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

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*, *sdtout*, 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 <fnctl.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.

Example usage: 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

Example: createfile

```
# 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@bowser> ./createfile
Managed to get to the file successfully
mema@bowser> 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@bowser> cat > mytest
This is a test
mema@bowser> ./createfile
Managed to get to the file successfully
mema@bowser> ls
a.out   createfile.c   mytest
mema@bowser> more mytest
This is a test
mema@bowser>
```

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).

Example usage: 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

Example: count

```
# 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(argv[1], O_RDONLY))== -1){
      printf (" error in opening  %s \n“, argv[1]);
       exit (1) ;
}

while ((nread = read (filedes, buffer , BUFSIZE)) > 0 )
       total += nread ;
       printf ("Total char in %s %ld \n", argv[1],total);
       exit (0) ;
}
```

```
mema@bowser> ./count Makefile
Total char in Makefile 1989
mema@bowser>
```

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

The write() and close() calls

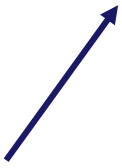
- int write(int fd, char *buf, int count)
  - Writes at most *count* bytes of content from the buffer to the file that is described by *fd*
  - *write()* returns the number of bytes successfully written out to the file, or -1 in case of failure.
  - use the write call with:
    #include <unistd.h>
-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

Example usage: 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

Example: readwriteclose

```
#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);
```

Example: readwriteclose

```
 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;
}
```

```
mema@bowser> ./read-write-close
13 bytes were written.
14 bytes were written.
27 bytes were read
First write. Second Write.
mema@bowser> 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@bowser>
```

Example: writeafterend

```
# 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);
}
```

Example: writeafterend

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

Copying a file with variable buffer size: buffect

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define  SIZE          30
#define  PERM          0644

int mycopyfile(char *name1, char *name2, int BUFFSIZE
      int infile, outfile;
      ssize_t nread;
      char buffer[BUFFSIZE];

    if ( (infile=open(name1,O_RDONLY)) == -1 )
            return(-1);

    if ( (outfile=open(name2, O_WRONLY|O_CREAT|
                              O_TRUNC, PERM)) == -1){
          close(infile);
          return(-2);
          }

    while ((nread=read(infile, buffer, BUFFSIZE)) > 0 ){
           if ( write(outfile,buffer,nread) < nread ){
                  close(infile); close(outfile); return(-3);
                  }
           }
```

Copying a file with variable buffer size: buffect

```
        close(infile); close(outfile);

        if (nread == -1 ) return(-4);
        else    return(0);
}

int main(int argc, char *argv[]){
        int     status=0;

         status=mycopyfile(argv[1],argv[2],atoi(argv[3]));
        exit(status);

}
```

Copying a file with variable buffer size: buffect

```
mema@bowser> time ./buffeffect longFile foo.txt 8192

real    0m0.038s
user    0m0.002s
sys     0m0.033s
mema@bowser> time ./buffeffect longFile foo.txt 4096

real    0m0.034s
user    0m0.003s
sys     0m0.029s
mema@bowser> time ./buffeffect longFile foo.txt 256

real    0m0.163s
user    0m0.020s
sys     0m0.142s
mema@bowser> time ./buffeffect longFile foo.txt 1

real    0m34.870s
user    0m4.204s
sys     0m30.544s
mema@bowser>
```

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.*

The fcntl() system call

- int fcntl(int filedes, int cmd);

  int fcntl(int filedes, int cmd, long arg);

  int fcntl(int filedes, int cmd, struct flock *lock);
- Provides (some) control over already-opened files; headers required: <sys/types.h>, <unistd.h>, <fcntl.h>.
- fcntl() performs one of the operations described below on the open file descriptor filedes. The operation is determined by cmd – values for the cmd appear in the <fcntl.h>.
- Value of 3rd param (arg) depends on what cmd does.
- Among other operations, fcntl() carries out two commands:

  1. F_GETFL: Read file status flags; arg is ignored.
  2. F_SETFL: Set file status flags to value specified by arg.

A routine for checking the flags of an open file

```
#include<fcntl.h>
int filestatus(int filedes) {
      int myfileflags ;
      if ((myfileflags = fcntl(filedes , F_GETFL)) == -1) {
           printf(" file status failure \ n ") ; return (-1) ;
      }
      printf("file descriptor : % d", filedes) ;
      // test against the open file flags
      switch (myfileflags & O_ACCMODE) {
      case O_WRONLY:
            printf("write - only"); break;
      case O_RDWR:
            printf("read - write"); break;
      case O_RDONLY:
      printf("read - only") ; break;
      default :
      printf ("no such mode") ;
}
      if (myfileflags & O_APPEND)
            printf (" - append flag set") ; printf (" \ n ");
      return (0);
}
```

- *& : bitwise AND operator*
- *fcntl can be used to acquire record locks (or locks on file segments).*

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*.

Example: dupdup2

```
#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("dupdup2file, 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);
```

Example: dupdup2

```
 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;
}
```

Example: dupdup2

```
mema@bowser> ./dupdup2
fd1 = 3
fd2 = 4
fd3 = 0
mema@bowser> ls -l dupdup2file
-rw-r--r-- 1 mema mema 17 Dec 29 07:58 dupdup2file
mema@bowser> more dupdup2file
What time is it?
mema@bowser>
```

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
- butf->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");
```

Example: samplestat

```
#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("Time/Date  : %s",ctime(&statbuf.st_atime));
   printf("-------------------------------\n");
   printf("entity name: %s\n",argv[1]);
   printf("accessed   : %s", ctime(&statbuf.st_atime));
   printf("modified   : %s", ctime(&statbuf.st_mtime));
   }
return 1;
}
```

Example: samplestat

```
mema@bowser> ./samplestat
Failed to get file status: Bad address
mema@bowser> ./samplestat f1
Time/Date  : Sun Mar 19 10:46:27 2023
-------------------------------
entity name: f1
accessed   : Sun Mar 19 10:46:27 2023
modified   : Sun Mar 19 10:46:06 2023
mema@bowser>
```

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.

Example: openreadclosedir

```
#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("inode %d of the entry %s \n", \
                  (int)direntp->d_ino, direntp->d_name);
        closedir(dir_ptr);
        }
}
}
```

Example: openreadclosedir

```
int main(int argc, char *argv[]) {

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

Example: openreadclosedir

```
mema@bowser> ./openreadclosedir direct1/
direct1/:
inode 2850201 of the entry .
inode 2702911 of the entry ..
inode 2850223 of the entry longFile
inode 2850218 of the entry foo.txt
inode 2850213 of the entry bla
mema@bowser>
```

Program that behaves like ls –al (morewithls)

```
#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 *);

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

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

Program that behaves like ls –al (morewithls)

```
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
  }
}
```

Program that behaves like ls –al (morewithls)

```
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);
}
```

Program that behaves like ls –al (morewithls)

```
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);  /*try without 4 */
}
```

Program that behaves like ls –al (morewithls)

```
mema@bowser> ./morewithls 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@bowser>
```

link and *unlink*

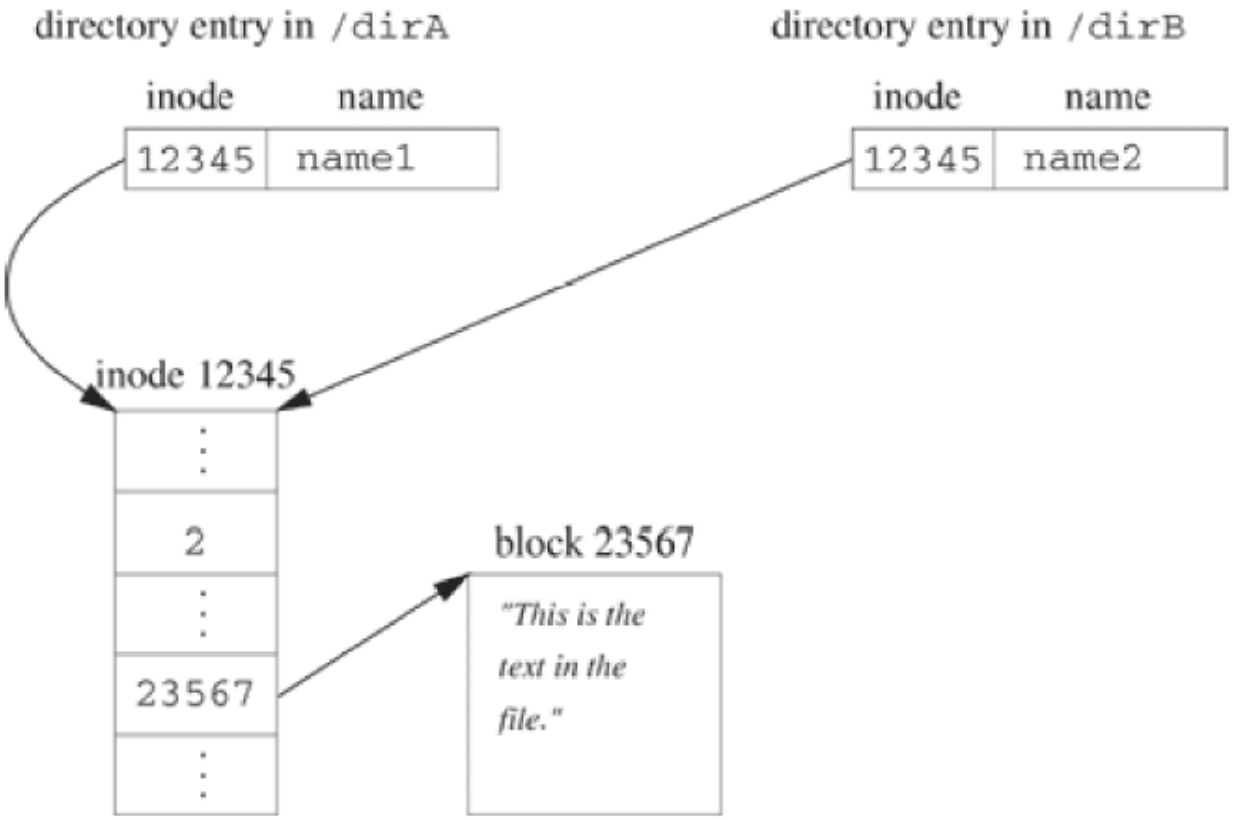
- *int unlink(char *pathname)*
- Deletes a name from the file system; if that name is the last link to a file and no other process has the file open, the file is deleted and its space is made available.

- *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*.

Example usage: link()

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

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



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

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

mkdir and *readdir*

- *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

Signals

- Signals provide a simple method to transmit software interrupts (i.e., notifications of events) to processes. They occur asynchronously when:
  - There is an error during the execution of a job
  - Events are created with the help of input devices (cntrl-z, cntrl-c, cntrl-\ etc.).
  - A process notifies another one about an event.
  - Issuing of a kill command to a job.
- Signals are identified with an integer number.
  - a unique number represent a different type of signal.
- Signals provide a way to handle asynchronous events: e.g., a user at a terminal typing the interrupt key to suspend a program in execution.

Signals

- Signals take place at what appears to be "random time" to the process.
- We can ask the kernel to do one of the following things when a signal occurs:
  - Ignore the signal (two signals, however, can never be ignored: SIGKILL & SIGSTOP)
  - Catch the signal (we do that by informing the kernel to call a "signal-handling" function of ours whenever a signal occurs)
  - Let the default action apply (every signal has a default action)
  - Block the signal (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; Abnornal 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.

Sending a signal with kill command

Shell command:

kill [-signalName] processId

kill -s signalName processId

send a specific signal to process(es)

kill -l (lists all available signals)

Examples:

kill -USR1 3424

kill -s USR1 3424

kill -9 3424

```
mema@bowser> kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN +1 36) SIGRTMIN +2 37) SIGRTMIN +3
38) SIGRTMIN +4 39) SIGRTMIN +5 40) SIGRTMIN +6 41) SIGRTMIN +7 42) SIGRTMIN +8
43) SIGRTMIN +9 44) SIGRTMIN +10 45) SIGRTMIN +11 46) SIGRTMIN +12 47) SIGRTMIN +13
48) SIGRTMIN +14 49) SIGRTMIN +15 50) SIGRTMAX -14 51) SIGRTMAX -13 52) SIGRTMAX -1
53) SIGRTMAX -11 54) SIGRTMAX -10 55) SIGRTMAX -9 56) SIGRTMAX -8 57) SIGRTMAX -7
58) SIGRTMAX -6 59) SIGRTMAX -5 60) SIGRTMAX -4 61) SIGRTMAX -3 62) SIGRTMAX -2
63) SIGRTMAX -1 64) SIGRTMAX
mema@bowser>
```

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.

The *signal* system call

- *#include<signal.h>*

  *void (*signal (int signum,*

  *void (*handler) (int))) (int)*

- The *signal()* call installs a new signal handler for the signal with number *signum*. The signal handler is set to the handler which may be a user-specified function or either SIG_IGN or SIG_DFL.
- *signal()* returns the previous value of the signal handler, or SIG_ERR on error.
- This call is the traditional way of handling signals.

Example: sigSimExample

```
#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);
  }
}

/* no explicit call to function f */
void f(int signum){
/* re-establish signal-handing function for SIGINT */
  signal(SIGINT, f);
  printf("OUCH!\n");
}
```

Example: sigSimExample

```
mema@bowser>  ./sigSimExample
hello
hello
hello
hello
^COUCH!
hello
mema@bowser>
```

Example: sigIgnore

```
#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.

Example: sigIgnore

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

The pause(), raise() calls

- int pause(void);
  - Causes the invoking process or thread to sleep until a signal is received that either terminates it (i.e., process or thread) or causes it to call a signal handler
  - Returns when a signal is caught and the signal-handing function has run and returned. In this case, pause returns -1, and errno is set to EINTR
  - #include <unistd.h>
- int raise(int signalId);
  - Sends signalId to the invoking process; it is equivalent to: kill(getpid(), signalId)
  - Returns 0 on success, non-zero on failure

  E.g.:

  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 for good!\n");
  while (1);
  printf("This line should never be executed.\n");
}
```

```
mema@bowser> date;   ./alarm;  date
Mon Apr 12 21:20:41 EEST 2011
Looping for good!
Alarm clock
Mon Apr 12 21:20:44 EEST 2011
mema@bowser>
```

Example: alarmpause

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

main(){
  void  wakeup(int);

  printf("about to sleep for 5 seconds \n");
  signal(SIGALRM, wakeup);

  alarm(5);
  pause();  /* pauses the process until a sig arrives */
  printf("Hola Amigo!  Un abrazo!\n");
}

void wakeup(int signum){
  printf("Alarm received from kernel\n");
}
```

Example: alarmpause

```
mema@bowser> ./alarmpause
about to sleep for  seconds
Alarm received from kernel
Hola Amigo!  Un abrazo!
mema@bowser>
```

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 signal handler begins execution, another signal may occur.
2. The second signal would cause the default action which is to *terminate* the process.
3. An unsuccessful effort is to (re-)state the signal's handler as the 1st line of the handler.
4. Although the program may seem to work, this is not "bullet-proof" as we may "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, the 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

Example: lostsig

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
void foohandler(int);
int flag=0;
int main(){
  int lpid=0;
  printf("The process ID of this program is %d \n",getpid());
  lpid=getpid();
  signal(SIGINT, foohandler);
  while (flag==0){
        kill(lpid, SIGINT);
        printf(“flag is %d\n”, flag); fflush(stdout);
        pause();
        printf(“Hello! \n”);
        }
}
void foohandler(int signum){
     signal(SIGINT, foohandler);  /* re-establish
                                     handler for next time */
     flag=1;
}
```

Example: lostsig

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

```
mema@bowser> ./lostsig
The process ID of this program is 3245
flag is 1
```

- 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.

Example: lostsig

- Forcing a second interrupt with control-C, we terminate the program (by getting out of the loop):

```
mema@bowser> ./lostsig
The process ID of this program is 3245
flag is 1
^CHello!
mema@bowser>
```

- **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_handle*r 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!

Example: sigaction

```
# include < stdio .h >
# include < stdlib .h >
# include < signal .h >
void catchinterrupt (int signo ){
printf("\ nCatching : 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@bowser> ./sigaction
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@bowser> ./sigaction
sleep call #1
sleep call #2
^C
Catching : signo =2
Catching : returning
sleep call #3
sleep call #4
Exiting
mema@bowser>
```

Changing the Behavior of program in interrupt: sigaction2

```
# 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@bowser> ./sigaction2
sleep call #1
^Csleep call #2
^Z^Csleep call #3
sleep call #4
sleep call #5
^Zsleep call #6
Exiting
mema@bowser> ./sigaction2
sleep call #1
sleep call #2
sleep call #3
sleep call #4
sleep call #5
^C
mema@bowser> ./sigaction2
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@bowser>
```

Restoring a *previous* action: sigaction3

```
# 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@bowser> ./sigaction3
Saving the default way of handling the control =c
sleep call #1
^C
mema@bowser>
mema@bowser>
mema@bowser>
mema@bowser> ./sigaction3
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^CRestablishing old way of handling control-c
sleep call #3
^C
mema@bowser>
```

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: 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@bowser> ./sigprocmask
This is simple code ...
^C
mema@bowser> ./sigprocmask
This is simple code ...
This is CRITICAL code ...
^Z
This is less CRITICAL code ...
Suspended
mema@bowser> fg
./signal-sigprocmask
All signals are welcome!
mema@bowser> ./sigprocmask
This is simple code ...
This is CRITICAL code ...
^C
This is less CRITICAL code ...

mema@bowser>
```

```
mema@bowser> ./sigprocmask
This is simple code ...
This is CRITICAL code ...
^Z^C
This is less CRITICAL code ...
mema@bowser> fg
bash : fg: current : no such job
mema@bowser> ./sigprocmask
This is simple code ...
This is CRITICAL code ...
^Z^C^Z^C^Z^C^Z
This is less CRITICAL code ...
^\
Quit
mema@bowser> fg
bash : fg: current : no such job
mema@bowser> ./sigprocmask
This is simple code ...
This is CRITICAL code ...
This is less CRITICAL code ...
All signals are welcome !
mema@bowser>
```

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