Error Handling
Low-Level I/O
Signals

March 2017

Error Handling

- 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)
  ```

  - The above prints out the string pointed to by `estring` denoting a specific kind of a mistake (choice of the programmer).
- Should 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.

C program with Error Handling

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

int main(){
   FILE *fp=NULL;  char *p=NULL; int stat=0;

   fp=fopen("a_non_existent_file","r");
   if (fp == NULL) {
    printf("errno = %d \n", errno);
    perror("fopen");
    }

   p=(char *)malloc(2147483647);
   if (p==NULL) {
    printf("errno = %d \n",errno);
    perror("malloc");
    }
   else {
    printf("Carry on\n");
    }

   stat=unlink("/etc/motd");
   if (stat == -1) {
    printf("errno = %d \n",errno);
        perror("unlink");
    }

   return(1);
}
```

Running the errors_demo.c executable

```
ad@thales:~/src$ gcc errors_demo.c
ad@thales:~/src$ ./a.out
errno = 2
fopen: No such file or directory
Carry on
errno = 13
unlink: Permission denied
ad@thales:~/src$
```

Low-Level Input/Output

- The `stdio` library enables the average user carry out I/Os 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 when
  1. the amenities that `stdio` are not desirable (for whatever reason) in accessing files/devices, or
  2. interprocess communication (IPC) occurs with the help of pipes/sockets.

Low-Level I/Os

- 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` where respective identifiers are file pointers (for formatted I/O).

- Designated (fixed) file descriptors:

  0 : standard input
  1 : standard output
  2 : standrad 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 parent-"inherited" to any child process that the parent in question creates.

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` designate the way (i.e., a number) with which the file can be accessed; the value 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 exists.
  - `O_TRUNC`: size of file is to be truncated to 0, if file exists.

The `open()` system call

- required: `#include <fnctl.h>`
  $\Rightarrow$ `fnctl.h` defines all these (and more) flags.
- The not-compulsory `mode` parameter is an integer that designates the desired access primitives 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.

createfile.c

```
#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 permissions

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);
}
```

Running the executable for `createfile.c`

```
ad@thales:~/src$ gcc createfile.c
ad@thales:~/src$ ./a.out
Managed to get to the file successfully
ad@thales:~/src$ ls -l
total 20
-rwxr-xr-x 1 ad ad 8442 2010-04-06 21:50 a.out
-rw-r--r-- 1 ad ad  375 2010-04-06 21:49 createfile.c
-rw-r--r-- 1 ad ad  506 2010-04-06 16:24 errors_demo.c
-rw-r--r-- 1 ad ad    0 2010-04-06 21:50 mytest
ad@thales:~/src$ cat > mytest
This is Kon Tsakalozos
ad@thales:~/src$ ./a.out
Managed to get to the file successfully
ad@thales:~/src$ ls
a.out  createfile.c  errors_demo.c  mytest
ad@thales:~/src$ more mytest
This is Kon Tsakalozos
ad@thales:~/src$
```

Setting modes with symbolic names

S_IRWXU	00700 owner has read, write and execute permission
S_IRUSR	00400 owner has read permission
S_IWUSR	00200 owner has write permission
S_IXUSR	00100 owner has execute permission
S_IRWXG	00070 group has read, write and execute permission
S_IRGRP	00040 group has read permission
S_IWGRP	00020 group has write permission
S_IXGRP	00010 group has execute permission
S_IRWXO	00007 others have read, write and execute permission
S_IROTH	00004 others have read permission
S_IWOTH	00002 others have write permission
S_IXOTH	00001 others have execute permission

Working with access modes

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

1. If the call to `open()` is successful, the file is opened for reading/writing by the user.
2. Those in the “group” and “others” can read the file.

The `creat()` call

```
int creat(char *pathname, mode_t mode);
```

- The `creat` is an alternative way to create a file (istead of using `open()`).
- `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/tsak",0644);
```

is equivalent to:

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

The `read()` call

```
ssize_t read(int filedes, char *buffer, size_t n)
```

- Reads at most `n` bytes from a file, device, end-point of a pipe, socket that is designated by `filedes` and place the bytes on `buffer`.

- 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?
  1. The file has less characters left to be read.
  2. The operation is "interrupted" by a signal.
  3. Reading on pipe/socket takes place and a character becomes available (in which case a while-loop is needed to read all characters).

Using the read() call (count.c)

```
#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);
}
```

Running the executable:

```
ad@thales:~/src$ ./a.out
Total char in anotherfile 936
ad@thales:~/src$
```

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

The write() and close() system calls

```
ssize_t write(int filedes, char *buffer, size_t n);
```

- The call writes at most `n` bytes of content from the `buffer` to the file that is described by `filedes`.
- `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 filedes);
```

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

Working with open, read, write and close calls

Write a program that appends the content of a file at the very end of the content of another file.

```
#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);
}
```

Execution Outcome:

```
ad@thales:~/src$ ls
anotherfile    count.c        dupdup2file     mytest
                writeafterend.c
a.out          createfile.c   errors_demo.c   mytest1
buffeffect.c   dupdup2.c      filecontrol.c   readwriteclose.c
ad@thales:~/src$ more mytest
This is Konstantinos Tsakalozos
ad@thales:~/src$ more mytest1
that I use to show something silly
use to show something silly
to show something silly
ad@thales:~/src$ ./a.out
Usage: ./a.out from-file to-file
ad@thales:~/src$ ./a.out mytest mytest1
ad@thales:~/src$ cat mytest1
that I use to show something silly
use to show something silly
to show something silly
This is Konstantinos Tsakalozos
ad@thales:~/src$
```

Using open read, write and close calls

```c
#include  <stdio.h>
#include  <stdlib.h>
#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);
  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);
}
```

Running the program..

```
ad@thales:~/src$ ls
anotherfile   count.c       errors_demo.c   readwriteclose.c
a.out         createfile.c  mytest
ad@thales:~/src$ ./a.out
13 bytes were written.
14 bytes were written.
27 bytes were read
First write. Second Write.
ad@thales:~/src$ ls
anotherfile   count.c       errors_demo.c   readwriteclose.c
a.out         createfile.c  mytest          t
ad@thales:~/src$
```

Copying a file with variable buffer size

```c
#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);
                        }
                }
        close(infile); close(outfile);
```

Copying a file with variable buffer size

```
        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);
}
```

Running the program for various size buffers..

```
ad@thales:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 8192
real    0m0.012s user   0m0.000s sys    0m0.012s
ad@thales:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 4096
real    0m0.010s user   0m0.000s sys    0m0.008s
ad@thales:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 256
real    0m0.071s user   0m0.000s sys    0m0.072s
ad@thales:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 32
real    0m0.454s user   0m0.012s sys    0m0.444s
ad@thales:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 1
real    0m13.738s user  0m0.428s sys    0m13.305s
ad@thales:~/src$
```

`lseek` call

```
off_t lseek(int filedes, off_t offset, int start_flag);
```

- `lseek` repositions the offset of the open file associated with `filedes` to the argument `offset` according to the directive `start_flag` 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

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

Positions the read/write pointer 32 bytes BEFORE the end of the file.

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

```c
#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);
    switch ( myfileflags & O_ACCMODE ){ //test against the open file flags
    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`

```
int dup(int oldfd);
```

returns the lowest-numbered unused descriptor as the new descriptor.

```
int dup2(int oldfd, int newfd);
```

makes `newfd` be the copy of `oldfd` - 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 of dup and dup2

```c
#include  <stdio.h>
#include  <stdlib.h>
#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);

  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;
}
```

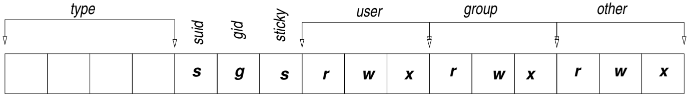
Execution Outcome:

```
ad@thales:~/src$ ls
anotherfile    count.c        dupdup2file    mytest
a.out          createfile.c   errors_demo.c  readwriteclose.c
buffeffect.c   dupdup2.c      filecontrol.c
ad@thales:~/src$ ./a.out
fd1 = 3
fd2 = 4
fd3 = 0
ad@thales:~/src$ ls
anotherfile    count.c        dupdup2file    mytest
a.out          createfile.c   errors_demo.c  readwriteclose.c
buffeffect.c   dupdup2.c      filecontrol.c
ad@thales:~/src$ cat dupdup2file
What time is it?
ad@thales:~/src$
```

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 helps "carry" all derived information.
- such information includes:
  1. `buff→st_dev`: ID of device containing file
  2. `buff→st_ino`: inode number
  3. `buff→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 catalog, if outcome is S_IFREG, the mode is a regular file etc.)
  4. `buff→st_nlink`: number of hard links
  5. `buff→st_uid`: user-ID of owner
  6. `buff→st_gid`: group ID of owner
  7. `buff→st_size`: total size, in bytes
  8. `buff→st_atime`: time of last access
  9. `buff→st_mtime`: time of last modification of content
  10. `buff→st_ctime`: time of last status change

st_mode is a 16-bit quantity



1. 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).
2. the next three bits set the flags: `set-user-ID`, `set-group-ID` and the `sticky` bits respectively.
3. next three groups of 3 bits a piece indicate the *read/write/execute* access right for the the groups: `owner`, `group` and `others`.
4. masking can be used to decipher the permissions each file system entity is given.

stat-ing inodes

- 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 fileds `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
- 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.

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 sspecial */
#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");
```

Accessing information from `inode`

```c
#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)+4);
                printf("modified    : %s", ctime(&statbuf.st_mtime));
                }
        return(1);
}
```

Running the program..

```
ad@haiku:~/src-set004$ ./samplestat git.pdf
Time/Date   : Mon Mar 21 10:12:30 2016
---------------------------------
entity name: git.pdf
accessed    : Mar 21 10:12:30 2016
modified    : Mon Mar 21 10:11:55 2016
ad@haiku:~/src-set004$
```

Accessing Catalog Content

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

calls: `opendir`, `readdir`, `closedir`

- `DIR *opendir(char *name)`:
  1. Opens up the catalog termed `name` and returns a pointer type `DIR` for accessing the catalog.
  2. If there is a mistake, the call returns NULL

- `struct dirent *readdir(DIR *dirp);`
  1. the call returns a pointer to a `dirent` structure representing the next directory entry in the directory pointed to by `dirp`
  2. if for the current entry, the field `d_ino` is 0, the respective entry has been deleted.
  3. returns NULL if there are no more entries to be read.

- `int closedir(DIR *dirp);`
  1. closes the directory associated with `dirp`
  2. function returns 0 on success. On error, -1 is returned, and `errno` is set appropriately.

Example

```
#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);
        }
}

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

Execution Outcome

```
ad@haiku:~/src-set004$ ./openreadclosedir
inode 11403323 of the entry myreadlink
inode 11403324 of the entry myctime
inode 11403322 of the entry .
inode 11403325 of the entry dupdup2
inode 11403326 of the entry signal-example
inode 10883777 of the entry count
inode 11403328 of the entry myalarm1.c
inode 11403310 of the entry errors_demo
inode 11403330 of the entry signal-ignore.c
inode 11403331 of the entry morewithls.c
inode 11403332 of the entry myalarm.c
inode 11403393 of the entry openreadclosedir.c
inode 10883835 of the entry t
inode 11403335 of the entry myreadlink.c
inode 11403336 of the entry samplestat.c
inode 11403305 of the entry ..
inode 11403337 of the entry signal-exampleD
inode 10883705 of the entry createfile
inode 11403339 of the entry jj.ps

.....
ad@haiku:~/src-set004$ ./openreadclosedir
```

Creating a program that behaves as `ls -la`

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
                            /* eight distinct modes */
char *modes[]={"---","--x","-w-","-wx","r--","r-x","rw-","rwx"};
void list(char *);
void printout(char *);

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

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

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      */
  }
}
```

Creating a program that behaves as `ls -la`

```c
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);
}
```

Creating a program that behaves as `ls -la`

```
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, (int)mybuf.st_nlink, mybuf.st_uid, \
                mybuf.st_gid, (int)mybuf.st_size, \
                ctime(&mybuf.st_mtime)+4, name); /* try without 4 */
}
```

```
ad@haiku:~/src-set004$ ./morewithls mydir morewithls.c
drwx------ 10  1000/1000      4096 Mar  9 07:51 mydir/.
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/b
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/e
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/d
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/a
drwx------  4  1000/1000      4096 Mar 12 13:24 mydir/..
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/f
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/h
-rwxr-xr-x  1  1000/1000       750 Mar  9 07:51 mydir/j
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/g
-rwxr-xr-x  1  1000/1000        12 Mar  9 07:51 mydir/k
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/c
-rwxr-xr-x  1  1000/1000       368 Mar  9 07:51 mydir/i
-rwxr-xr-x  1  1000/1000      1680 Mar 12 13:18 morewithls.c
ad@haiku:~/src-set004$
```

`link` and `unlink`

```
int link(char *oldpath, char *newpath)
```

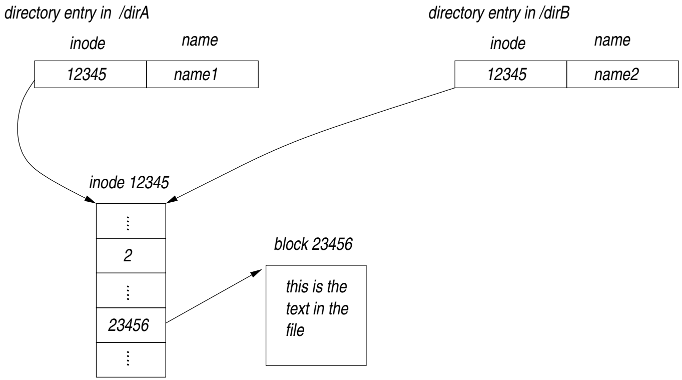
- It creates an new hard link to an existing file.
  If `newpath` exists, it will not be overwritten.
- The created link essentially connects the inode of the `oldpath` with the name of the `newpath`.

```
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 have the file open, the file is deleted and its space is made available.

Example on `link()`

```
#include <stdio.h>
#include <unistd.h>
....
if ( link("/dirA/name1","/dirB/name2")== -1 )
    prerror("Failed to make a new hard 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 `path` name 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 content of the symbolic link `path` in the buffer `buf` that has size `bufsiz`.
- On success, `readlink` returns the number of bytes placed in `buf`; otherwise, -1.

Signals

- Signals provide a simple method to transmit software interrupts to processes. They occur asynchronously when:
    - There is an error during the execution of a job.
    - Events created with the help if 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 integer number.
    - a unique number represent a different type of signal.
- Signals provide a way to handle asynchronous events:
  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 though can never be ignored: `SIGKILL` & `SIGSTOP`).
    - Catch the signal (we do that by informing the kernel to call a function of ours whenever a signal occurs).
    - Let the default action apply (every signal has a default action)

Some of the POSIX Signals

```
                             Action
                             -----
       SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2       Term    Interrupt from keyboard
       SIGQUIT       3       Core    Quit from keyboard
       SIGILL        4       Core    Illegal Instruction
       SIGABRT       6       Core    Abort signal from abort(3)
       SIGBUS        7       Core    Bus error (bad memory access)
       SIGFPE        8       Core    Floating point exception
       SIGKILL       9       Term    Kill signal
       SIGSEGV      11       Core    Invalid memory reference
       SIGPIPE      13       Term    Broken pipe: write to pipe with no
                                     readers
       SIGALRM      14       Term    Timer signal from alarm(2)
       SIGTERM      15       Term    Termination signal
       SIGUSR1      10       Term    User-defined signal 1
       SIGUSR2      12       Term    User-defined signal 2
       SIGCHLD      17       Ign     Child stopped or terminated
       SIGCONT      18       Cont    Continue if stopped
       SIGSTOP      19       Stop    Stop process
       SIGTSTP      20       Stop    Stop typed at tty
       SIGTTIN      21       Stop    tty input for background process
       SIGTTOU      22       Stop    tty output for background process
```

Actions

The "*Action*" column above specifies the $\boxed{\textit{default disposition}}$ for each (how the process behaves when it is delivered the signal):

- `Term`: Default action is to **terminate** the process.
- `Ign`: Default action is to **ignore** the signal.
- `Core`: Default action is to terminate the process & **dump-core**.
- `Stop`: Default action is to **stop** the process.
- `Cont`: Default action is to **continue** the process if it is currently stopped.

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

Sending a signal with `kill`

- `kill [ -signal ] pid ...`

  `kill [ -s signal ] pid ...`

  send a specific signal to process(es)

```
kill -USR1 3424
kill -s USR1 3424
kill -9 3424
```

- `kill -l [signal]`: lists all available signals

```
ad@sydney:~/Set004$ 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-12
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
ad@sydney:~/Set004$
```

Sending a signal to a process through the `kill` call

```
int kill(pid_t pid, int sig);
```

- Signal `sig` is sent to process with `pid`
- `#include <sys/types.h>`
  `#include <signal.h>`
- 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>
  typedef void (*sighandler_t)(int);
  sighandler_t  signal(int signum, sighandler_t handler);
  ```
- The `signal()` call installs a new signal handler for the signal with number `signum`.
  - The signal handler is set to `handler`.
  - `handler` may be user-specified function, or `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

```c
#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           */
  signal(SIGINT, f);   /* re-establish disposition of the signal SIGINT   */
  printf("OUCH!\n");
}
```

```
ad@sydney:~/src$ ./a.out
hello
hello
^COUCH!
hello
hello
^COUCH!
hello
^COUCH!
ad@sydney:~/src$
```

Ignoring a Signal

```c
#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 */
```

```
ad@sydney:~/Desktop/Set004/src$ ./a.out
you can't stop me here!
haha
haha
haha
^Chaha
haha
haha
^Ghaha
haha
^Chaha
haha
haha
^\Quit
ad@sydney:~/Desktop/Set004/src$
```

The `pause()`, `raise()` calls

```
int pause(void);
```

- `#include <unistd.h>`
- causes the *invoking process or thread* to sleep until a signal is received that either terminates it (i.e., process of thread) or causes it to call a signal-handler.
- returns when a signal was caught and the signal-handling function returned. In this case `pause` returns -1, and `errno` is set to `EINTR`.

```
int raise(int sig);
```

- `#include <signal.h>`
- sends `sig` to the invoking process; it is equivalent to: `kill(getpid(), sig);`
- returns 0 on success, non-zero for failure.

The alarm call

```
unsigned int alarm(unsigned int seconds);
```

- `#include <unistd.h>`
- delivers a `SIGALRM` to invoking process in *seconds*.
- any previously set `alarm()` is cancelled.
- returns the number of seconds remaining until any previously scheduled alarm was due to be delivered; otherwise, 0.

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

main(){
    alarm(3); // schedule an alarm signal
    printf("Looping for good!\n"); fflush(stdout);
    while (1) ;
    printf("This line should be never part of the output\n"); fflush(stdout);
    }
```

```
ad@sydney:~/src$ date; ./a.out ; date
Mon Apr 12 22:20:41 EEST 2010
Looping for good!
Alarm clock
Mon Apr 12 22:20:44 EEST 2010
ad@sydney:~/src$
```

Example with signal, alarm and pause

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

void  wakeup(int);

main(){
  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");
  }
```

```
ad@haiku:~/src-set004$ ./signal-alarmpause
about to sleep for 5 seconds
Alarm received from kernel
Hola Amigo! Un abrazo!
ad@haiku:~/src-set004$
```

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 call to `sig_int` is in the signal handler body, another signal may occur!
2. The second signal would cause the default action: this may force the process to *terminate*.
3. A unsuccessful effort is to (re-)state the signal's expected disposition as the *1st line* of the handler..
4. Although this may occassionally appear to work correctly, the mechanism is not "bullet–proof" as we may "lose" a signal along the way.

Unreliable Signals

- A process could “ignore” signals (with a trick):

```
int my_sig_flag=0;
    ...
main(){
    int my_sig_int();
    ...
    signal(SIGINT, my_sig_int);
    ...
    while (my_sig_flag == 0){
                /* point with possible problem in here */
                pause();
                }
    ...
}

my_sig_int(){
    signal(SIGINT, my_sig_int);
    my_sig_flag=1;
    }
```

◇ Under “regular” circumstances the process would “pause” until it received a `SIGINT` and then, it continue on to other actions past the `while` statement; the `while` predicate would disqualify.

Unreliable Signals

There is a small chance that things would go wrong...

1. If the signal takes place **after** the predicate evaluation **but before** the call to *pause*, the process could go on to sleep for ever! (provided that another signal is not generated)
2. In the above scenario the signal is *lost*!
3. Such code is not correct yet it works most of the times...

Unreliable Signal-ing

```
#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);    /* lost signal?? */
        printf("flag is %d\n", flag); fflush(stdout);
        pause();
        printf("Hello! \n");
        }
}

void foohandler(int signum){   /* no explicit call to handler foo */
  signal(SIGINT, foohandler);  /* re-establish handler for next time */
  flag=1;
}
```

Unreliable signal-ing

• Running the program, we get into the *pause* (the first signal does not appear to get into handler):

```
ad@haiku:~/src-set004$ ./signal-exampleA
The process ID of this program is 22792
flag is 1
```

The (first) signal *seems* to be "lost" for the time being..

• Forcing now an interrupt with *control-C*, we terminate the program (by getting out of the loop):

```
ad@haiku:~/src-set004$ ./signal-exampleA
The process ID of this program is 22792
flag is 1
^CHello!
ad@haiku:~/src-set004$
```

⊙ *Signal Sets* provide a (*POSIX*) reliable way to deal with signals.

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 add-ing signals or by using a full set and then by selectively delete-ing certain signals.
- Initialization of signals happens through:
  - `int sigemptyset(sigset_t *set);`
  - `int sigfillset(sigset_t *set);`
- Manipulation of signals sets happens via:
  - `int sigaddset(sigset_ *set, int signo);`
  - `int sigdelset(sigset_ *set, int signo);`
- Membership in a signal set:
  - `int sigismember(sigset_t *set, int signo)`

Example in 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
...
sigdelset(&mask2, SIGCHLD); // remove signal SIGCHLD
....
...
```

- `mask1` is created entirely empty.
- `mask2` is created entirely full.

sigaction() call

- Once a set has been defined, we can elect a specific method to handle a signal with the help of sigaction().

```
int sigaction(int signo, const struct sigaction *act,
                          struct sigaction *oldact);
```

- The sigaction structure is:

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

Elements of the sigaction structure (a)

- 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.

Elements of the `sigaction` structure (b)

- `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.
- Use either `sa_handler` or `sa_sigaction` but not both!

Use of sigaction

```c
#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 & carries on

```
ad@ad-desktop:~/Set004/src$ ./a.out
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
ad@ad-desktop:~/Set004/src$
```

```
ad@ad-desktop:~/Set004/src$ ./a.out
sleep call #1
sleep call #2
^C
Catching: signo=2
Catching: returning
sleep call #3
sleep call #4
Exiting
ad@ad-desktop:~/Set004/src$
```

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 the 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);
    }
```

Running the Program...

```
ad@ad-desktop:~/Set004/src$ ./a.out
./a.out
sleep call #1
^Csleep call #2
^Z^Csleep call #3
sleep call #4
sleep call #5
^Zsleep call #6
Exiting
ad@ad-desktop:~/Set004/src$ ./a.out
sleep call #1
sleep call #2
sleep call #3
sleep call #4
sleep call #5
^C
ad@ad-desktop:~/Set004/src$ ./a.out
sleep call #1
^Csleep call #2
^C^Z^Zsleep call #3
^Z^Zsleep call #4
^Z^Zsleep call #5
^Z^Zsleep call #6
^ZExiting
ad@ad-desktop:~/Set004/src$
```

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 to old way of handling\n");
    sigaction(SIGINT, &oldact, NULL);
    printf("sleep call #3\n"); sleep(4);

    printf("Exiting \n");
    exit(0);
    }
```

Example in restoring a *previous* action

```
ad@ad-desktop:~/Set004/src$ ./a.out
Saving the default way of handling the control=c
sleep call #1
^C
ad@ad-desktop:~/Set004/src$
ad@ad-desktop:~/Set004/src$
ad@ad-desktop:~/Set004/src$
ad@ad-desktop:~/Set004/src$
ad@ad-desktop:~/Set004/src$ ./a.out
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^C^C^C^C^C^CRestablishing to old way of handling
sleep call #3
^C
ad@ad-desktop:~/Set004/src$
```

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_SETMASK`: group of blocked signals is set to `set`
  2. `SIG_BLOCK`: set of blocked signals is the union of the current set and the `set` argument.
  3. `SIG_UNBLOCK`: signals in `set` are removed from the current set of blocked signals.
- 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);

Code snippet using `sigprocmask()`

```c
#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 set minus INT & TSTP

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

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

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

Working with the `sigprocmask()`

```
ad@ad-desktop:~/Set004/src$ ./a.out
This is simple code...
^C
ad@ad-desktop:~/Set004/src$ ./a.out
This is simple code...
This is CRITICAL code...
^Z^Z^C^C^X^X^C^C^Z^Z
This is less CRITICAL code...
^C
ad@ad-desktop:~/Set004/src$ ./a.out
This is simple code...
This is CRITICAL code...
^Z^C^Z^C^Z^C^Z
This is less CRITICAL code...
^\Quit
ad@ad-desktop:~/Set004/src$ fg
bash: fg: current: no such job
ad@ad-desktop:~/Set004/src$ ./a.out
This is simple code...
This is CRITICAL code...
This is less CRITICAL code...
All signals are welcome!
ad@ad-desktop:~/Set004/src$
ad@ad-desktop:~/Set004/src$
```

About Signals..

- When a signal is dispatched and the receiving process executes a sys-call, the signal has no effect until the sys-call compeletes.

  • *Exception*: a few calls such as `read/write/open` on slow devices could be interrupted by a signal.
- In general, signals cannot be stacked (ie, you can never have more than one signal of each type outstanding at any moment).

  • In this context, when it comes to reliability there are always some questions as a process can never be sure that a signal has not been "lost".
- The effectiveness of signals with respect to IPC is somewhat limited.