CS214 Recitation Sec.7

Oct. 17, 2017

Topics

1. Processes (ps/top)

2. Forking (pid/ppid)

3. HW4 answers

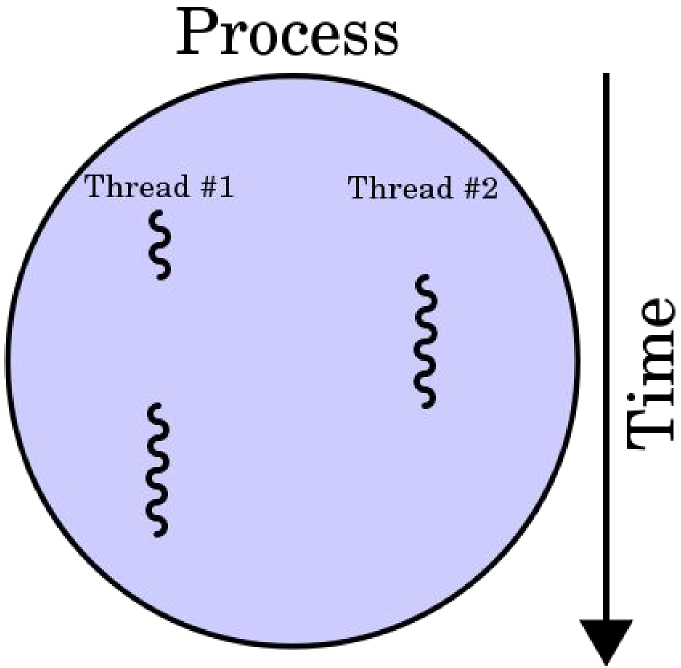
4. HW5: Implementing “ps” command in C

Process vs Thread

A *process* is basically a program in execution. It is fundamentally a container that holds all the resources needed to run a program.

A thread is the smallest entity scheduled for execution on the CPU. *In general, a thread is a component of a process.*

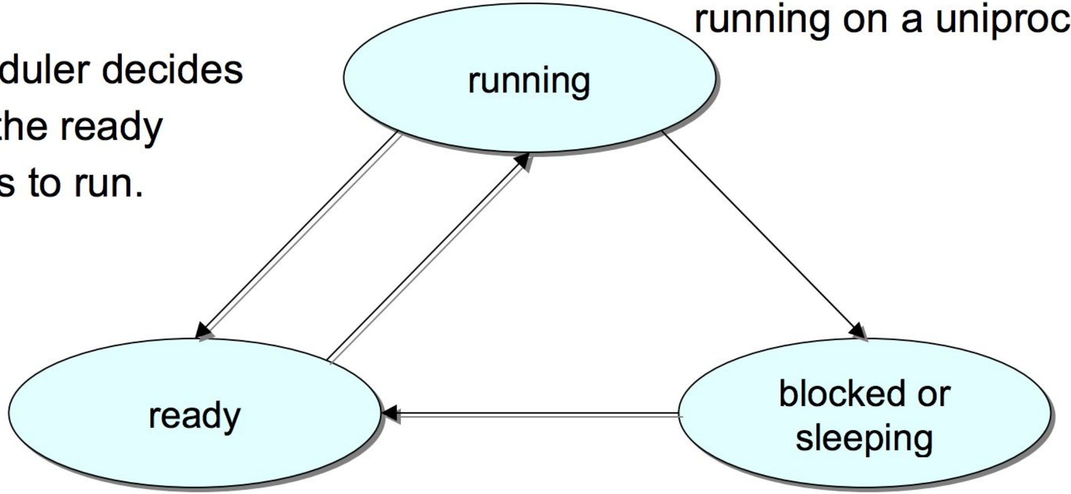
The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.



Process states (main memory)

Only one process can be running on a uniprocessor

The scheduler decides which of the ready processes to run.



A process is ready if it could use the CPU immediately.

A process is blocked if it waiting for an event (I/O, signal)

“ps” command

process status: *ps [options]*

The *ps* utility displays a header line, followed by lines containing information about all of your processes that have controlling terminals.

-a show all users’ processes which have controlling terminals
-e show all users’ processes
-x When displaying processes matched by other options, include processes which do not have a controlling terminal.
-f show detailed information
u show user information for processes

“top” command

display and update sorted information about processes

-o key   Order the process display by sorting on key in descending order. A + or - can be prefixed to the key name to specify ascending or descending order, respectively.

-O skey Use skey as a secondary key when ordering the process display

-s delay-secs Set the delay between updates to delay-secs seconds. The default delay between updates is 1 second

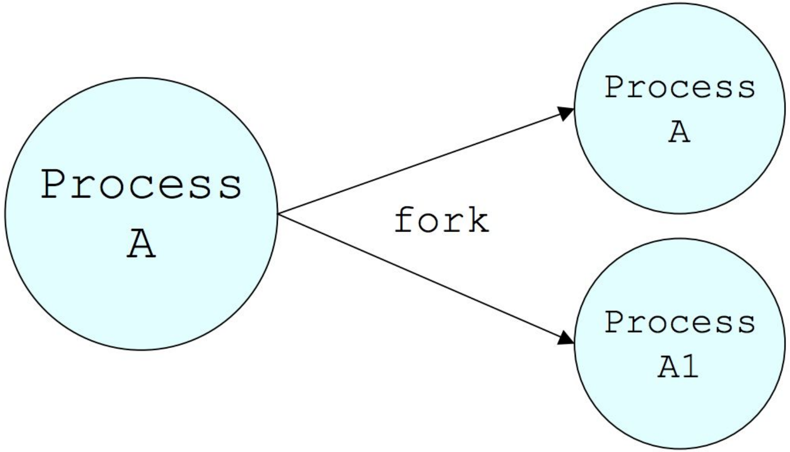
-n nprocs Only display up to nprocs processes

Fork

The fork system call creates a *duplicate* of the currently running program.

The duplicate (*child process*) and the original (*parent process*) both proceed from the point of the fork with exactly the same data.

The only difference is the return value from the fork call.



PID & PPID

System call: `int fork(void)`

- If `fork()` succeeds it returns the child PID to the parent and returns 0 to the child;
- If `fork()` fails, it returns -1 to the parent (no child is created) and sets `errno`

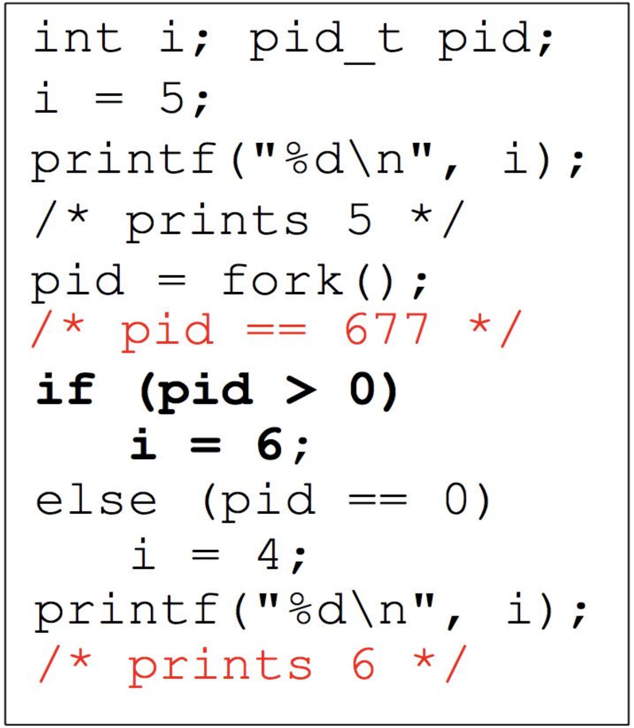
#include <unistd.h>

Related system calls:

- `int getpid()` – returns the PID of current process
- `int getppid()` – returns the PID of parent process (ppid of 1 is 1)

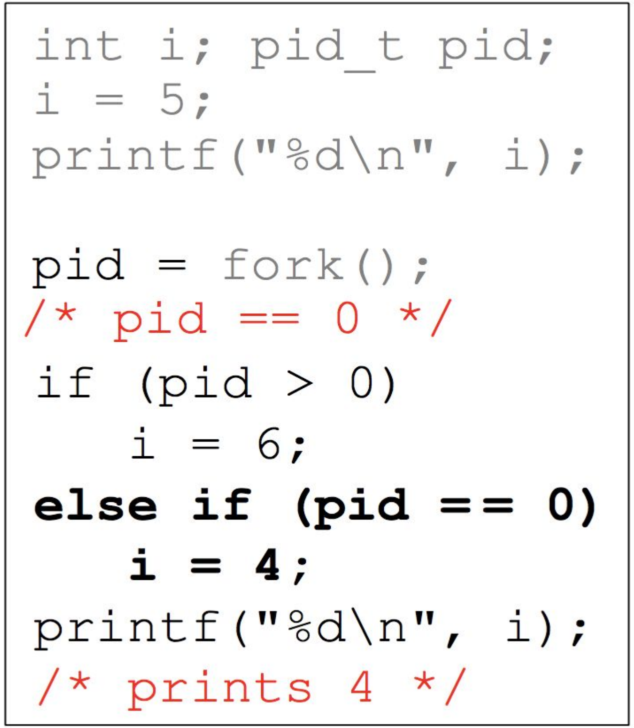
Different return values

Original process (parent)



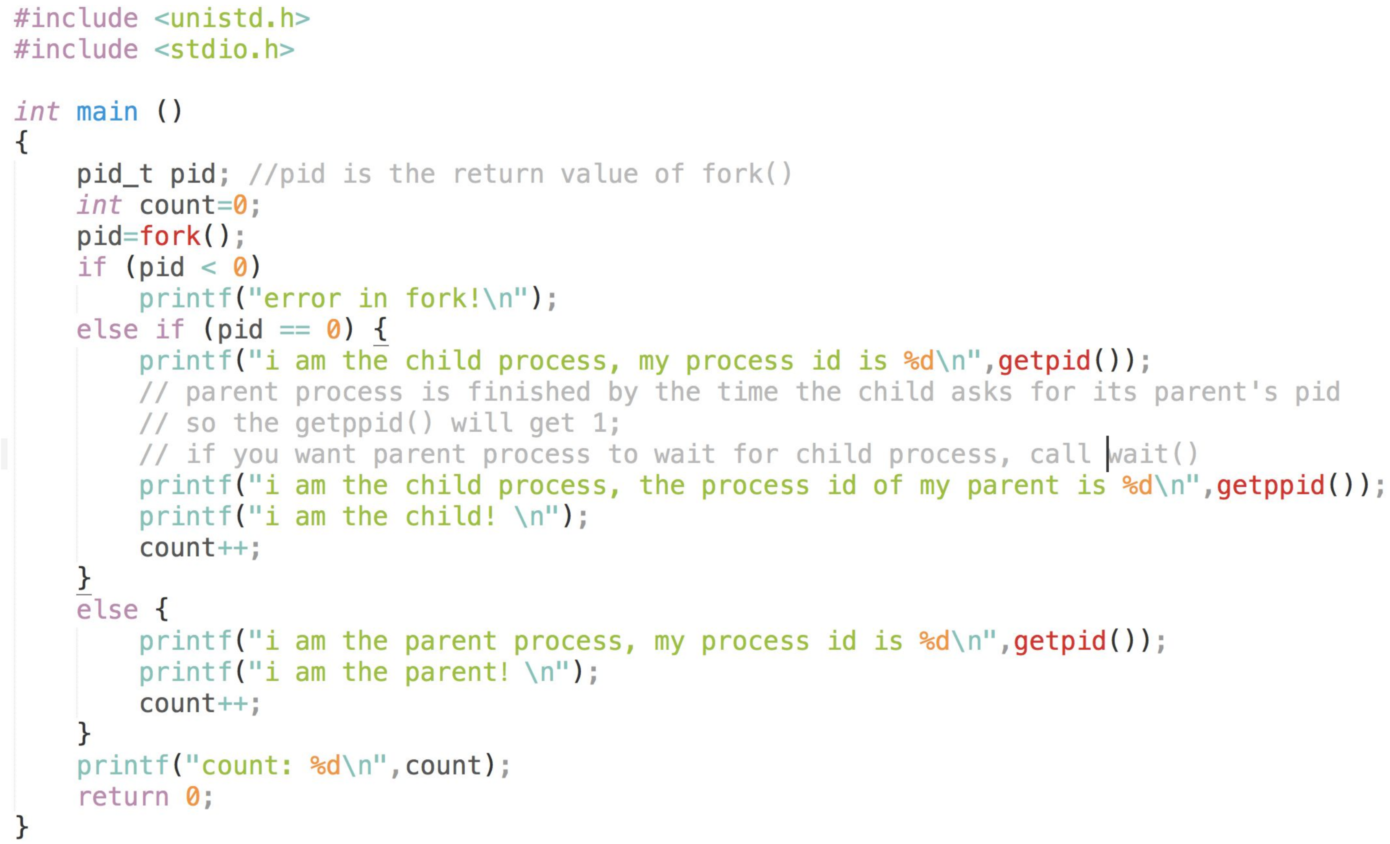
```
int i; pid_t pid;
i = 5;
printf("%d\n", i);
/* prints 5 */
pid = fork();
/* pid == 677 */
if (pid > 0)
    i = 6;
else (pid == 0)
    i = 4;
printf("%d\n", i);
/* prints 6 */
```

Child process



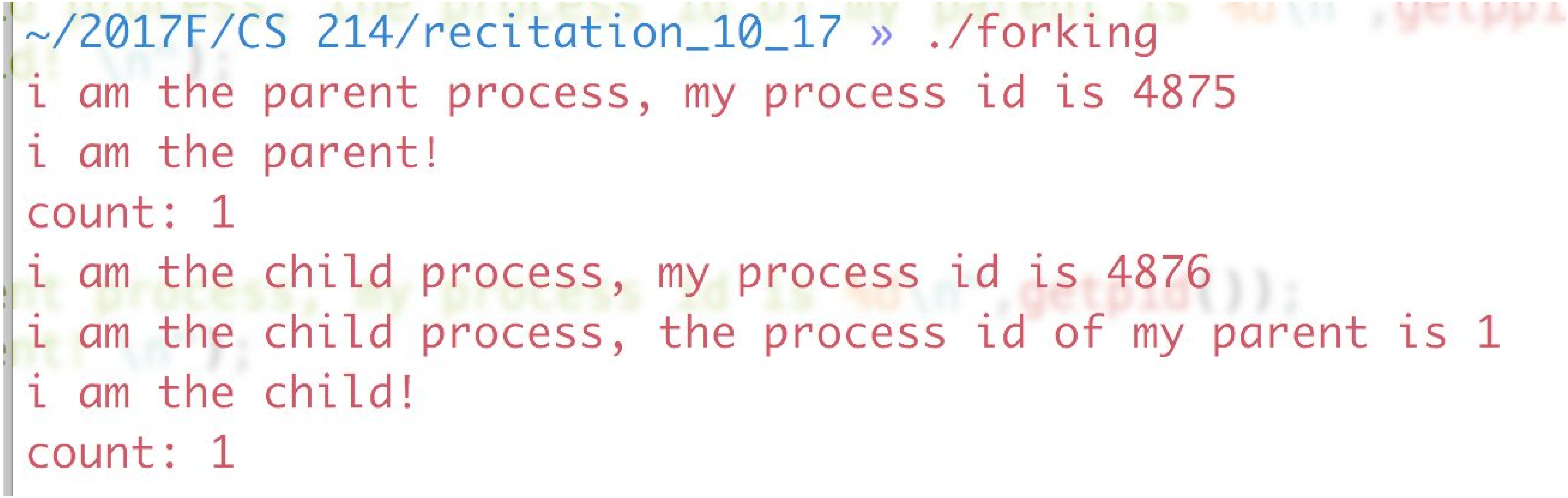
```
int i; pid_t pid;
i = 5;
printf("%d\n", i);

pid = fork();
/* pid == 0 */
if (pid > 0)
    i = 6;
else if (pid == 0)
    i = 4;
printf("%d\n", i);
/* prints 4 */
```

Fork example



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

int main ()
{
    pid_t pid; //pid is the return value of fork()
    int count=0;
    pid=fork();
    if (pid < 0)
        printf("error in fork!\n");
    else if (pid == 0) {
        printf("i am the child process, my process id is %d\n",getpid());
        // parent process is finished by the time the child asks for its parent's pid
        // so the getppid() will get 1;
        // if you want parent process to wait for child process, call wait()
        printf("i am the child process, the process id of my parent is %d\n",getppid());
        printf("i am the child! \n");
        count++;
    }
    else {
        printf("i am the parent process, my process id is %d\n",getpid());
        printf("i am the parent! \n");
        count++;
    }
    printf("count: %d\n",count);
    return 0;
}
```

Fork example - running output



```
~/2017F/CS 214/recitation_10_17 » ./forking
i am the parent process, my process id is 4875
i am the parent!
count: 1
i am the child process, my process id is 4876
i am the child process, the process id of my parent is 1
i am the child!
count: 1
```

HW4.0 - Implementing "ls"

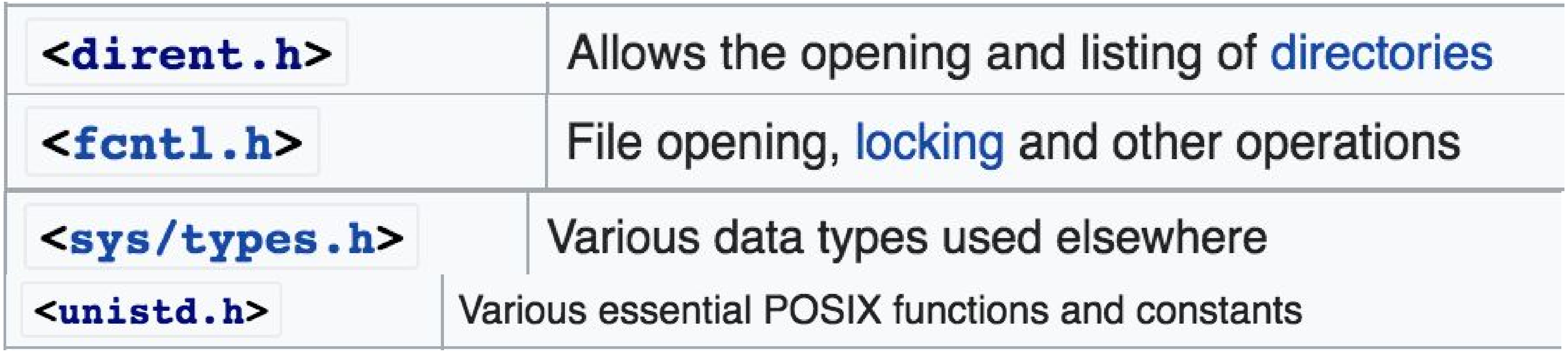
0. Using opendir and readdir, open the current directory and output all filenames until there are no more

```
char * base = "./";
DIR * thingy = opendir(base);
dirent * newfile = readdir(thingy);
```

HW4.0 - C POSIX library

The C POSIX library is a specification of a C standard library for *POSIX* (abbr. for Portable Operating System Interface) systems. It includes many header files.

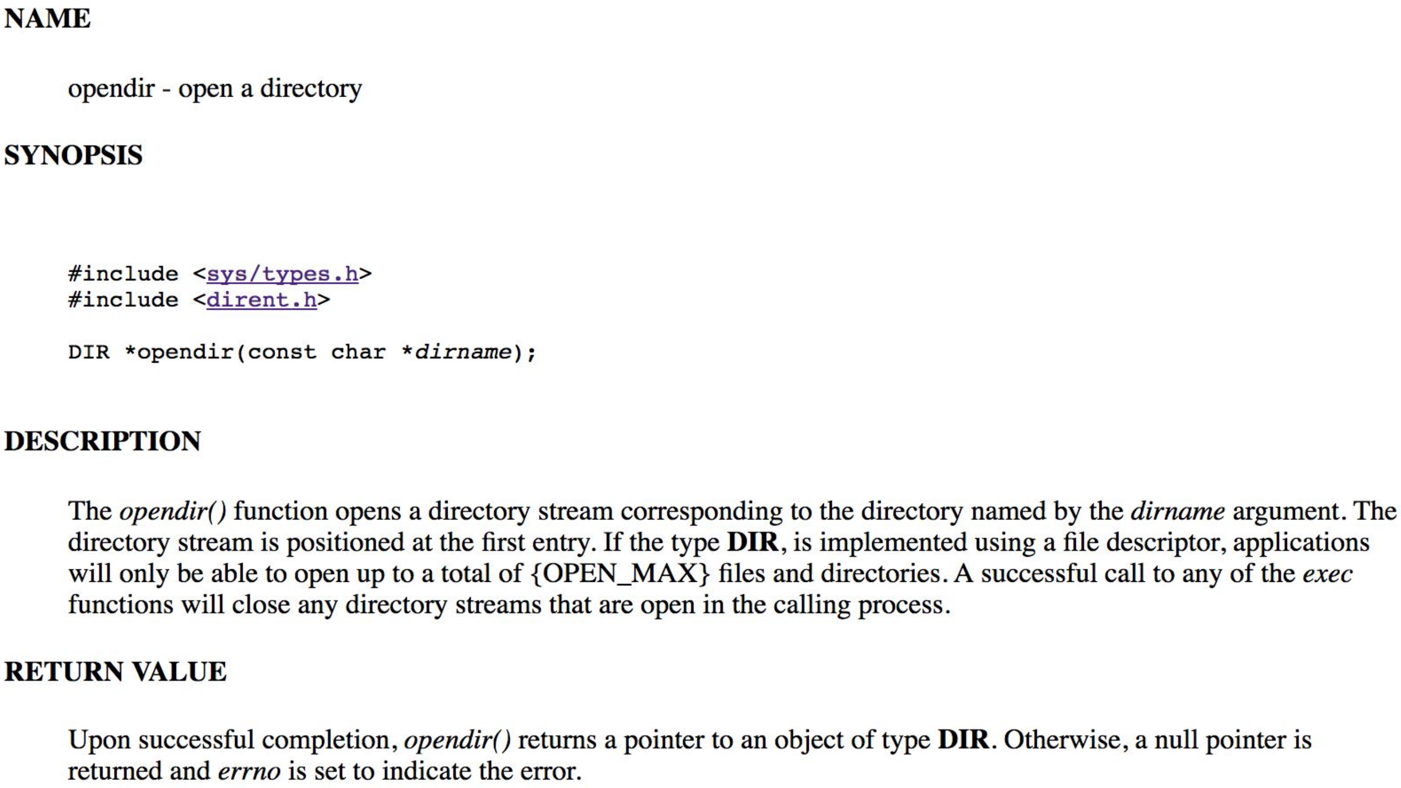
For reference: https://en.wikipedia.org/wiki/C_POSIX_library



`<dirent.h>`	Allows the opening and listing of directories
`<fcntl.h>`	File opening, locking and other operations
`<sys/types.h>`	Various data types used elsewhere
`<unistd.h>`	Various essential POSIX functions and constants

HW4.0 - Header file <dirent.h>

opendir



NAME

opendir - open a directory

SYNOPSIS

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

DIR *opendir(const char *dirname);
```

DESCRIPTION

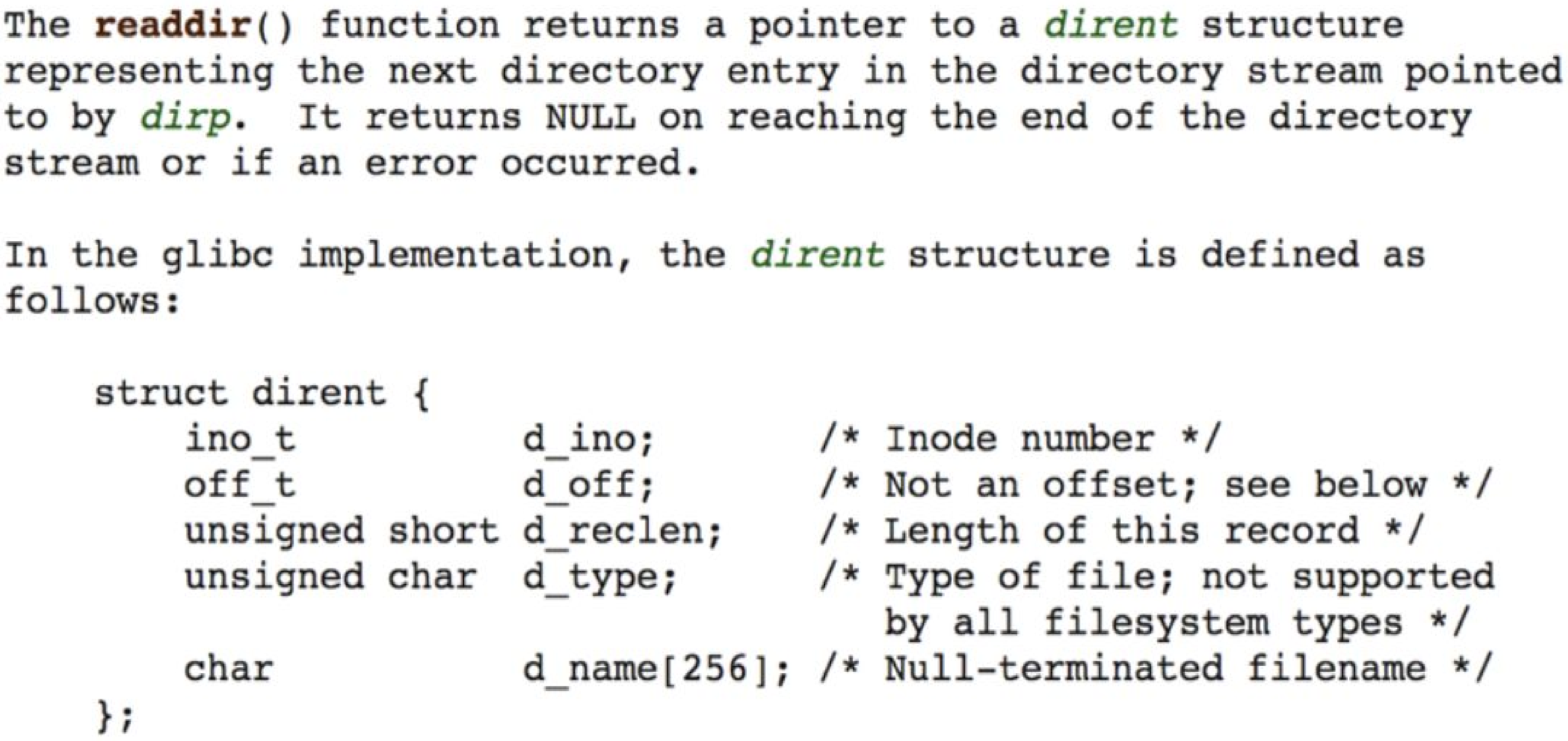
The *opendir()* function opens a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR**, is implemented using a file descriptor, applications will only be able to open up to a total of {OPEN_MAX} files and directories. A successful call to any of the *exec* functions will close any directory streams that are open in the calling process.

RETURN VALUE

Upon successful completion, *opendir()* returns a pointer to an object of type **DIR**. Otherwise, a null pointer is returned and *errno* is set to indicate the error.

HW4.0 - Header file <dirent.h>

readdir



The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

HW4.0 - Answer

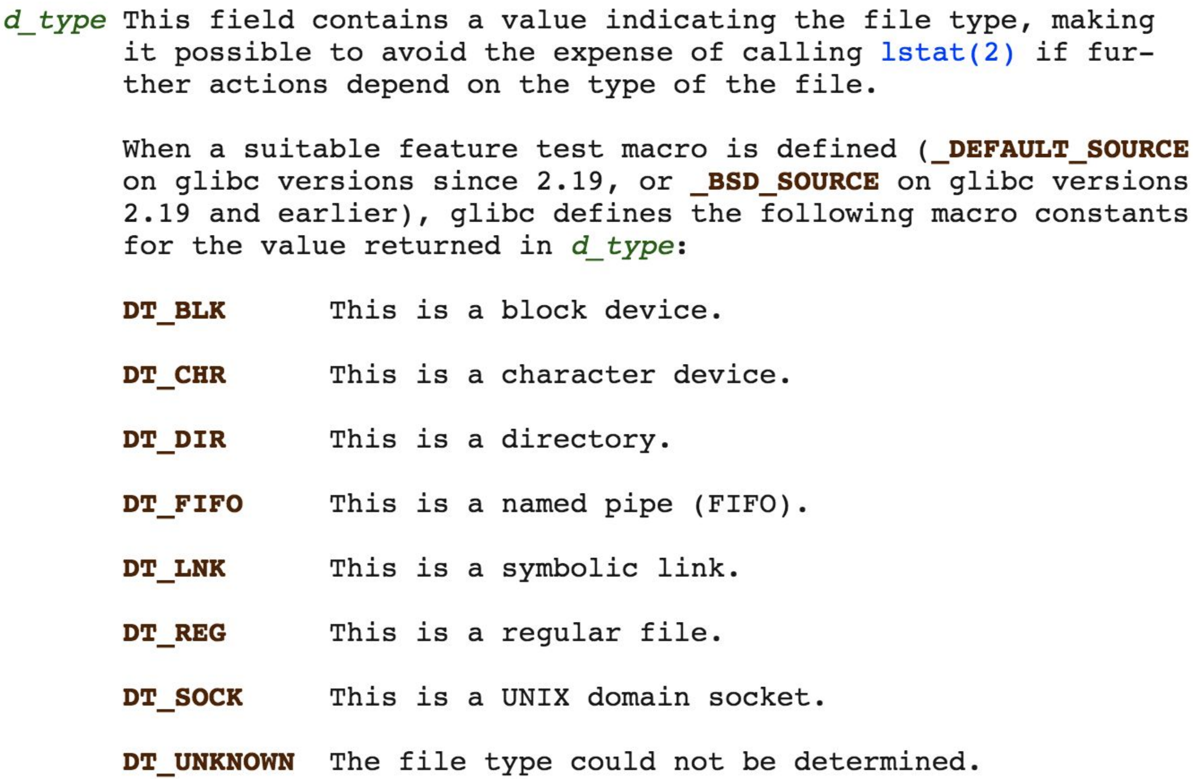
```
#include <sys/types.h>
#include <dirent.h>

char * base = "./";

DIR *thingy = opendir(base);
struct dirent *newfile;

char buf[1024] = {0};
while((newfile = readdir(thingy)) != NULL) {

}
```

HW4.1 - Implementing "ls"

1. Parse the dirent struct to see if an entry is a directory or a file. If it is a file, prepend "./" to the filename, if it is a directory, don't.

```
  .. if newfile != NULL
  //check type field of newfile dirent struct to determine the type of this file endpoint
newfile->d_type
// compare with system defines for different endpoint types (3rd paragraph under 'NOTES' in man 3 readdir).
  ... if == DT_REG //regular file
  elseif == DT_DIR //directory
....
```

HW4.1 - Values of d_type



d_type This field contains a value indicating the file type, making it possible to avoid the expense of calling lstat(2) if further actions depend on the type of the file.

When a suitable feature test macro is defined (**_DEFAULT_SOURCE** on glibc versions since 2.19, or **_BSD_SOURCE** on glibc versions 2.19 and earlier), glibc defines the following macro constants for the value returned in *d_type*:

DT_BLK This is a block device.

DT_CHR This is a character device.

DT_DIR This is a directory.

DT_FIFO This is a named pipe (FIFO).

DT_LNK This is a symbolic link.

DT_REG This is a regular file.

DT_SOCK This is a UNIX domain socket.

DT_UNKNOWN The file type could not be determined.

HW4.1 - Answer

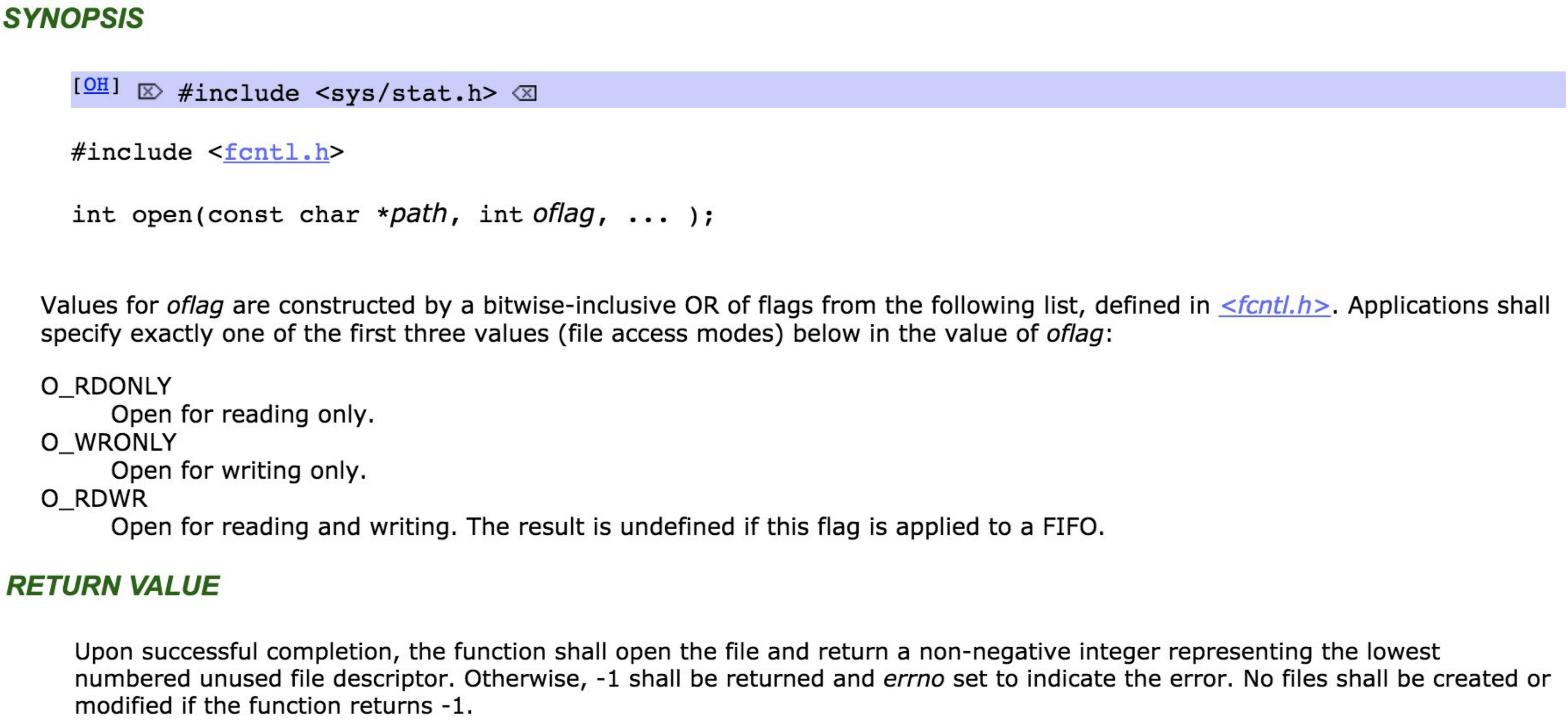
```
while((newfile = readdir(thingy)) != NULL) {
if (newfile->d_type == DT_REG) {

}

else if (newfile->d_type == DT_DIR) {

}

else return 0;
```

HW4.2 - Implementing "ls"

2. Open a file handle to each file and use lseek to determine the file's size in bytes, and print out the file's size next to its name.

```
//assemble name of file using base directory and current path/name
// concatenate all path up until now...
// add current name if it is a file...

int checkFD = open(newerpath, RD_ONLY);
 ... if no error...
int len = lseek(checkFD, 0, SEEK_END);
close(checkED);

printf( filename with full path, either color to indicate file/dir or put a "/" at the end to indicate
dir, and number of bytes of size, if a file )

//be sure to closedir() when done with dir descriptor
```

HW4.2 - open function in <fcntl.h>



SYNOPSIS

```
[OH] ⌦ #include <sys/stat.h> ⌫

#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

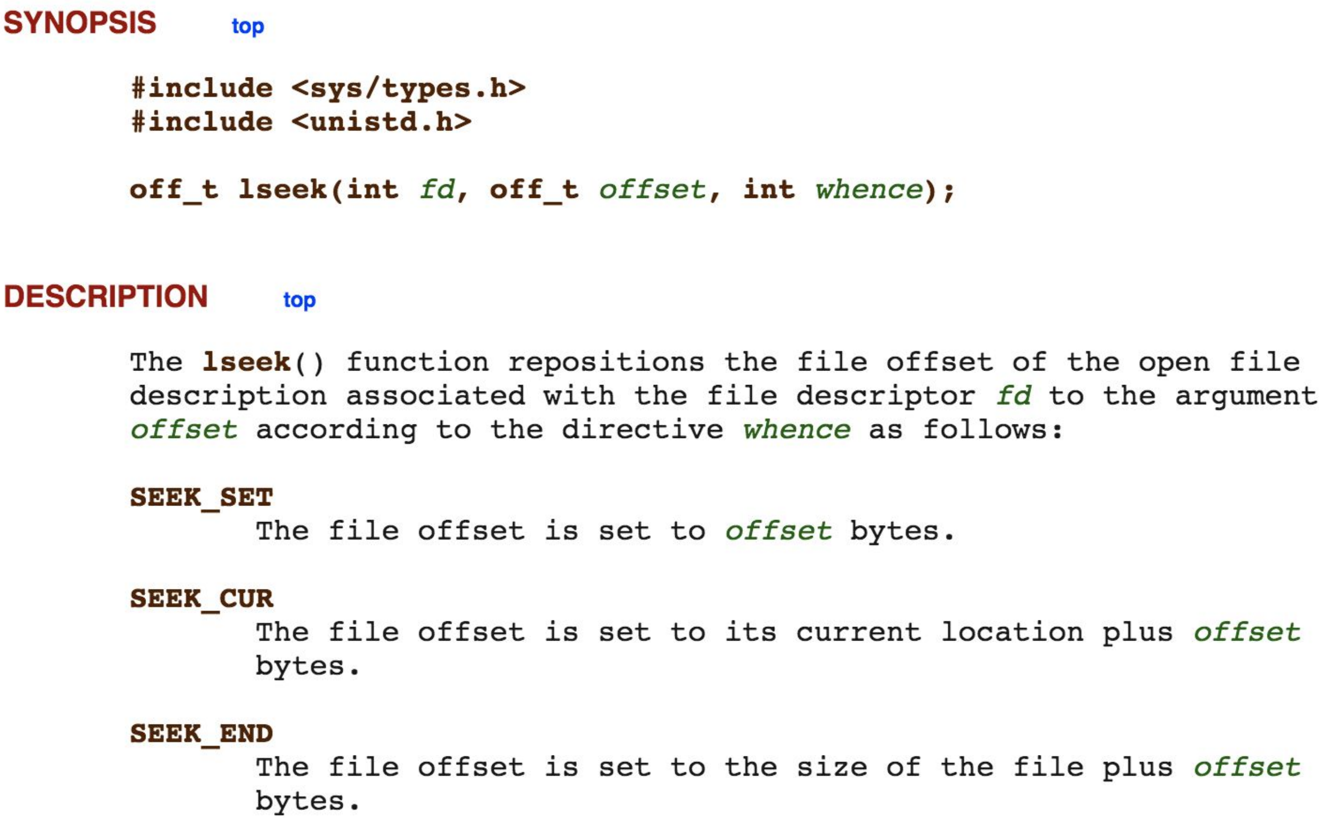
Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in *<fcntl.h>*. Applications shall specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY
    Open for reading only.
O_WRONLY
    Open for writing only.
O_RDWR
    Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

RETURN VALUE

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 shall be returned and *errno* set to indicate the error. No files shall be created or modified if the function returns -1.

HW4.2 - lseek function in <unistd.h>



SYNOPSIS top

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

off_t lseek(int fd, off_t offset, int whence);
```

DESCRIPTION top

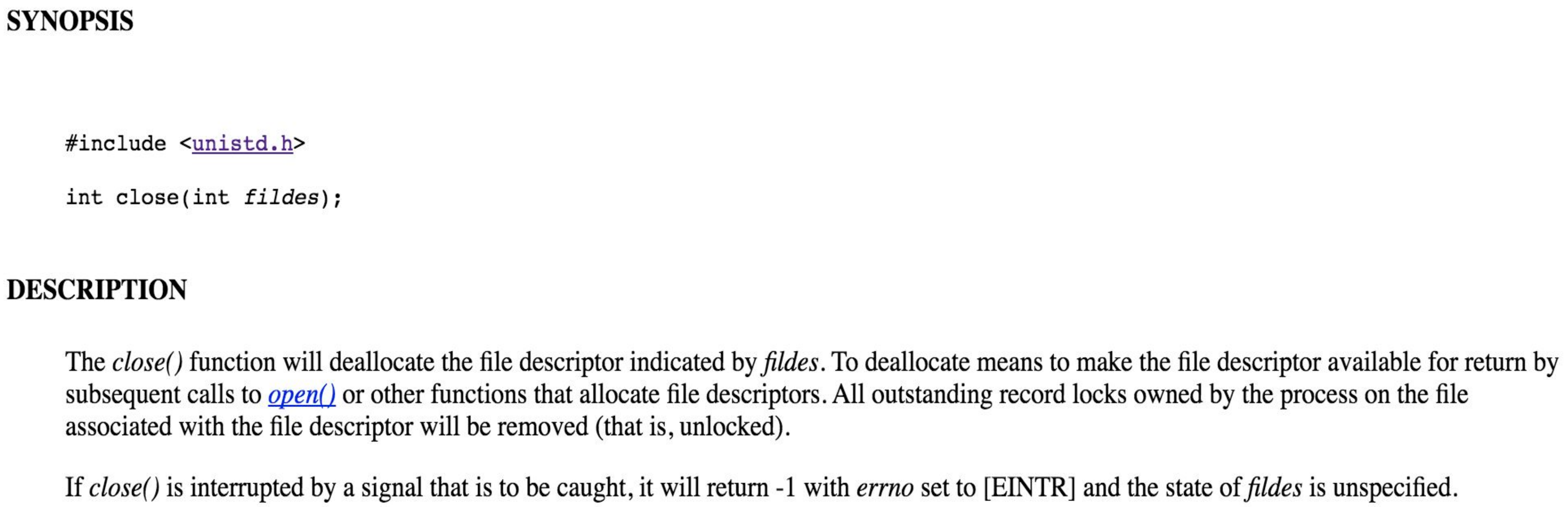
The **lseek**() function repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:

SEEK_SET
The file offset is set to *offset* bytes.

SEEK_CUR
The file offset is set to its current location plus *offset* bytes.

SEEK_END
The file offset is set to the size of the file plus *offset* bytes.

HW4.2 - close function in <unistd.h>



SYNOPSIS

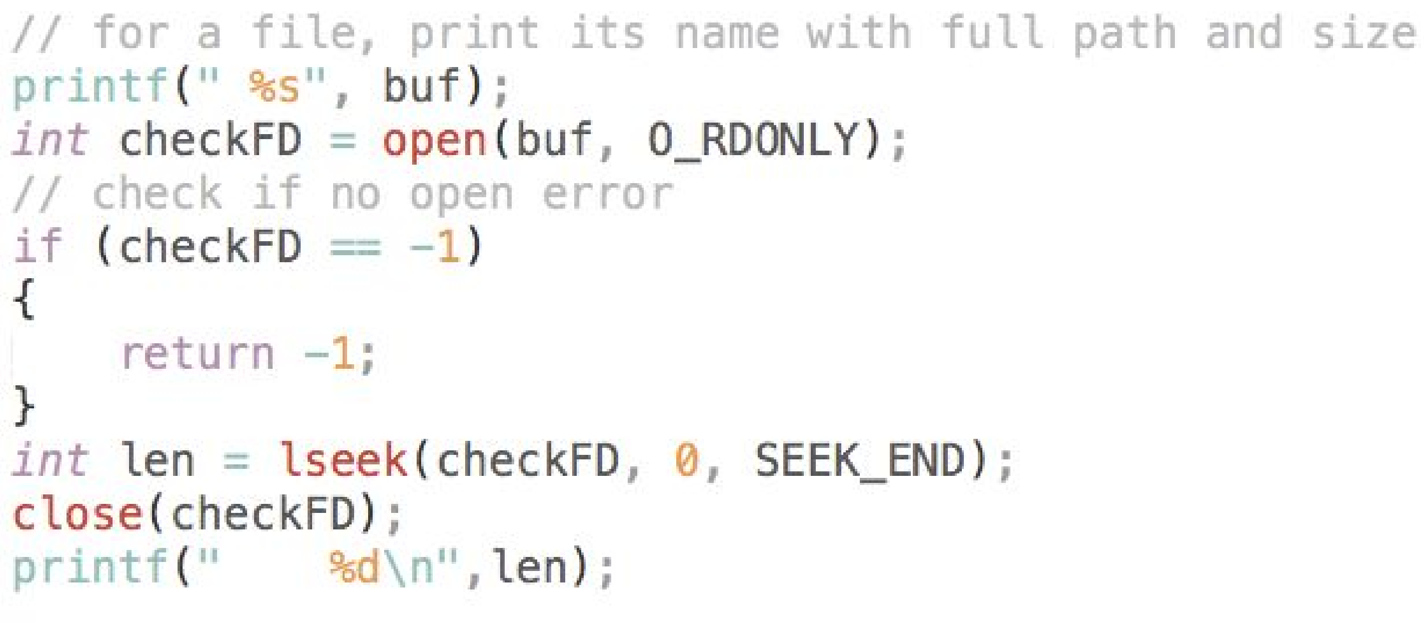
```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

The *close()* function will deallocate the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to *open()* or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor will be removed (that is, unlocked).

If *close()* is interrupted by a signal that is to be caught, it will return -1 with *errno* set to [EINTR] and the state of *fildes* is unspecified.

HW4.2 - Answer



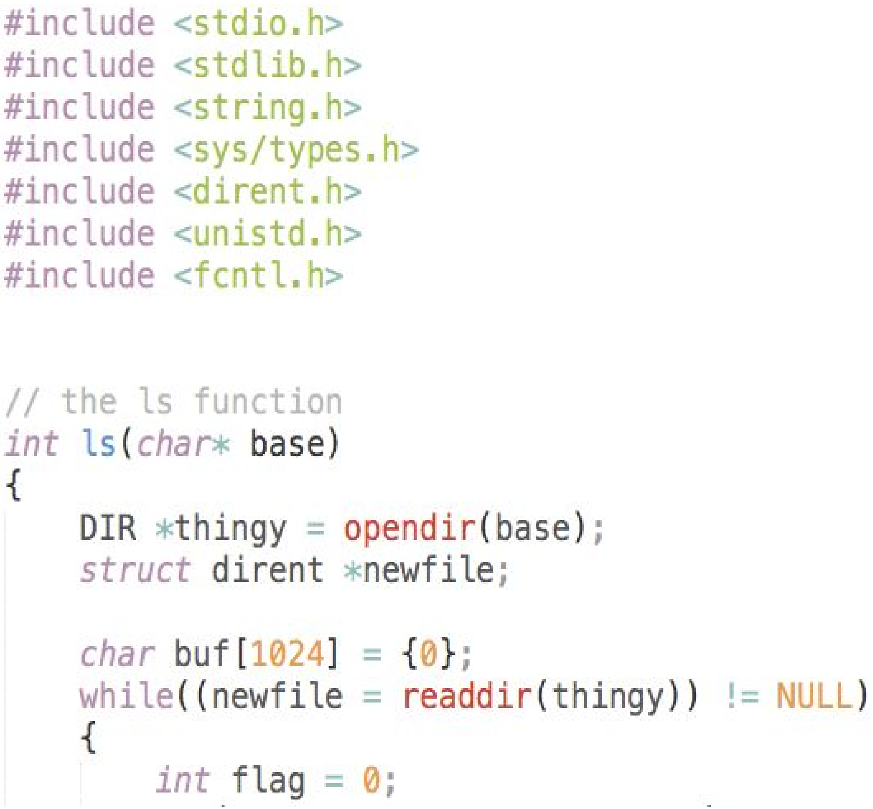
```
// for a file, print its name with full path and size
printf(" %s", buf);
int checkFD = open(buf, O_RDONLY);
// check if no open error
if (checkFD == -1)
{
    return -1;
}
int len = lseek(checkFD, 0, SEEK_END);
close(checkFD);
printf("    %d\n",len);
```

HW4.3 - Implementing "ls"

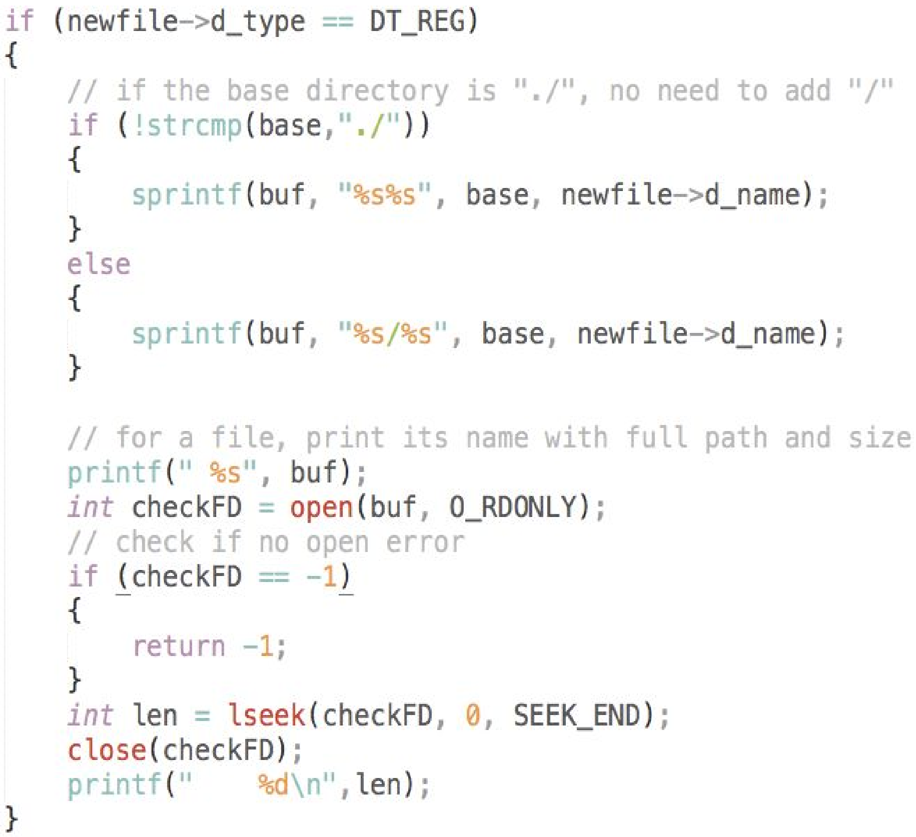
3. Add a recursive element. If you find a directory, recursively call your code on that directory and prepend that directory name to each filename and directory name outputted.

```
elseif newfile->d_type == DT_DIR
strcat(newpath, base)
// add currend name if it is a file...
newerpath = realloc(newpath,
strlen(newpath)+strlen(newfile->d_name));
... recursively call my_LS on newerpath
```

HW4.3 - A Sample Code

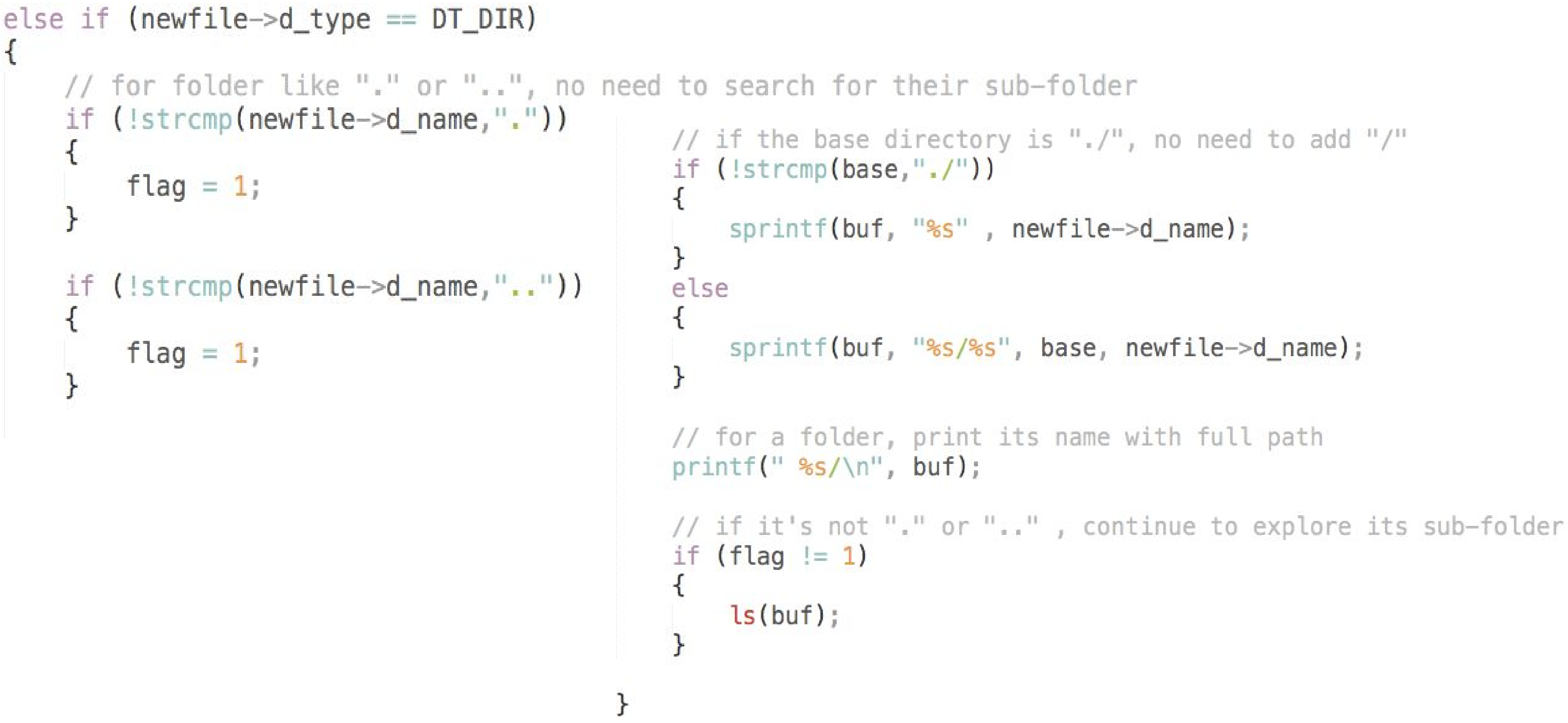


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
#include <fcntl.h>


// the ls function
int ls(char* base)
{
    DIR *thingy = opendir(base);
    struct dirent *newfile;

    char buf[1024] = {0};
    while((newfile = readdir(thingy)) != NULL)
    {
        int flag = 0;
```



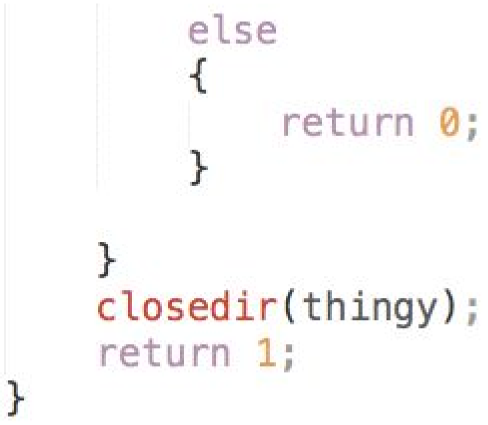
```c
if (newfile->d_type == DT_REG)
{
    // if the base directory is "./", no need to add "/"
    if (!strcmp(base,"./"))
    {
        sprintf(buf, "%s%s", base, newfile->d_name);
    }
    else
    {
        sprintf(buf, "%s/%s", base, newfile->d_name);
    }

    // for a file, print its name with full path and size
    printf(" %s", buf);
    int checkFD = open(buf, O_RDONLY);
    // check if no open error
    if (checkFD == -1)
    {
        return -1;
    }
    int len = lseek(checkFD, 0, SEEK_END);
    close(checkFD);
    printf("    %d\n",len);
}
```

HW4.3 - A Sample Code (Cont.)

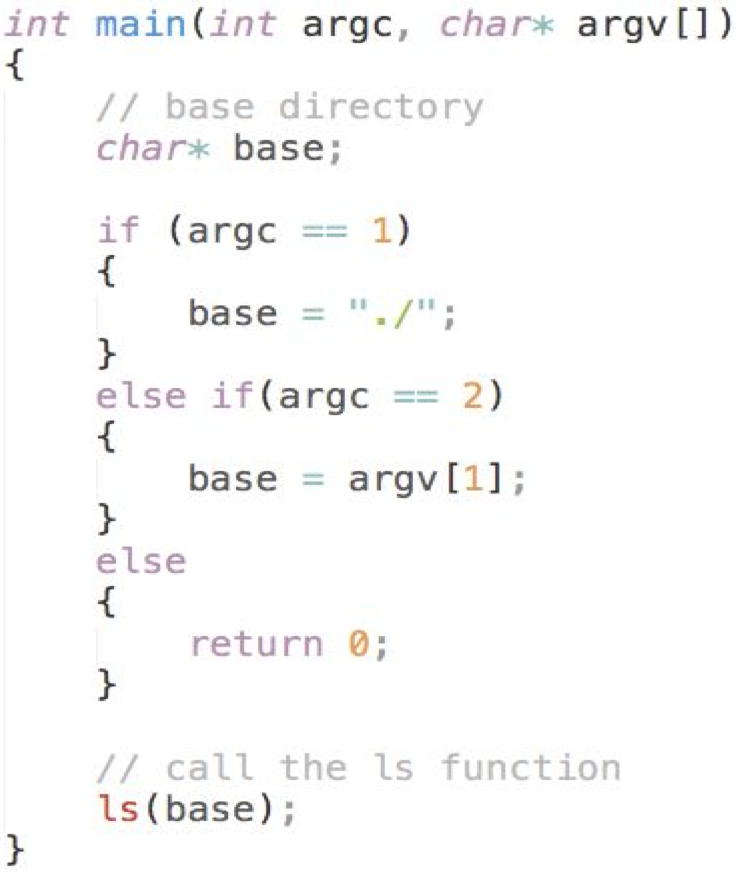


```c
else if (newfile->d_type == DT_DIR)
{
    // for folder like "." or "..", no need to search for their sub-folder
    if (!strcmp(newfile->d_name,"."))
    {
        flag = 1;
    }

    if (!strcmp(newfile->d_name,".."))
    {
        flag = 1;
    }

    // if the base directory is "./", no need to add "/"
    if (!strcmp(base,"./"))
    {
        sprintf(buf, "%s" , newfile->d_name);
    }
    else
    {
        sprintf(buf, "%s/%s", base, newfile->d_name);
    }

    // for a folder, print its name with full path
    printf(" %s/\n", buf);

    // if it's not "." or ".." , continue to explore its sub-folder
    if (flag != 1)
    {
        ls(buf);
    }

}
```

HW4.3 - A Sample Code (Cont.)



```
        else
        {
            return 0;
        }

    }
    closedir(thingy);
    return 1;
}
```



```
int main(int argc, char* argv[])
{
    // base directory
    char* base;

    if (argc == 1)
    {
        base = "./";
    }
    else if(argc == 2)
    {
        base = argv[1];
    }
    else
    {
        return 0;
    }

    // call the ls function
    ls(base);
}
```

HW5.0 - Implementing "ps"

0. Modify the *ls* we wrote this week except *output /proc*
Then open the file 'status', look for the uid section and extract the owner's *uid*.
Using *pwd.h*, determine the name of the user who owns the process and print it out as well.

HW5.0 - Implementing "ps"

Coding reference:

```
char * base = "/proc"
DIR * stuff = opendir(base, RD_ONLY)
dirent * pidnumber = NULL;

char * newCmdline = NULL;
char * workingName = NULL;

int fd = -1;
```

```
while
{
pidnumber = readdir(stuff);
if ( pidnumber != NULL && pidnumber->d_type ==
DT_DIR) {newCmdline = ... malloc
strlen(base)+strlen(pidnumber->d_name)+9;
newCmdline[0] = '\0';
strcat(newCmdline, base);
strcat(newCmdline, "/");
strcat(newCmdline, pidnumber->d_name);
strcat(newCmdline, "/cmdline"

fd = open(newCmdline, RD_ONLY);

... read loop ...
 .. .while read from fd != 0, printf it out ...

close(fd);
}
}
```

```
do(pidnumber != NULL);
```

... this gets you all command lines run for all pids for all procs on the system

printf(command run and its pid(i.e. directory name of cmdline))

HW5.1 - Implementing "ps"

1. Open status alongside/after cmdline (but before clocking your readdir loop! readdir is destructive!) read through and parse the status file looking for 'uid'

```
while reading in status file ...
if buffer[i] == 'u'
if bufferLength - i >= 2
 if (buffer[i+1] == 'i' && buffer[i+2] == 'd')
 ... can start reading in uid that called this code ... w00t!
```

printf(command run, its pid and the uid that called it) .. boring .. want userNAME, not UID .. :^P .. but only place in system this information is together is in passwd file ... very well, then...

struct passwd * getUname = getpwuid(UID parsed out of status above)

new can print out:

 command run (from cmdline file)
 username that called it (passwd->pw_name)
.. for every current PID

Congrats! ... you wrote basic “ps”

HW5.2 - Implementing "ps"

2. Then open the file *schedstat* and read in order: time spent running on CPU (in nanoseconds), time spent waiting on a run queue, # of times context switched (be careful of decimals! you may want to check the status file to be sure your degree is correct)

Then, for some fun times, print out some stats. Maybe also add command line options! Like default “ps” only prints out info for YOUR procs by default - so can get current uid within your code using getuid and comb through /proc and only print out status and schedstat information for staff whose uid matches