
Topic 3: Error Handling, Low-level I/O, Signals*

K24: Systems Programming

Instructor: Mema Roussopoulou

Error Handling

◆ Murphy's Law

- *Anything that can go wrong, will go wrong.*
- Potential errors/mistakes have to be anticipated and corresponding corrective action (if possible) should be adopted.

◆ Instead of using an `fprintf()`, the call `perror()` could be used:

- `void perror(char *estring)`
 - Prints out the string pointed to by `estring` denoting a specific kind of a mistake (choice of the programmer), plus a system-generated error string
- ◆ If we include the header file `#include <errno.h>`, the variable `errno` will have as its value an integer corresponding to the latest error that occurred.

```
if (amt=read(fd, buf, numbyte)) == -1)
then fprintf(stderr, "read failed; errno= %d\n", errno);
```

C Program with Error Handling (errors_demo.c)

```
# include < stdio .h > /* for fopen, printf */
# include < stdlib .h > /* for malloc */
# include < errno .h > /* for errno */
int main () {
    FILE *fp = NULL; char *p = NULL; int stat =0;
    fp = fopen ( "a_non_existent_file", "r" );
    if ( fp == NULL ) { /* check for error */
        printf ("errno = %d \n ", errno);
        perror ("fopen");
    }

    p =( char *) malloc (2147483647) ;
    if ( p == NULL ) { /* check for error */
        printf ( "errno = %d \n" , errno);
        perror ( " malloc " ) ;
    } else {
        printf ( "Carry on \n");
    }
    stat = unlink ( "/etc/motd" );
    if ( stat == -1) { /* check for error */
        printf ( "errno = %d\n" , errno) ;
        perror ( " unlink " ) ;
    }
    return (1) ;
}
```

C Program with Error Handling (errors_demo.c)

```
mema@browser> gcc errors_demo.c  
mema@browser> ./a.out  
errno = 2  
fopen : No such file or directory  
errno=12  
malloc: Cannot allocate memory  
errno = 13  
unlink : Permission denied  
mema@browser>
```

Low-level Input/Output

- ◆ The stdio library enables the average user to carry out I/O without worrying about buffering and/or data conversion.
- ◆ The stdio is a user-friendly set of system calls.
- ◆ Low-level I/O functionality is required
 1. when the amenities of stdio are not desirable (for whatever reason) in accessing files/devices, or
 2. when interprocess communication occurs with the help of pipes/sockets.

Το χαμηλότερο επίπεδο αλληλεπίδρασης με το OS UNIX είναι οι κλήσεις συστήματος. Είναι τα «σημεία εισόδου» στον πυρήνα.

Low-level I/O (cont'd)

- ◆ In low-level I/O, file descriptors that identify files, pipes, sockets and devices are small integers
 - The above is in contrast to what happens in the *stdio* library where respective identifiers are pointers.
- ◆ Designated (fixed) file descriptors:
 - 0 : standard input
 - 1 : standard output
 - 2 : standard error (for error diagnostics).
- ◆ The above file descriptors 0, 1, and 2 correspond to pointers to the *stdin*, *stdout*, and *stderr* files of the *stdio* library.
- ◆ The file descriptors are “inherited” by any child process that the parent in question creates.

Low-level I/O (cont'd)

- ◆ A FILE pointer points to a data structure that includes, among other things, the file descriptor.
- ◆ When the shell starts up and runs a program, the program inherits three open files with file descriptors 0,1,2.
- ◆ If the program opens any additional files, they will have file descriptors 3,4, etc.

The *open()* system call

- ◆ `int open(char *pathname, int flags
 [, mode_t mode])`
- ◆ The call opens or creates a file with absolute or relative *pathname* for reading/writing.
- ◆ *flags* designates the way (a number) with which the file can be accessed; values for flags may be constructed by a bitwise-inclusive OR of flags from the following set:
 - `O_RDONLY`: open for reading only.
 - `O_WRONLY`: open for writing only.
 - `O_RDWR`: open for both reading and writing.
 - `O_APPEND`: write at the end of the file.
 - `O_CREAT`: create a file if it does not already exist.
 - `O_TRUNC`: the size of the file will be truncated to 0 if the file exists.
- ◆ Returns a file descriptor on success or -1 on failure.

The *open()* system call

- ◆ required: `#include <fcntl.h>` defines all these (and more) flags.
- ◆ The non-mandatory *mode* parameter is an integer that designates the desired access permissions during the creation of a file (access rights not allowed from the *umask* are not allowed).
- ◆ *open* returns an integer that designates the file created and, in case of no success, it returns -1.

Παραδείγματα χρήσης open

```
int myfd;  
myfd = open("/home/mema/my.dat", O_RDONLY);
```

```
mode_t fdmode = (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);  
  
if ((fd = open("info.dat", O_RDWR | O_CREAT, fdmode)) == -1)  
    perror("Failed to open info.dat");
```

S_IRUSR	Ανάγνωση ιδιοκτήτη
S_IWUSR	Γραφή ιδιοκτήτη
S_IXUSR	Εκτέλεση ιδιοκτήτη
S_IRWXU	Ανάγνωση, γραφή, εκτέλεση ιδιοκτήτη
S_IRGRP	Ανάγνωση ομάδα
S_IWGRP	Γραφή ομάδα
S_IXGRP	Εκτέλεση ομάδα
S_IRWXG	Ανάγνωση, γραφή, εκτέλεση ομάδα
S_IROTH	Ανάγνωση λοιποί
S_IWOTH	Γραφή λοιποί
S_IXOTH	Εκτέλεση λοιποί
S_IRWXO	Ανάγνωση, γραφή, εκτέλεση λοιποί

```
# include < stdio .h > // to have access to printf ()
# include < stdlib .h > // to enable exit calls
# include < fcntl .h > // to have access to flags def

# define PERMS 0644 // set access p e r m i s s i o n s

char *workfile ="mytest" ;

main () {
int filedes;

if (( filedes = open (workfile, O_CREAT | O_RDWR,
                    PERMS ) ) == -1) {
    perror(" creating ");
    exit(1); }
else {
    printf (" Managed to get to the file successfully
\n" ); }

exit (0) ;
}
```

```
mema@browser> gcc createfile .c
mema@browser> ./a.out
Managed to get to the file successfully
mema@browser> ls -l
total 20
-rwxr-xr-x 1 mema mema 8442 2010 -04 -06 21:50 a.out
-rw-r--r-- 1 mema mema 375 2010 -04 -06 21:49 createfile.c
-rw-r--r-- 1 mema mema 0 2010 -04 -06 21:50 mytest
mema@browser> cat > mytest
This is a test
mema@browser> ./a.out
Managed to get to the file successfully
mema@browser> ls
a.out  createfile.c  mytest
mema@browser> more mytest
This is a test
mema@browser>
```

The *creat*() call

- ◆ The *creat* call is an alternative way to create a file (instead of using *open*()).
- ◆ `int creat(char *pathname, mode_t mode);`
- ◆ *pathname* is any Unix pathname giving the target location in which the file is to be created.
- ◆ *mode* helps set up the access rights.
- ◆ *creat* will always truncate an existing file before returning its file descriptor.

```
filedes = creat ("/tmp/mr.txt", 0644) ;
```

is equivalent to:

```
filedes = open ("/tmp/mr.txt", O_WRONLY |  
O_CREAT | O_TRUNC , 0644) ;
```

The read() call

- ◆ `int read(int fd char *buf, int count)`
- ◆ Reads at most *count* bytes from a file, device, end-point of a pipe, or socket that is designated by *filedes* and places the bytes in *buf*.
- ◆ The call returns the number of bytes *successfully read*, 0 if we are past the last byte-already read, and -1 if a problem occurs.
- ◆ When do we read less bytes?
 - If file has less characters left to be read.
 - If the `read()` is “interrupted” by a signal.
 - If reading on pipe/socket, and a character becomes available (in which case a while-loop is needed to read all characters).

Παράδειγμα συνάρτησης read

```
char buf[100];
```

```
ssize_t bytesread;
```

```
bytesread = read(STDIN_FILENO, buf, 100);
```

```
char *buf;
```

```
ssize_t bytesread;
```

```
bytesread = read(STDIN_FILENO, buf, 100);
```

↑
Τι θα συμβεί εδώ?

Στα επόμενα παραδείγματα,

STDIN_FILENO=0,

STDOUT_FILE=1,

STDERR_FILE=2

```
# include <stdio .h>
# include <stdlib .h>
# include <fcntl .h>
# include <unistd .h>
# define BUFSIZE 27
main (){
char buffer [BUFSIZE];
int filedes;  ssize_t nread;  long total =0;

if ((filedes = open("anotherfile", O_RDONLY))== -1){
    printf (" error in opening anotherfile \n");
    exit (1) ;
}

while ((nread = read (filedes, buffer , BUFSIZE)) > 0 )
    total += nread ;
    printf ("Total char in anotherfile %ld \n",total )
    exit (0) ;
}
```

```
mema@browser> ./a.out
Total char in anotherfile 936
mema@browser>
```

*What happens if char *buffer=NULL; is used instead of char buffer[BUFSIZE];*

The write() and close() calls

◆ `int write(int fd, char *buf, int count)`

- Writes at most *count* bytes of content from the buffer to the file that is described by *fd*
- `write()` returns the number of bytes successfully written out to the file, or -1 in case of failure.
- use the write call with:
`#include <unistd.h>`

◆ `int close(int fd)`

- releases the file descriptor *fd*; returns 0 in case of successful release and -1 otherwise.
- use the close call with:
`#include <unistd.h>`

Παράδειγμα συνάρτησης write

Ίσως διαβάσει
< 1024 bytes

```
#define BLKSIZE 1024  
char buf[BLKSIZE];  
ssize_t bytesread;
```

```
bytesread = read(STDIN_FILENO, buf, BLKSIZE);  
if (bytesread > 0)  
    write(STDOUT_FILENO, buf, bytesread);
```

Καμιά εγγύηση ότι θα
γράψει bytesread bytes



Παράδειγμα χρήσης read, write, close

```
#include <stdio.h>
```

```
#include <stdlib.h> /* for exit */
```

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
#include <sys/stat.h>
```

```
int main(){
```

```
    int fd, bytes, bytes1, bytes2;
```

```
    char buf[50];
```

```
    mode_t fdmode = S_IRUSR|S_IWUSR;
```

```
    if ( ( fd=open("t", O_WRONLY |  
                O_CREAT, fdmode ) ) == -1 ){
```

```
        perror("open");
```

```
        exit(1);
```

```
    }
```

```
    bytes1 = write(fd, "First write. ", 13);
```

```
    printf("%d bytes were written. \n", bytes1);
```

```
    close(fd);
```

```
    if ( (fd=open("t", O_WRONLY | O_APPEND)) == -1 ){
```

```
        perror("open");
```

```
        exit(1);
```

```
    }
```

```
    bytes2 = write(fd, "Second Write. \n", 14);
```

```
    printf("%d bytes were written. \n", bytes2);
```

```
    close(fd);
```

Παράδειγμα χρήσης read, write, close

```
if ( (fd=open("t", O_RDONLY)) == -1 ){
    perror("open");
    exit(1);
}
bytes=read(fd, buf, bytes1+bytes2);
printf("%d bytes were read \n",bytes);
close(fd);

buf[bytes]='\0';
printf("%s\n",buf);
return 1;
}
```

Παράδειγμα χρήσης read, write, close

```
mema@browser> ./read-write-close
13 bytes were written.
14 bytes were written.
27 bytes were read
First write. Second Write.
mema@browser> ls -lt
total 12
-rwxr-xr-x 1 mema mema 5777 Dec 29 07:30 read-write-close*
-rw-r--r-- 1 mema mema 842 Dec 29 07:19 read-write-close.c
-rw----- 1 mema mema 41 Dec 29 07:35 t
mema@browser>
```

Παράδειγμα χρήσης read, write, close

```
# include <stdio .h>
# include <string .h>
# include <stdlib .h>
# include <fcntl .h>
# include <unistd .h>
# include <sys / stat .h>
# define BUFFSIZE 1024
int main( int argc , char * argv []) {
    int n, from , to; char buf[ BUFFSIZE ];
    mode_t fdmode = S_IRUSR | S_IWUSR |
                    S_IRGRP | S_IROTH ;
    if (argc !=3) {
        write(2," Usage : ", 7);
        write(2, argv[0] , strlen (argv[0]));
        write(2," from-file to-file \n", 19) ; exit (1) ; }

    if ( (from = open(argv [1], O_RDONLY)) < 0 ){
        perror (" open "); exit (1); }

    if ( (to= open(argv[2], O_WRONLY | O_CREAT |
                    O_APPEND , fdmode)) < 0 ){
        perror (" open "); exit (1); }

    while ( (n= read(from, buf , sizeof( buf ))) > 0 )
        write(to ,buf ,n);
    close(from); close(to); return (1);
```

Παράδειγμα χρήσης read, write, close

```
mema@browser> ls
file1 file2
mema@browser> more file1
This is file1.
Will append it to file2.
mema@browser> more file2
This is file2.
mema@browser> ./a.out
Usage : ./a.out from-file to-file
mema@browser> ./a.out file1 file2
This is file2.
This is file1.
Will append it to file2.
mema@browser>
```

Calls: *dup*, *dup2*

(Duplicate file descriptors)

- ◆ *int dup(int oldfd);*
finds the lowest numbered unused file descriptor and makes it refer to the same entity to which *oldfd* refers.
- ◆ *int dup2(int oldfd, int newfd);* frees *newfd* (if in use) and makes *newfd* refer to the same entity to which *oldfd* refers - note:
 1. If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
 2. If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then *dup2()* does nothing, and returns *newfd*.
- ◆ After a successful return from one of these system calls, the old and new file descriptors may be used *interchangeably*.

Παράδειγμα dup, dup2

```
#include <stdio.h>
#include <stdlib.h> /* for exit */
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(){
    int fd1, fd2, fd3;
    mode_t fdmode = S_IRUSR|S_IWUSR|
                    S_IRGRP| S_IROTH;

    if ( ( fd1=open("r", O_WRONLY | O_CREAT |
                    O_TRUNC, fdmode ) ) == -1 ){
        perror("open");
        exit(1);
    }
    printf("fd1 = %d\n", fd1);
    write(fd1, "What ", 5);
    fd2=dup(fd1);
    printf("fd2 = %d\n", fd2);
    write(fd2, "time", 4);
    close(0);
}
```

Παράδειγμα dup, dup2

```
fd3=dup(fd1);  
printf("fd3 = %d\n", fd3);  
write(fd3, " is it", 6);  
dup2(fd2, 2);  
write(2,"?\n",2);  
close(fd1);  
close(fd2);  
close(fd3);  
return 1;  
}
```

Παράδειγμα dup, dup2

```
mema@browser> ./dup
fd1 = 3
fd2 = 4
fd3 = 0
mema@browser> ls -l r
-rw-r--r-- 1 mema mema 17 Dec 29 07:58 r
mema@browser> more r
What time is it?
mema@browser>
```

Γράψτε ένα πρόγραμμα που αντιγράφει ένα αρχείο στο τέλος ενός άλλου

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> /* for exit */
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char *argv[]){
    int n, from, to;
    char buf[1024];

    mode_t fdmode = S_IRUSR|S_IWUSR|
                    S_IRGRP|S_IROTH;

    if (argc!=3) {
        write(2,"Usage: ", 7);
        write(2, argv[0], strlen(argv[0]));
        write(2," from-file to-file\n", 19);
        exit(1);
    }

    if ( ( from=open(argv[1], O_RDONLY)) < 0 ){
        perror("open");
        exit(1);
    }
```

Γράψτε ένα πρόγραμμα που αντιγράφει ένα αρχείο στο τέλος ενός άλλου

```
if ( (to=open(argv[2], O_WRONLY|
    O_CREAT|O_APPEND, fdmode)) < 0 ){
    perror("open");
    exit(1);
}

while ( (n=read(from, buf, sizeof(buf))) > 0 )
    write(to,buf,n);
close(from);
close(to);
return 1;
}
```

Παράδειγμα from-to

```
mema@browser> more f1
```

```
This is a test.
```

```
mema@browser> ./from-to f1 f2
```

```
mema@browser> more f2
```

```
This is a test.
```

Accessing inode information with stat()

- ◆ `int stat(char *path, struct stat *buf);`
- ◆ `int fstat(int fd, struct stat *buf);`
 - returns information about a file; path points to the file (or fd) and the buf structure is filled with information.
- ◆ **such information includes:**
 - `buf->st_dev`: ID of device containing file
 - `buf->st_ino`: inode number
 - `buf->st_mode`: the last 9 bits represent the access rights of owner, group, and others. The first 4 bits indicate the type of the node (after a bitwise-AND with the constant `S_IFMT`, if the outcome is `S_IFDIR`, the node is a directory, if outcome is `S_IFREG`, the mode is a regular file, etc.)
 - `buf->st_nlink`: number of hard links

Accessing inode information with stat()

- buf->st_uid: user-ID of owner
 - buf->st_gid: group ID of owner
 - buf->st_size: total size, in bytes
 - buf->st_atime: time of last access
 - buf->st_mtime: time of last modification of content
 - buf->st_ctime: time of last status change
- ◆ Header files needed: `<sys/stat.h>` and `<sys/types.h>`
 - ◆ `int fstat(int fd, struct stat *buf);` is identical to `stat` but it works with file descriptors.
 - ◆ `int lstat(char *path, struct stat *buf);` is identical to `stat`, except that if `path` is a symbolic link, then the link itself is `stat`-ed, not the file that it refers to.

st_mode is a 16-bit quantity

- ◆ 4 first bits indicate the type of the file (16 possible values -less than 10 file types are in use now: regular file, dir, block-special, char-special, fifo, symbolic link, socket).
- ◆ the next 3 bits set the flags: set-user-ID, set-group-ID and the sticky bits respectively.
- ◆ next 9 bits (three groups of 3 bits per group) indicate the read/write/execute access rights for the groups: owner, group and others.
- ◆ masking can be used to decipher the permissions each file system entity is given.

stat() (cont'd)

- ◆ The fields `st_atime`, `st_mtime` and `st_ctime` designate time as number of seconds past since 1/1/1970 of the Coordinated Universal Time (UTC).
- ◆ The function `ctime` helps bring the content of the fields `st_atime`, `st_mtime` and `st_ctime` in a more readable format (that of the date). The call is:
 - `char *ctime(time t *timep);`
- ◆ `stat` returns 0 if successful; otherwise, -1

Definitions in <sys/stat.h>

```
#define S_IFMT 0170000      /* type of file */
#define S_IFREG 0100000    /* regular */
#define S_IFDIR 0040000    /* directory */
#define S_IFBLK 0060000    /* block special */
#define S_IFCHR 0020000    /* character special */
#define S_IFIFO 0010000    /* fifo */
#define S_IFLNK 0120000    /* symbolic link */
#define S_IFSOCK 0140000   /* socket */
```

Testing for a specific type of a file is easy using code fragments of the following style:

```
if ( ( info.st_mode & S_IFMT ) == S_IFIFO )
    printf ( " this is a fifo queue .\n" );
```

Παράδειγμα stat

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>
```

```
int main(int argc, char *argv[]){
    struct stat statbuf;
```

```
    if (stat(argv[1], &statbuf) == -1)
        perror("Failed to get file status");
    else
        printf("%s accessed: %s modified: %s",
               argv[1], ctime(&statbuf.st_atime),
               ctime(&statbuf.st_mtime));
```

```
    return 1;
}
```

Παράδειγμα stat

```
mema@browser> ./stat
Failed to get file status: Bad address
mema@browser> ./stat f1
f1 accessed: Tue Dec 29 08:47:35 2013
  modified: Tue Dec 29 08:47:35 2013
mema@browser>
```

Accessing Directory Content

- ◆ A directory's content (ie, pairs of inodes and node names) can be accessed with the help of the calls: `opendir`, `readdir` and `closedir`.
- ◆ Access to a directory happens via a pointer `DIR *` (similar to the `FILE *` pointer that is used by the `stdio`).
- ◆ Every item in the directory is described by a structure called `struct dirent` that includes the following two elements:
 - `d_no`: inode number;
 - `d_name[]`: a character string giving the filename (null terminated)
- ◆ Note: it is not feasible to change the content of the directory or its structure, using these calls.
- ◆ Required header files: `<sys/types.h>` and `<dirent.h>`

opendir, readdir, closedir

- ◆ **DIR *opendir(char *name);**
 - opens up the directory *name* and returns a pointer type DIR for accessing the directory
 - if there is a mistake, the call returns NULL
- ◆ **struct dirent *readdir(DIR *dirp);**
 - returns a pointer to a dirent structure representing the next directory entry in the directory pointed to by *dirp*
 - if for the current entry, the field *d_ino* is 0, the respective entry has been deleted.
 - returns NULL if there are no more entries to be read.
- ◆ **int closedir(DIR *dirp);**
 - closes the directory associated with *dirp*
 - function returns 0 on success. On error, -1 is returned, and *errno* is set appropriately.

Παράδειγμα (dir)

opendir, readdir, closedir

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
```

```
void do_ls(char dirname[]){
    DIR          *dir_ptr;
    struct dirent *direntp;
```

```
    if ( ( dir_ptr = opendir( dirname ) ) == NULL )
        fprintf(stderr, "cannot open %s \n",dirname);
    else {
        while { ( ( direntp=readdir(dir_ptr) ) != NULL )
            printf("%s\n", direntp->d_name) ;
        }
        closedir(dir_ptr);
    }
}
```


Παράδειγμα (dir)

opendir, readdir, closedir

```
int main(int argc, char *argv[]) {  
  
    if (argc == 1 )  
        do_ls(".");  
    else  
        while ( --argc ){  
            printf("%s: \n", *++argv ) ;  
            do_ls(*argv);  
        }  
}
```

Παράδειγμα (dir)

opendir, readdir, closedir

```
mema@browser> ./dir . mydir
```

```
.:  
.  
..  
read-write-close.c  
dup.c  
read-write-close  
t  
dup  
r  
from-to.c  
from-to  
f1  
f2  
stat.c  
stat  
dir.c  
mydir  
dir
```

Παράδειγμα (dir)

opendir, readdir, closedir

```
mydir:
```

```
.
```

```
..
```

```
a
```

```
b
```

```
c
```

```
d
```

```
e
```

```
f
```

```
g
```

```
mema@browser>
```

Παράδειγμα- Πρόγραμμα (ls-l) με συμπεριφορά ls -al

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <dirent.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
char *modes[]={ "---", "--x", "-w-", "-wx", "r--", "r-x",  
                "rw-", "rwx"}; // 8 distinct modes
```

```
void list(char *);
```

```
void printout(char *);
```

```
main(int argc, char *argv[]){  
    struct stat mybuf;
```

```
    if (argc<2) { list("."); exit(0);}  
}
```

Παράδειγμα- Πρόγραμμα (ls-1) με συμπεριφορά ls -al

```
while(--argc){  
    if (stat(*++argv, &mybuf) < 0)  
        { perror(*argv); continue;}  
  
    if ((mybuf.st_mode & S_IFMT) == S_IFDIR )  
        list(*argv);    // directory encountered  
    else  
        printout(*argv); // file encountered  
}  
}
```

Συνέχεια παραδείγματος (ls-1)

```
void list(char *name){
    DIR *dp;
    struct dirent *dir;
    char *newname;

    if ((dp=opendir(name))== NULL )
        {perror("opendir"); return;}
    while ((dir = readdir(dp)) != NULL ) {
        if (dir->d_ino == 0 ) continue;
        newname=(char *)malloc(strlen(name)+
                                strlen(dir->d_name)+2);
        strcpy(newname,name);
        strcat(newname,"/");
        strcat(newname,dir->d_name);
        printout(newname);
        free(newname); newname=NULL;
    }
    closedir(dp);
}
```

Συνέχεια παραδείγματος

```
void printout(char *name){
struct stat mybuf;
char type, perms[10];
int i,j;

stat(name, &mybuf);
switch (mybuf.st_mode & S_IFMT){
  case S_IFREG: type = '-'; break;
  case S_IFDIR: type = 'd'; break;
  default:     type = '?'; break;
}
*perms='\0';
for(i=2; i>=0; i--){
  j = (mybuf.st_mode >> (i*3)) & 07;
  strcat(perms,modes[j]); }

printf("%c%s%3d %5d/%-5d %7d %.12s %s \n",
  type,perms,mybuf.st_nlink,
  mybuf.st_uid, mybuf.st_gid, mybuf.st_size,
  ctime(&mybuf.st_mtime)+4, name);
}
```

Αποτέλεσμα

```
mema@browser> ./ls-l mydir
drwxr-xr-x 5 1000/1000 4096 Apr 26 19:48 mydir/.
drwxr-xr-x 3 1000/1000 4096 Apr 28 00:09 mydir/..
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/a
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/b
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/c
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/d
drwxr-xr-x 2 1000/1000 4096 Apr 27 22:20 mydir/e
drwxr-xr-x 2 1000/1000 4096 Apr 26 19:48 mydir/f
drwxr-xr-x 2 1000/1000 4096 Apr 26 19:48 mydir/g
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/i
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/j
-rw-r--r-- 1 1000/1000 0 Apr 26 19:48 mydir/k
mema@browser>
```


mkdir and rmdir

- ◆ *int mkdir(char *path, int mode)*
- ◆ Creates a new directory *path* with permissions *mode* (permissions not allowed by *umask* are not given)
- ◆ Analogous to *open* for files

- ◆ *int rmdir(char *path)*
- ◆ Removes the directory *path*, if it is empty
- ◆ Analogous to *unlink* for files

chmod, rename calls

- ◆ *int chmod(char *path, mode_t mode)*
int fchmod(int fd, mode_t mode)
- ◆ Change the permissions (on files with name *path* or having an *fd* descriptor) according to what *mode* designates.
- ◆ On success, 0 is returned; otherwise -1

- ◆ *int rename(const char *oldpath,*
*const char *newpath)*
- ◆ Renames a file, moving it between directories (indicated with the help of *oldpath* and *newpath*) if required.
- ◆ On success, 0 is returned; otherwise -1

lseek call

- ◆ *off_t lseek(int filedes, off_t offset, int startflag);*
- ◆ *lseek* repositions the offset of the open file associated with *filedes* to the argument *offset* according to the directive *startflag* as follows:
 - 1. SEEK_SET: The offset is set to offset bytes; usual actual integer value = 0
 - 2. SEEK_CUR: The offset is set to its current location plus offset bytes; usual actual integer value = 1
 - 3. SEEK_END: The offset is set to the size of the file plus offset bytes. usual actual integer value = 2
 -

lseek call

```
off_t newposition;  
...  
newposition = lseek(fd, (off_t) -32, SEEK_END);
```

- Positions the read/write pointer 32 bytes BEFORE the end of the file.
- When used with *read()* and *write()* system calls, provides all tools necessary to do *input and output randomly*.

link and *unlink*

- ◆ *int unlink(char *pathname)*
- ◆ Deletes a name from the file system; if that name is the last link to a file and no other process has the file open, the file is deleted and its space is made available.

- ◆ *int link(char *oldpath, char *newpath)*
- ◆ It creates a new hard link to an existing file. If *newpath* exists, it will not be overwritten.
- ◆ The created link essentially connects the inode of the *oldpath* with the name of the *newpath*.

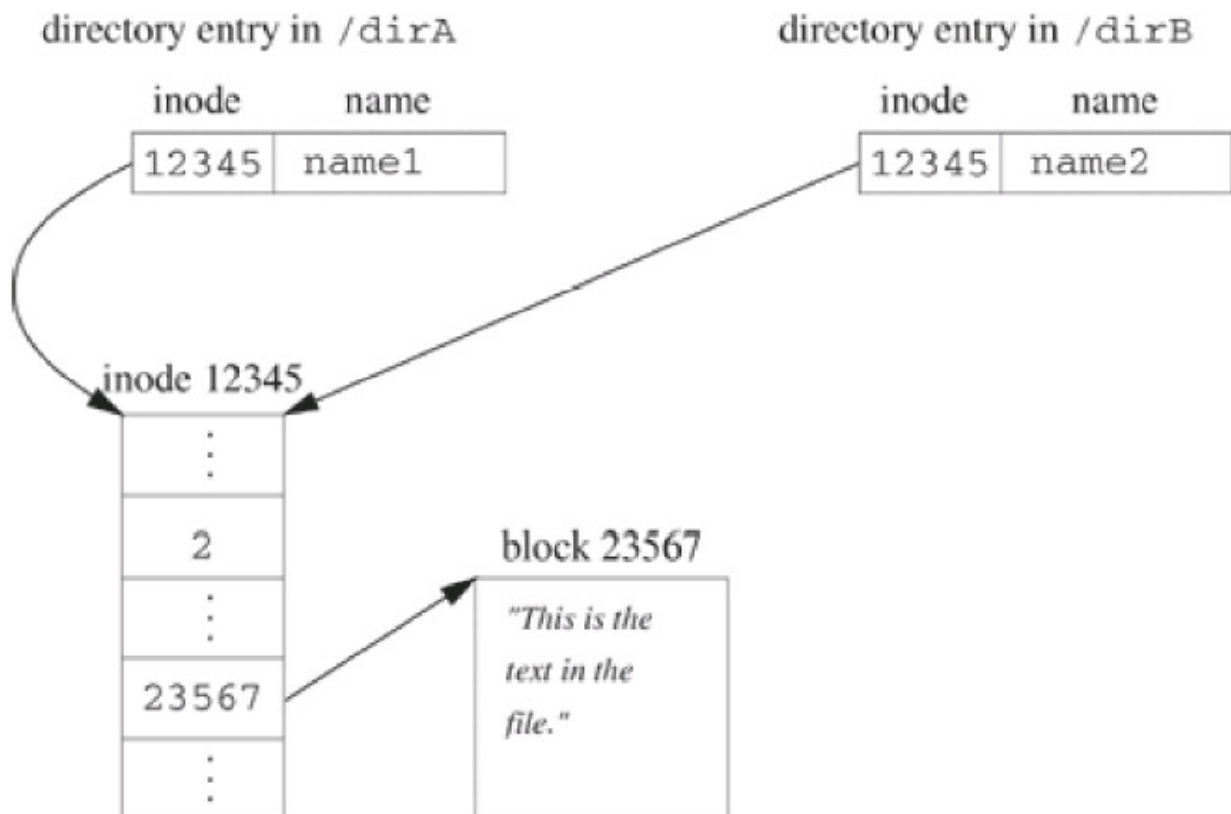
symlink and *readlink* calls

- ◆ *int symlink(const char *oldpath, const char *newpath)*
 - ◆ Creates a symbolic link named *newpath* that contains the string *oldpath*.
 - ◆ A symbolic link (or soft link) may point to an existing file or to a nonexistent one; the latter is known as a **dangling link**.
 - ◆ On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.
-
- ◆ *ssize_t readlink(char *path, char *buf, size_t bufsiz)*
 - ◆ Places the contents of the symbolic link *path* (i.e., the string for the path that the link points to) in the buffer *buf* that has size *bufsiz*.
 - ◆ On success, *readlink* returns the number of bytes placed in *buf*; otherwise, -1

Παράδειγμα link

```
#include <stdio.h>
#include <unistd.h>
```

```
if (link("/dirA/name1", "/dirB/name2") == -1)
    perror("Failed to make a new link in /dirB");
```



Σήματα

- ◆ Ειδοποίηση μέσω λογισμικού σε διεργασία ότι κάτι συνέβη
 - Σφάλμα εκτέλεσης, core
 - Από πληκτρολόγιο (Ctrl-C, Ctrl-Z)
 - Μια διεργασία σε μία άλλη
 - Εντολή kill κτλ...

- ◆ 4 Επιλογές για κάθε σήμα:
 - Αγνόησέ το
 - Προκαθορισμένη δράση
 - Ορισμός συνάρτησης (διαχειριστής σήματος – signal handler) για το πώς να αντιμετωπιστεί
 - Μπλοκάρисέ το (possible with POSIX Reliable Signals)

SIGNAL NAME	#	DESCRIPTION; DEFAULT ACTION
SIGABRT	6	Process abort; Implementation dependent
SIGALRM	14	Alarm clock; (Μετρητής) Abnormal termination
SIGBUS	7	Access undefined part of memory object; Implementation dependent
SIGCHLD	17	Child terminated, stopped or continued; Ignore
SIGCONT	18	Execution continued if stopped; Continue
SIGFPE	8	Error in arithmetic operation as in division by zero; Implementation dependent
SIGHUP	1	Hang-up (death) on controlling terminal (process); Abnormal termination
SIGILL	4	Invalid hardware instruction; Implementation dependent

SIGNAL NAME	#	DESCRIPTION; DEFAULT ACTION
SIGINT	2	Interactive attention signal (usually Ctrl-C); Abnormal termination
SIGKILL	9	Terminated (cannot be caught or ignored); Abnormal termination
SIGPIPE	13	Write on a pipe with no readers; Abnormal termination
SIGQUIT	3	Interactive termination: core dump (usually Ctrl-\); Implementation dependent
SIGSEGV	11	Invalid memory reference; Implementation dependent
SIGSTOP	19	Execution stopped (cannot be caught or ignored); Stop
SIGTERM	15	Termination; Abnormal termination
SIGTSTP	20	Terminal stop (usually Ctrl-Z); Stop

SIGNAL NAME	#	DESCRIPTION; DEFAULT ACTION
SIGTTIN	21	Background process attempting to read; Stop
SIGTTOU	22	Background process attempting to write; Stop
SIGUSR1	10	User-defined signal 1; Abnormal termination
SIGUSR2	12	User-defined signal 2; Abnormal termination

- If any signal is used, the header file `<signal.h>` must be included.

Εντολή kill

Εντολή κελύφους:

kill [-signalName] processId

kill -s signalName processId

send a specific signal to process(es)

kill -l (λίστα συμβολικών ονομάτων για
σήματα -το γράμμα l εδώ, όχι το νούμερο)

Παραδείγματα:

kill -USR1 3424

kill -s USR1 3424

kill -9 3424

kill call

- ◆ `#include <sys/types.h>`
- ◆ `#include <signal.h>`
- ◆ `int kill(pid_t pid, int sig)`
- ◆ Signal *sig* is sent to process with *pid*
- ◆ Should the receiving and dispatching processes belong to the same user or the dispatching process is the superuser, the signal can be successfully sent.
- ◆ If *sig* is 0, then no signal is dispatched.
- ◆ On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set appropriately.

Παράδειγμα με signal - handle signal

```
#include <stdio.h>  
#include <signal.h>
```

```
void f(int);
```

```
int main(){  
  int i;  
  
  signal(SIGINT, f);  
  for(i=0;i<5;i++){  
    printf("hello\n");  
    sleep(1);  
  }  
}
```

```
void f(int signum){  /* no explicit call to  
                    function f */  
  printf("OUCH!\n");  
}
```

Παράδειγμα με signal - handle signal

```
mema@browser> ./handleSignal
hello
hello
hello
hello
^COUCH!
hello
mema@browser>
```


Παράδειγμα με signal – ignore signal

```
#include <stdio.h>  
#include <signal.h>
```

```
int main(){  
  int i;
```

```
  signal(SIGINT, SIG_IGN);
```

```
  printf("you can't stop me here! \n");
```

```
  while(1){  
    sleep(1);  
    printf("haha \n");
```

```
  }  
}
```

```
/* use cntrl-\ to get rid of this process */
```

If you change SIG_IGN to SIG_DFL, then default action is taken.



Παράδειγμα με signal – ignore signal

```
mema@browser> ./ignoreSignal  
you can't stop me here!  
haha  
haha  
haha  
haha  
haha  
haha  
haha  
haha  
haha  
haha  
^-\Quit  
mema@browser>
```

Συναρτήσεις raise, pause

◆ Εντολή `int pause()`

- Αναμονή διεργασίας μέχρι τη λήψη σήματος που η δράση της είτε εκτελεί συνάρτηση χρήστη είτε τερματίζει την διεργασία
- Πάντα επιστρέφει -1
- `#include <unistd.h>`

◆ Εντολή `int raise(int signalId)`

- Στέλνει σήμα στην ίδια διεργασία!!!
- Σε επιτυχία 0. Αλλιώς $\neq 0$ και θέτει το `errno`

Πχ:

```
if (raise(SIGUSR1) != 0)
    perror("Failed to raise SIGUSR1");
```

The *alarm* call

- ◆ *#include <unistd.h>*
unsigned int alarm(unsigned int count);
- ◆ *alarm()* delivers a **SIGALRM** to the invoking process in *count* seconds.
- ◆ The default signal handler prints a message and terminates the process.
- ◆ If *count=0*, previous *alarm()* call is canceled.
- ◆ Returns the number of seconds remaining until any previously scheduled alarm was due to be delivered; otherwise, 0.

The *alarm* call

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

    alarm(3); /* schedule an alarm signal in 3sec */
    printf("Looping forever...\n");
    while (1);
    printf("This line should never be executed.\n");
}
```

```
mema@browser> date; ./alarm_demo; date
Mon Apr 12 21:20:41 EEST 2011
Looping forever...
Alarm clock
Mon Apr 12 21:20:44 EEST 2011
mema@browser>
```

Παράδειγμα: signal, alarm, pause

```
#include <stdio.h>  
#include <signal.h>
```

```
main(){  
    void wakeup(int);  
  
    printf("about to sleep for 4 seconds \n");  
    signal(SIGALRM, wakeup);  
  
    alarm(4);  
    pause();  
    printf("Hola Amigo\n");  
}  
  
void wakeup(int signum){  
    printf("Alarm received from kernel\n");  
}
```

Παράδειγμα: signal, alarm, pause

```
mema@browser> ./signal-alarm-pause  
about to sleep for 4 seconds  
Alarm received from kernel  
Hola Amigo  
mema@browser>
```

Unreliable Signals – a headache in “older” Unix

```
int sig_int();
...
signal(SIGINT, sig_int);
...

sig_int() {
/* this is the point of possible problems */
signal(SIGINT, sig_int);
...
}
```

1. After a signal has occurred but before the first line in `sig_int` has completed executing, another signal occurs.
2. The second signal would cause the default action which is to *terminate* the process.
3. Although the program may seem to work, we “lose” a signal in the process.

Unreliable Signals

- ◆ A process could NOT turn off (i.e., block) a signal to prevent the signal from interrupting it at a later time
- ◆ All process could do was to *ignore* the signal (with a trick)
- ◆ Below, intent is to have program wait until signal has occurred before continuing work

```
int flag = 0;
main() {
int handler();

...
signal(SIGINT, handler);

...
while(flag == 0) {
    /* potential problem here*/
    pause();
...
}

handler() {
signal(SIGINT, handler);
flag = 1;
}
```

Unreliable Signals

- ◆ Under “regular” circumstances the process would “pause” until it received a SIGINT and then continue on to other actions (after the while statement) as the predicate would disqualify.
- ◆ Problem: signal occurs **after** test (flag =0), but **before** call to *pause* → *process could sleep forever* (if no other signals are generated)
 - Signal is lost!
- ◆ Code NOT correct, yet works most of the time

Unreliable Signal-ing

- ◆ Running the program, we get into the pause without noticing first signal got into the handler:

```
mema@browser> ./signal-exampleA  
The process ID of this program is 3245
```

- ◆ The (first) signal seems to be “lost”...
- ◆ Even though handler is invoked on first signal and flag gets set on the first signal, program is unaware of this because its check occurred before first signal is sent

Forcing a second interrupt with control-C, we terminate the program:

```
mema@browser> ./signal-exampleA  
The process ID of this program is 3245  
^Cmema@browser>
```

- ◆ **Signal Sets provide a (POSIX) reliable way of dealing with signals**

POSIX Signals -- first, some definitions

- ◆ When signal occurs, we say signal is *generated*
- ◆ *Delivered* means signal has been handled (default action, ignore, user-defined handler)
- ◆ Between generation and delivery a signal is *pending*
 - If another signal of same type is generated, the delivery of the new signal is unspecified (implementation-specific)
- ◆ A thread can keep signals in *pending* state by **blocking** them
- ◆ Signal mask: the **set** of all blocked signals (for a particular thread)
- ◆ With POSIX (reliable) signals, when in a signal handler, the signal type is temporarily added to the mask so that another signal of same type doesn't get delivered.

POSIX Signal Sets

- ◆ Signal sets are defined using the type `sigset_t`
- ◆ Sets are large enough to hold a representation of *all* signals in the system
- ◆ We may indicate interest in specific signals by empty-ing a set and then adding signals or by using a full set and then by selectively deleting certain signals

POSIX Signal Sets

- ◆ Initialization of signals happens through:
 - `int sigemptyset(sigset_t *set);`
 - `int sigfillset(sigset_t *set);`
- ◆ Manipulation of signal sets happens via
 - `int sigaddset(sigset_t *set, int signo);`
 - `int sigdelset(sigset_t *set, int signo);`
- ◆ Membership in a signal set:
 - `int sigismember(sigset_t *set, int signo)`

Example: creating different signal sets

```
# include <signal .h>
sigset_t mask1 , mask2 ;
...
...
sigempty (&mask1 ); // create an empty mask
...
sigaddset (&mask1 , SIGINT ); // add signal SIGINT
sigaddset (&mask1 , SIGQUIT ); //add signal SIGQUIT
...
sigfillset (&mask2 ); // create a full mask
...
// remove signal SIGCHLD
sigdelset (&mask2 , SIGCHLD );
....
...
• mask1 is created entirely empty
• mask2 is created entirely full
```


sigaction() call

- ◆ Once a set has been defined, we can define a specific method to handle a signal using `sigaction()`.
- ◆ `int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);`
- ◆ When `oldact` is non-null, on return, points to old signal handling action (old `sigaction` structure)
- ◆ The `sigaction` structure is:

```
struct sigaction {  
    void (*sa_handler)(int); // action to be taken  
    sigset_t sa_mask; // additional signals to be  
                        // blocked during the handling  
                        // of the signal  
    int sa_flags; // flags controlling handler  
                 // invocation  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    // pointer to a signal handler for  
    // real-time signals  
};
```

The sigaction structure

- ◆ ***sa_handler* field:** identifies the action to be taken when the signal *signo* is received (previous slide)
 1. SIG_DFL: restores the system's default action
 2. SIG_IGN: ignores the signal
 3. The address of a function which takes an int as argument. The function will be executed when a signal of type *signo* is received and the value of *signo* is passed as parameter. Control is passed to function as soon as signal is received and when function returns, control is passed back to the point at which the process was interrupted.
- ◆ ***sa_mask* field:** the signals specified here will be blocked during the execution of the *sa_handler*.

The sigaction structure

- ◆ **sa_flags field:** used to modify the behavior of *signo* – the originally specified signal.
 1. A signal's action is reset to SIG_DFL on return from the handler by **sa_flags=SA_RESETHAND**
 2. Extra information will be passed to signal handler, if **sa_flags=SIG_INFO**. Here, *sa_handler* is redundant and the final field *sa_sigaction* is used.
 3. Do not interrupt ongoing syscall if **sa_flags=SA_RESTART**
- ◆ Good idea to either use *sa_handler* or *sa_sigaction*. NOT both!

Use of sigaction

```
# include < stdio .h >
# include < stdlib .h >
# include < signal .h >
void catchinterrupt (int signo ){
printf("\n Catching : signo =%d\n",signo );
printf(" Catching : returning \n");
}
main (){
static struct sigaction act ;
act.sa_handler = catchinterrupt ;
sigfillset(&(act.sa_mask));
sigaction(SIGINT , &act , NULL );
printf(" sleep call #1\n");
sleep (1) ;
printf(" sleep call #2\n");
sleep (1) ;
printf(" sleep call #3\n");
sleep (1) ;
printf(" sleep call #4\n");
sleep (1) ;
printf(" Exiting \n");
exit (0) ;
}
```

Regardless of where the program is interrupted,
it resumes execution and carries on

```
mema@browser> ./signal-exampleB
sleep call #1
sleep call #2
^C
Catching : signo =2
Catching : returning
sleep call #3
^C
Catching : signo =2
Catching : returning
sleep call #4
^C
Catching : signo =2
Catching : returning
Exiting
```

Another execution:

Regardless of where the program is interrupted,
it resumes execution and carries on

```
mema@browser> ./signal-exampleB
```

```
sleep call #1
```

```
sleep call #2
```

```
^C
```

```
Catching : signo =2
```

```
Catching : returning
```

```
sleep call #3
```

```
sleep call #4
```

```
Exiting
```

```
mema@browser>
```

Changing the Behavior of program in interrupt

```
# include < stdio .h >
# include < stdlib .h >
# include < signal .h >
main (){
static struct sigaction act ;

act.sa_handler = SIG_IGN ; // the handler is set to IGNORE
sigfillset (&( act.sa_mask ));

sigaction (SIGINT , &act , NULL ); // control -c
sigaction (SIGTSTP , &act , NULL ); // control -z

printf(" sleep call #1\ n"); sleep (1) ;
printf(" sleep call #2\ n"); sleep (1) ;
printf(" sleep call #3\ n"); sleep (1) ;

act.sa_handler = SIG_DFL ; // reestablish DEFAULT behavior
sigaction (SIGINT , &act , NULL ); // default for control -c

printf(" sleep call #4\ n"); sleep (1) ;
printf(" sleep call #5\ n"); sleep (1) ;
printf(" sleep call #6\ n"); sleep (1) ;

sigaction (SIGTSTP , &act , NULL ); // default for control -z
printf(" Exiting \n");
exit (0) ;
}
```

mema@browser> ./signal-exampleC

sleep call #1

^Csleep call #2

^Z^Csleep call #3

sleep call #4

sleep call #5

^Zsleep call #6

Exiting

mema@browser> ./signal-exampleC

sleep call #1

sleep call #2

sleep call #3

sleep call #4

sleep call #5

^C

mema@browser> ./signal-exampleC

sleep call #1

^Csleep call #2

^C^Z^Zsleep call #3

^Z^Zsleep call #4

^Z^Zsleep call #5

^Z^Zsleep call #6

^ZExiting

mema@browser>

Restoring a *previous* action

```
# include < stdio .h >
```

```
# include < stdlib .h >
```

```
# include < signal .h >
```

```
main () {
```

```
static struct sigaction act, oldact;
```

```
printf(" Saving the default way of handling the  
control =c\n");
```

```
sigaction (SIGINT, NULL, &oldact);
```

```
printf(" sleep call #1\n"); sleep (4) ;
```

```
printf("Changing (Ignoring) the way of handling \n");
```

```
act.sa_handler = SIG_IGN ; // the handler is set to IGNORE
```

```
sigfillset(&( act.sa_mask ));
```

```
sigaction(SIGINT , &act , NULL );
```

```
printf(" sleep call #2\n"); sleep (4) ;
```

```
printf("Reestablishing old way of handling control-c\n");
```

```
sigaction(SIGINT , &oldact , NULL );
```

```
printf(" sleep call #3\n"); sleep (4) ;
```

```
printf(" Exiting \n");
```

```
exit (0) ; }
```

```
mema@browser> ./signal-exampleD
```

```
Saving the default way of handling the control =c
```

```
sleep call #1
```

```
^C
```

```
mema@browser>
```

```
mema@browser>
```

```
mema@browser>
```

```
mema@browser> ./signal-exampleD
```

```
Saving the default way of handling the control =c
```

```
sleep call #1
```

```
Changing (Ignoring ) the way of handling
```

```
sleep call #2
```

```
^C^C^C^C^C Restablishing old way of handling control-c
```

```
sleep call #3
```

```
^C
```

```
mema@browser>
```

Blocking Signals

- ◆ Occasionally, a program wants to **block all together** (rather than ignore) incoming signals
 - for instance, when updating a data segment in a database.
- ◆ `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`
- ◆ `how` indicates what specific action `sigprocmask` should take:
 1. `SIG_BLOCK`: set of blocked signals is the union of the current set and the `set` argument.
 2. `SIG_UNBLOCK`: signals in `set` are removed from the current set of blocked signals.
 3. `SIG_SETMASK`: group of blocked signals is set to `set`
- ◆ If `oldset` is **non-null**, the previous value of signal mask is stored in `oldset`.
- ◆ If `set` is **NULL**, the signal mask is unchanged and current value of mask is returned in `oldset` (if it is not `NULL`);

Example with sigprocmask()

```
# include <stdio .h>
# include <stdlib .h>
# include <signal .h>
main (){
sigset_t set1 , set2 ;

sigfillset (&set1 ); // completely full set

sigfillset (&set2 );
sigdelset (&set2 , SIGINT);
sigdelset (&set2 , SIGTSTP ); // a full set minus INT & TSTP

printf(" This is simple code ... \n");
sleep (5) ;
// block everything here !
sigprocmask ( SIG_SETMASK , &set1 , NULL );

printf(" This is CRITICAL code ... \n"); sleep (10) ;

// unblock (allow) all but INT & TSTP
sigprocmask ( SIG_UNBLOCK , &set2 , NULL );
printf(" This is less CRITICAL code ... \n"); sleep (5) ;
// unblock (allow) all signals in set1
sigprocmask ( SIG_UNBLOCK , &set1 , NULL );
printf("All signals are welcome !\n");
exit (0) ;
}
```

```
mema@browser> ./signal-sigprocmask
```

```
This is simple code ...
```

```
^C
```

```
mema@browser> ./signal-sigprocmask
```

```
This is simple code ...
```

```
This is CRITICAL code ...
```

```
^Z
```

```
This is less CRITICAL code ...
```

```
Suspended
```

```
mema@browser> fg
```

```
./signal-sigprocmask
```

```
All signals are welcome!
```

```
mema@browser> ./signal-procmask
```

```
This is simple code ...
```

```
This is CRITICAL code ...
```

```
^C
```

```
This is less CRITICAL code ...
```

```
mema@browser>
```

```
mema@browser> ./signal-sigprocmask
```

```
This is simple code ...
```

```
This is CRITICAL code ...
```

```
^Z^C
```

```
This is less CRITICAL code ...
```

```
mema@browser> fg
```

```
bash : fg: current : no such job
```

```
mema@browser> ./signal-sigprocmask
```

```
This is simple code ...
```

```
This is CRITICAL code ...
```

```
^Z^C^Z^C^Z^C^Z
```

```
This is less CRITICAL code ...
```

```
^\
```

```
Quit
```

```
mema@browser> fg
```

```
bash : fg: current : no such job
```

```
mema@browser> ./signal-sigprocmask
```

```
This is simple code ...
```

```
This is CRITICAL code ...
```

```
This is less CRITICAL code ...
```

```
All signals are welcome !
```

```
mema@browser>
```

What signals to handle?

- ◆ If the program must do some cleanup before terminating, must handle:
 - SIGHUP
 - SIGINT
 - SIGTERM
- ◆ Professional programs try to catch and handle as many signals as possible to enable
 - Cleanup
 - Error recording in log
 - Display of error message
 - “*Internal error #78: contact technical support*”
 - is BETTER THAN
 - “Bus error: core dump”
- ◆ Make signal handler a short function that returns quickly
 - Better to raise a flag and have main program check that flag occasionally (see Rochkind for more)
 - flag should be of type: `volatile sig_atomic_t`