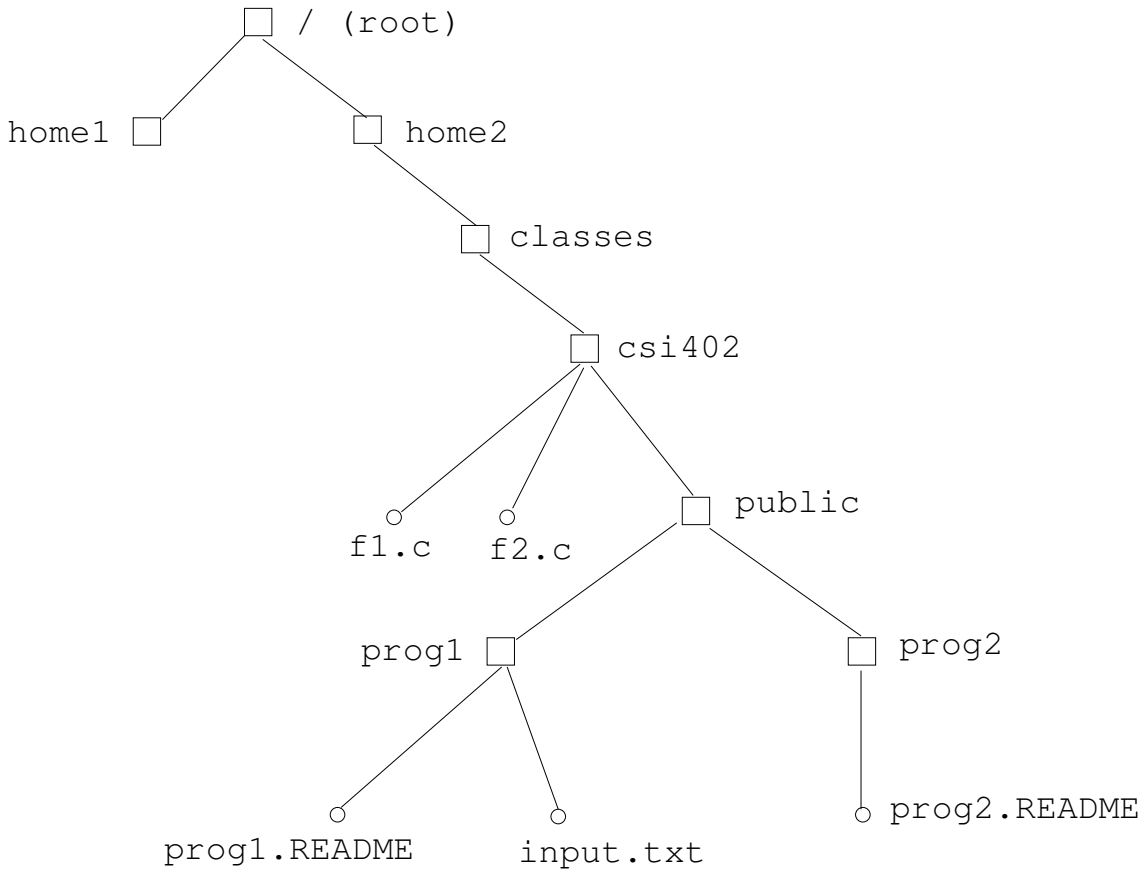
# CSI 402 – Lecture 10

# (Unix – File Access Primitives)

# Some Facilities Provided by Unix

**Ref:** Chapters 1 and 2 of [Haviland et al.].

**Tree-structured File System:**




**Note:** Users interact with the system using the **shell** (command interpreter).

# Home Directory and Absolute/Relative Path Names

**Home directory of user** `csi402`:

```
/home2/classes/csi402
```

**Example – Absolute Path Name:**

```
/home2/classes/csi402/public/prog1/prog1.README
```

**Example – Relative Path Name:**

- Suppose the current directory is `/home2/classes/csi402/public`.
- Then the name

  ```
  prog1/input.txt
  ```

  refers to the file `input.txt` in the subdirectory `prog1`.

# Protection Bits for Files

- **Three classes**: User (or Owner), Group and Others.
- Three bits, denoted by `rwx` (for `read`, `write` and `execute`) for each class. Permission is ON (OFF) if the bit is 1 (0).
- Permission bits are usually specified in octal.

**Example:** Suppose the permission bits are specified as 754 (octal).

- The user (owner) can read, write and execute the file.
- People in the same group can read and execute the file but *cannot* write to (i.e., modify) the file.
- Others can read the file but *cannot* write to it or execute it.

**Specifying protection bits – Command** chmod:

```
% chmod   754  input.txt
```

# Input/Output Redirection

- Any of the devices `stdin`, `stdout` `stderr` can be redirected.
- Redirection is transparent to a program.
- Syntax depends on the shell used.

**Redirection Commands for** bash:

- To redirect `stdin` to `infile`:

```
% prog  <  infile
```

- To redirect `stdout` to `outfile`:

```
% prog  >  outfile
```

- To redirect `stderr` to `errfile`:

```
% prog  2>  errfile
```

# Input/Output Redirection (continued)

**Redirection Commands for `bash` (continued):**

- To redirect `stdout` and `stderr` to the same file (`out_err_file`):

```
% prog  &>  out_err_file
```

- To redirect `stdout` to `outfile` and `stderr` to `errfile`:

```
% (prog  >  outfile) &> errfile
```

**Pipes:** Allow the output of one process to become the input to another.

**Examples:**

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

# The Kernel and System Calls

**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 Access Primitives (continued)

**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.
- Use the symbolic names of these devices (`STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`) in programs.
- System calls to open files return file descriptors which must be passed to other system calls.

# System Calls for File Access (continued)

**File Descriptors (continued):**

- Header files needed:

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

**System call** open: (**First form**)

- **Prototype:**

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

- This 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.

# System Calls (continued)

**System Call `open` (continued):**

**Values for access method:**

```
O_RDONLY    O_WRONLY    O_RDWR

O_APPEND    O_CREAT     O_TRUNC
```

- The above are symbolic constants defined in `<fcntl.h>`.
- The `oflag` parameter for `open` is usually the bitwise-or of some of the above constants.

# System Calls (continued)

**Sample code segment for `open`: (first form)**

```
int  fd;
   .
   .

fd = open ("/usr/smith/file.c", O_WRONLY | O_TRUNC);

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

# System Calls (continued)

**System call** open: (**Second form**)

- **Prototype:**

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

- This form is 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.

# System Call open (continued)

**Sample program segment for `open`: (second form)**

```
#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 Calls (continued)

**System call** creat:

- **Prototype:**

```
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 Calls (continued)

**System call** close:

- **Prototype:**

```
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.

# System Call `close` (continued)

**Sample code segment for `close`:**

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

# System Calls (continued)

**System call** read:

- **Prototype:**

```
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.

# System Calls (continued)

**Sample code segment for `read`:**

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

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

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

# System Calls (continued)

**System call** write:

- **Prototype:**

```
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.

# System Calls (continued)

**Sample code segment for `write`:**

```
#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 10.1.

# System Calls (continued)

**System call** lseek:

- **Prototype:**

```
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` is relative to the beginning of the file.
  - `SEEK_CUR` : `offset` is relative to the current position.
  - `SEEK_END` : `offset` is relative to the end of the file.
- Normally, returns the new position in the file. Returns -1 if an error occurs.

# System Calls (continued)

**Sample program segment for `lseek`:**

```
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);
}
```

**An interesting use of `lseek`:** Handout 10.2

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

# System Calls (continued)

**System calls `unlink` and `remove`:**

- **Prototypes:**

```
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.

# System Calls (continued)

**Reporting Errors:**

- **Purpose:** To provide more information when a system call reports error.
- Header file used: `<errno.h>`. (This file provides the 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].)
- Function `perror` available to print error messages:

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

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

# System Calls (continued)

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

```
int  fd;
if ((fd = open("/usr/nofile", O_RDONLY)) == -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.