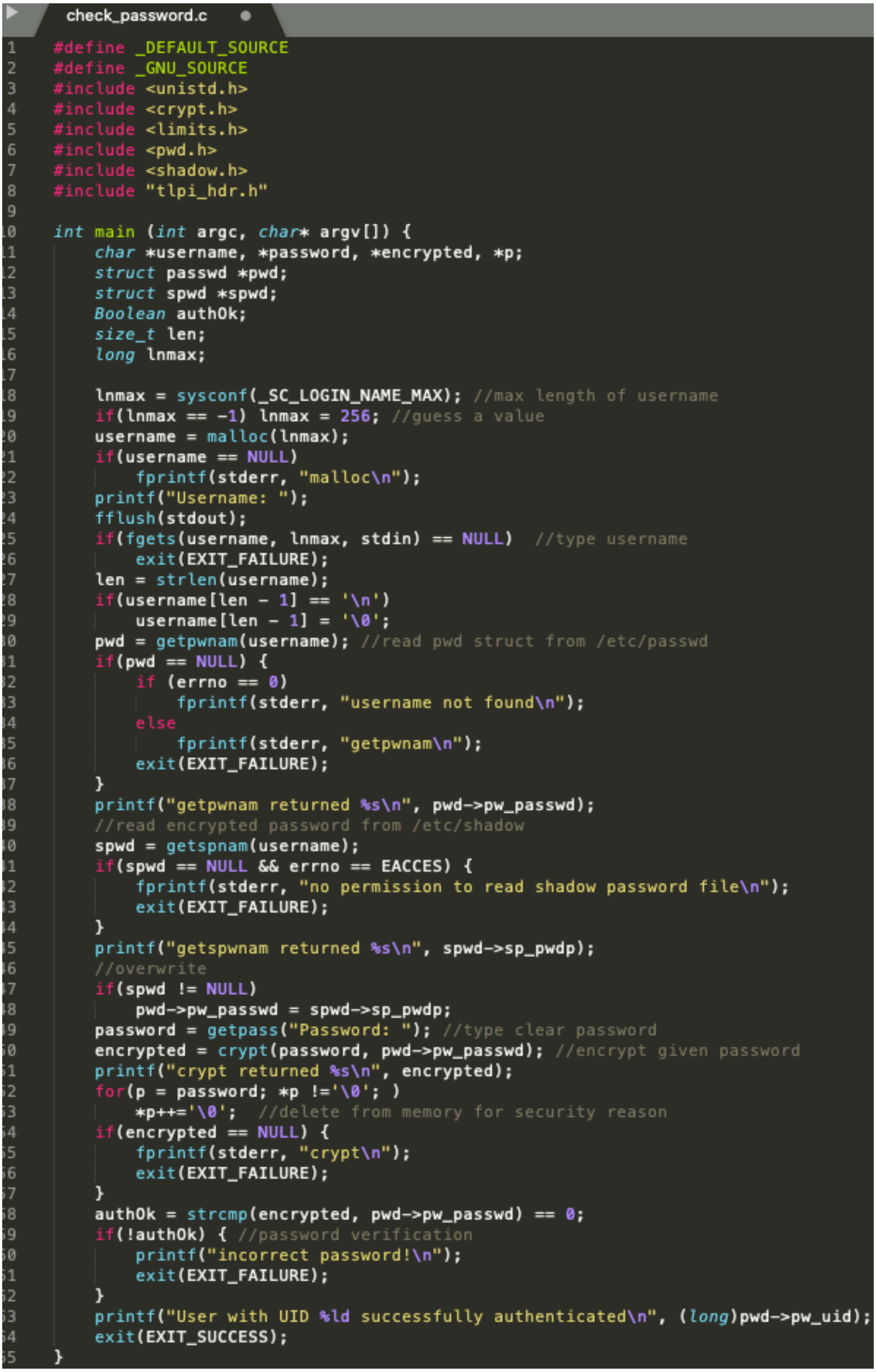
- For example, to display the user and group information from the login name



```c
#include "tlpi_hdr.h"
#include "pwd.h"

int main(int argc, char *argv[])
{
    struct passwd *pwd;
    if (argc < 2) return -1;
    pwd = getpwnam(argv[1]);
    if (pwd == NULL) {
        if (errno != 0)
        perror("getpwuid");
    }
    printf("Encrypted password: %s\n", pwd->pw_passwd);
    printf("UID:                %d\n", (int)pwd->pw_uid);
    printf("Login name:         %s\n", (char *)pwd->pw_name);
    printf("GID:                %d\n", (int)pwd->pw_gid);
    printf("gecos:              %s\n", pwd->pw_gecos);
    printf("HOME:               %s\n", pwd->pw_dir);
    printf("SHELL               %s\n", pwd->pw_shell);
    return 0;
}
```

- UNIX encrypts passwords using a *one-way encryption* algorithm
- */etc/passwd* contains a line for each user with login name, GID, home directory, login shell program, etc. This file is accessible to anyone. The second field is the encrypted password, or an empty string if no password exists, or a single char (typically x or *) if the encrypted password is stored elsewhere for security reasons, e.g. in */etc/shadow*
- struct passwd ***getpwnam**(const char *name) gets the username and returns the struct with related info from /etc/passwd
- macOs handles system user information differently. For example there is no */etc/shadow*
- char ***crypt**(const char **key*, const char **salt*) gets the password in clear (key) plus additional 2-char string (salt), and returns a pointer to a statically allocated string containing the printable ASCII encrypted password. Early UNIX used DES encryption, with 13 characters hash images.
  For example, a clear password "b3c6d787acb" with a random salt "fZ" would result in encoded password "fZI4JWFLZV8S2". Or password "toor" with a random salt "X0" would result in encoded password "X014elvznJq7E"
- In this program we use getpwnam(), getspnam(), getpass(), crypt() to verify username and password input by user

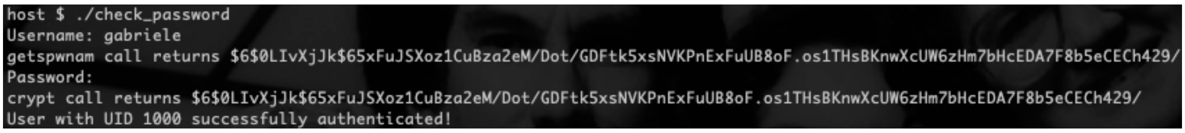


```c
#define _DEFAULT_SOURCE
#define _GNU_SOURCE
#include <unistd.h>
#include <crypt.h>
#include <limits.h>
#include <pwd.h>
#include <shadow.h>
#include "tlpi_hdr.h"

int main (int argc, char* argv[]) {
    char *username, *password, *encrypted, *p;
    struct passwd *pwd;
    struct spwd *spwd;
    Boolean authOk;
    size_t len;
    long lnmax;

    lnmax = sysconf(_SC_LOGIN_NAME_MAX); //max length of username
    if(lnmax == -1) lnmax = 256; //guess a value
    username = malloc(lnmax);
    if(username == NULL)
        fprintf(stderr, "malloc\n");
    printf("Username: ");
    fflush(stdout);
    if(fgets(username, lnmax, stdin) == NULL)  //type username
        exit(EXIT_FAILURE);
    len = strlen(username);
    if(username[len - 1] == '\n')
        username[len - 1] = '\0';
    pwd = getpwnam(username); //read pwd struct from /etc/passwd
    if(pwd == NULL) {
        if (errno == 0)
            fprintf(stderr, "username not found\n");
        else
            fprintf(stderr, "getpwnam\n");
        exit(EXIT_FAILURE);
    }
    printf("getpwnam returned %s\n", pwd->pw_passwd);
    //read encrypted password from /etc/shadow
    spwd = getspnam(username);
    if(spwd == NULL && errno == EACCES) {
        fprintf(stderr, "no permission to read shadow password file\n");
        exit(EXIT_FAILURE);
    }
    printf("getspwnam returned %s\n", spwd->sp_pwdp);
    //overwrite
    if(spwd != NULL)
        pwd->pw_passwd = spwd->sp_pwdp;
    password = getpass("Password: "); //type clear password
    encrypted = crypt(password, pwd->pw_passwd); //encrypt given password
    printf("crypt returned %s\n", encrypted);
    for(p = password; *p !='\0'; )
        *p++='\0';  //delete from memory for security reason
    if(encrypted == NULL) {
        fprintf(stderr, "crypt\n");
        exit(EXIT_FAILURE);
    }
    authOk = strcmp(encrypted, pwd->pw_passwd) == 0;
    if(!authOk) { //password verification
        printf("incorrect password!\n");
        exit(EXIT_FAILURE);
    }
    printf("User with UID %ld successfully authenticated\n", (long)pwd->pw_uid);
    exit(EXIT_SUCCESS);
}
```

COMPILE: *host $ gcc -Wall -Wextra -o check_password check_password.c -lcrypt*



```
host $ ./check_password
Username: gabriele
getspwnam call returns $6$0LIvXjJk$65xFuJSXoz1CuBza2eM/Dot/GDFtk5xsNVKPnExFuUB8oF.os1THsBKnwXcUW6zHm7bHcEDA7F8b5eCECh429/
Password:
crypt call returns $6$0LIvXjJk$65xFuJSXoz1CuBza2eM/Dot/GDFtk5xsNVKPnExFuUB8oF.os1THsBKnwXcUW6zHm7bHcEDA7F8b5eCECh429/
User with UID 1000 successfully authenticated!
```

Example 2
Creiamo un gruppo “hackers” con GID 200, e all’interno due nuovi utenti, l’utente “hector” (UID 201), login folder */home/hackers/monsegur* e password “sabu”, e l’utente “kevin” (UID 202), login folder */home/hackers/mitnick*
e password “condor”

```
root@kali:~# adduser --system --uid 201 --shell /bin/zsh --home /home/hackers/monsegur
--ingroup hackers hector
Adding system user `hector' (UID 201) ...
Adding new user `hector' (UID 201) with group `hackers' ...
Creating home directory `/home/hackers/monsegur' ...
root@kali:~# passwd hector
New password:
Retype new password:
passwd: password updated
successfully
root@kali:~# adduser --system
--uid 202 --shell /bin/sh --home /home/hackers/mitnick --ingroup hackers kevin
Adding system user `kevin' (UID 202) ...
Adding new user `kevin' (UID 202) with group `hackers' ...
Creating home directory `/home/hackers/mitnick'
...
root@kali:~# passwd kevin
New password:
Retype new password:
passwd: password updated
successfully
```

In */etc/group* trovo che è stata creata la linea relativa al nuovo gruppo

```
hackers:x:200:
```

In */etc/passwd* sono state create le linee relative ai due nuovi utenti

```
hector:x:201:200::/home/hackers/monsegur:/bin/zsh
kevin:x:202:200::/home/hackers/mitnick:/bin/sh
```

In */etc/shadow* sono state create le linee

```
hector:
$6$bAxxbKCjCkeSHtFq$QWLYapItFS5VTwPkTg3hz6JAka69S.fUOagJKKRKFJo0342Ce8igbzI
hXDTZNMSBP3/GpDYuq.h2hhzbJdI8W.:18122:0:99999:7:::
kevin:$6$EEdjsgb5KEGfGdGn$evpvzxVgZD7HAprfEA9981hi0U3TFnHCvR8yWnVUMYHjH/
AGzB8JkinlCucCwW1qz8m/WMIPazdASxrXhuq0V0:18122:0:99999:7:::
```

/proc

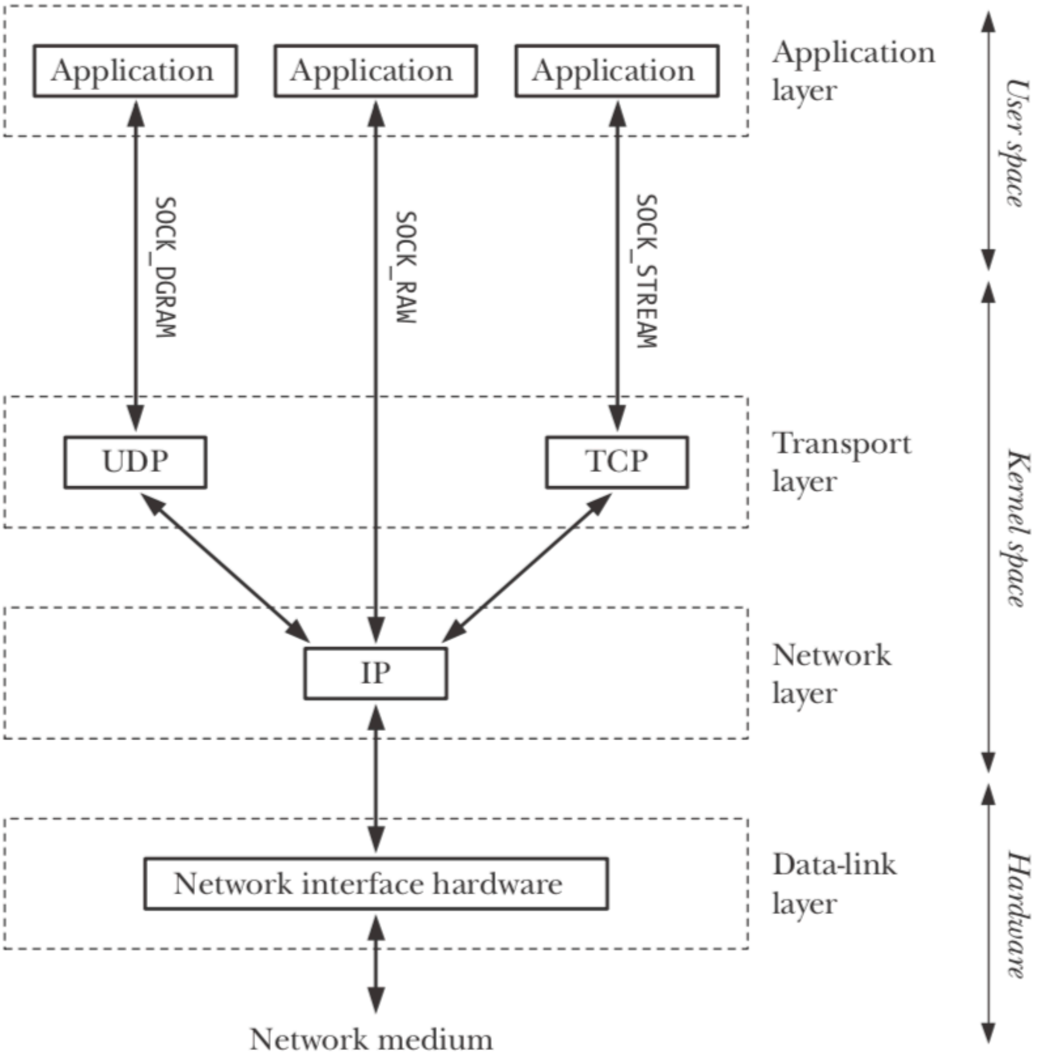
- The */proc* file system is a virtual file system containing some system and process-specific informations.
  $ echo 10000 > /proc/sys/kernel/pid_max
  $ cat /proc/sys/kernel/pid_max
  10000
- A virtual file system differs from the disk file system in that files and directories are created and destroyed on the fly. For example, a directory */proc/<PID>* only exists in the time frame the process <PID> exists
- *$ cat /proc/<PID>/status*
- */proc/<PID>/fd* contains one symbolic link for each file the process has open. For example */proc/1243/1* links to the stdout of process 1243

- */proc/<PID>/task/<TID>* contains information on thread <TID> of the process

stdio buffering

the glibc library I/O functions (e.g., fprintf, fscanf, fputs, fputc, fgetc, ..) perform data buffering in user-space to reduce read and write system calls.
int **fflush**(FILE **stream*) forces the data in a stdio output stream to be written. If *stream* is NULL, all stdio buffers get flushed.

Networking

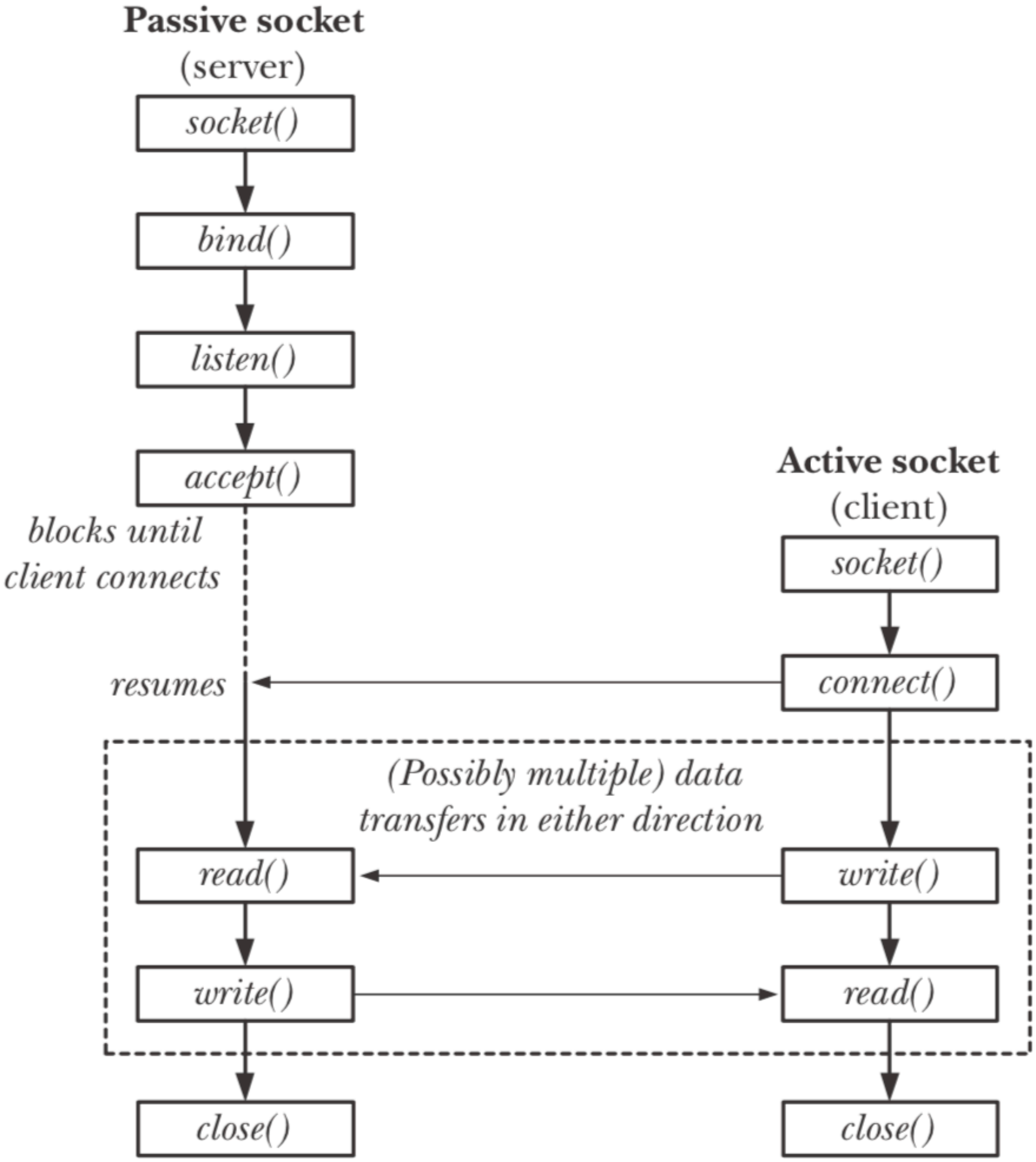


MTU (Maximum Transfer Unit): the upper limit of a Data-Link-Layer frame
An IPv4 datagram can be 65k bytes long. The MTU is much smaller. Then the Network Layer features fragmentation of the original datagram into smaller IP datagram by adding an
To see the network media interfaces and their MTU
$ netstat -i
IPv6 uses 128-bit address, where IPv3 uses 32-bit address
128.0.0.1 is the Loopback address

- *UNIX domain sockets* (AF_UNIX): allow communication between applications on the same host, in the kerne space
- *Internet domain sockets*: allow applications on different hosts to communicate via a TCP/IP network. There are two variants, IPv4 (AF_INET) and IPv6 (AF_INET6)
- For each domain we can have 3 types of sockets
  1. *Stream sockets* (SOCK_STREAM) or *TCP socket*: reliable, bidirectional full-dupex,

byte-stream. Connection-oriented: can only work in pairs, the local and the peer get connected

2. *Datagram sockets* (SOCK_DGAM) or *UDP socket*: not reliable. Connectionless
3. *Raw sockets* (SOCK_RAW)



- System calls:
  - *socket()* to create a new socket
    bind() to binda a socket to an address (including the port)
  - *listen()* to accept incoming connections from other stream sockets (typ. used by server)
  - *accept()* to accept a connection from a peer
  - *connect()* to establishes a connection with another socket (typ used by clinets.)
  - *close()* to close the connection

int **socket**(int *domain*, int *type*, int *protocol*);
Example, fd = **socket**(AF_INET, SOCK_STREAM, 0);
int **bind**(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int **listen**(int *sockfd*, int *backlog*);
int **accept**(int *sockfd*, struct sockaddr **addr*, socklen_t **addrlen*);

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```