Unix: File Access Primitives

Ref: Chapters 1 and 2 of [HGS].

Some facilities in Unix:

- Tree structured file system.
  - Root directory.
  - Absolute and relative path names.
  - File protection bits.
- Shell (command interpreter).
- Creation of multiple processes.
- Input/output redirection.
  - Any of `stdin`, `stdout` `stderr` can be redirected.
  - Syntax depends on the shell used. (Examples to be presented in class.)
- Pipes: Allow the output of one process to become the input to another.

Examples:

```
%  ls -l | wc -l
%  who | grep  smith | wc -l
```

Kernel:

- Constantly resides in memory.
- Controls and monitors processes and file accesses.
- User processes request kernel services through **system calls**.

File Access Primitives:

- So far: File access using `<stdio.h>`.
- For programming at the system level, system calls are needed.
- System calls provide primitives for file access.
- The `<stdio.h>` library is built on top of system calls for file access.

File Descriptors:

- Different from file pointers (variables of type `FILE *`) used in `<stdio.h>`.
- File descriptors are of type `int`.
- Kernel refers to all open files using file descriptors.
- File descriptors 0, 1 and 2 correspond to `stdin`, `stdout` and `stderr` respectively.
- Better to use their symbolic names in programs:

```
STDIN_FILENO
STDOUT_FILENO
STDERR_FILENO
```

- System calls to open files return file descriptors which must be passed to other system calls.

- Header files generally needed:

```
<unistd.h>
<sys/types.h>
<sys/stat.h>
<fcntl.h>
```

System call `open`:

First form:

```
int  open (const char *name,
           int oflag);
```

- First form used for opening an existing file.
- Returns file descriptor if successful; otherwise, returns -1.
- `name`: Name of file to be opened.
- `oflag`: File access method.

Examples for access method: (These are symbolic constants defined in <fcntl.h>.)

```
O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT
O_TRUNC
```

Note: The oflag parameter for open is usually the bitwise-or of some of the above constants.

Sample program segment:

```
int  fd;
   .
   .
fd = open ("/usr/smith/file.c",
           O_WRONLY | O_TRUNC);
if (fd == -1) {
   fprintf(stderr, "Open failed.\n");
   exit(1);
}
```


Second form:

```
int  open (const char  *name,
           int  oflag,
           mode_t  mode);
```

- Second form used for creating a new file.
- Returns file descriptor if successful; otherwise, returns -1.
- Parameters name and oflag as before.
- mode: Specifies permissions for the new file. (More on this later.)

Sample program segment:

```
#define  MODE  0644
int  fd;
   .
   .
fd = open ("/usr/smith/file.c",
           O_RDWR | O_CREAT, MODE);
```

```
if (fd == -1) {
   fprintf(stderr, "Open failed.\n");
   exit(1);
}
```

System call `creat`**:**

```
int  creat (const char *name,
            mode_t  mode);
```

- Can be used for creating a new file. (Using `open` is generally preferred.)
- Returns file descriptor if successful; otherwise, returns -1.
- `name` and `mode`: As in `open` system call.

Note: The call

```
fd = creat( "file", 0644);
```

is equivalent to:

```
fd = open( "file",
        O_WRONLY | O_CREAT | O_TRUNC,
        0644);
```


System call `close`**:**

```
int  close (int  filedes);
```

- Used to close the file referred to by the descriptor `filedes`.
- Returns 0 if successful; otherwise, returns -1.
- Although all open files are automatically closed when a program exits, it is a good idea to close the files explicitly.

Sample program segment:

```
int  fd;
   .
   .
/* Call to open etc. */
   .
   .
if (close(fd) == -1) {
   fprintf(stderr, "Close failed.\n");
   exit(1);
}
```

System call `read`:

```
ssize_t  read (int  fd,
               void  *buf,
               size_t  n);
```

- Reads `n` bytes from the file given by the descriptor `fd` into memory starting from the location given by `buf`.
- The file given by the descriptor `fd` must be open for reading.
- Normally, returns the number of bytes read. Returns 0 if EOF occurs before any bytes are read. Returns -1 if an error occurs.

Sample program segment:

```
#define  SIZE  25
int  fd;  int temp;  char  buf[SIZE];
   .
   .  /* Call to open etc. */
   .
temp = read(fd, buf, (size_t) SIZE);
```



```
if (temp == -1) {
   fprintf(stderr, "Error in read.\n");
   exit(1);
}

if (temp == 0) {
   .
   .   /* Code for handling EOF. */
   .
}
```

System call `write`:

```
ssize_t  write (int  fd,
                const void  *buf,
                size_t  n);
```

- Writes the contents of `n` bytes starting from the location given by `buf` into the file given by the descriptor `fd`.
- The file given by the descriptor `fd` must be open for writing (or appending).

- Normally, returns the number of bytes written. Returns -1 if an error occurs.

Sample program segment:

```
#define  SIZE  25
int  fd;  int temp;  char  buf[SIZE];
   .
   .  /* Call to open etc. */
   .

temp = write(fd, buf, (size_t) SIZE);

if (temp == -1) {
   fprintf(stderr, "Error in write.\n");
   exit(1);
}
```

Program example: Handout 11.1.

System call `lseek`:

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

- Similar to `fseek` of `stdio.h`, except that `lseek` uses a file descriptor while `fseek` uses a file pointer.
- The file given by the descriptor `fd` must be open for reading or writing.
- Parameter `offset` specifies the number of bytes for moving. (Note that `offset` may be negative.)
- `sflag` can be any of the following three constants.
  - `SEEK_SET` : `offset` specified relative to the beginning of the file.
  - `SEEK_CUR` : `offset` specified relative to the current position.
  - `SEEK_END` : `offset` specified relative to the end of the file.

- Normally, returns the new position in the file. Returns -1 if an error occurs.

Sample program segment:

```
int  fd;  off_t offset, new_pos;
   .
   .  /* Call to open etc. */
   .
new_pos = lseek(fd, offset, SEEK_END);

if (new_pos == -1) {
   fprintf(stderr, "Error in lseek.\n");
   exit(1);
}
```

Reading assignment: Program example on pages 24–25 of [HGS].

System calls `unlink` and `remove`:

```
int  unlink (const char *pathname);
int  remove (const char *pathname);
```

- Both `unlink` and `remove` eliminate the file specified by `pathname`.
- Originally, only `unlink` was in the list of system calls; ANSI C standard added `remove`.
- Return value: 0 if successful and -1 otherwise.

Reporting Errors:

- Purpose: To provide more information when a system call reports error.
- Header file `<errno.h>` and global `int` variable `errno`.
- System assigns a value to `errno` when an error occurs.
- Examples of `errno` values: `EACCES`, `EBADF`, `ENOENT`.

- Can determine the error given the value of `errno`. (See Appendix A of [HGS].)
- Library function `perror` available to print error message:

```
void  perror (const char *msg);
```

- Call to `perror` prints to `stdout` the message string along with an error message corresponding to the value of `errno`.

Sample code segment: Assume that the file `"/usr/nofile"` does not exist.

```
int  fd;
fd = open("/usr/nofile", O_RDONLY);
if (fd == -1) {
   perror("Error");
   exit(1);
}
```

Output:

```
Error: No such file or directory
```


Notes:

- The value of `errno` is not reset when a system call is successful.
- Value of `errno` must be used only when a system call returns a value indicating error.