CS3411 Systems Programming

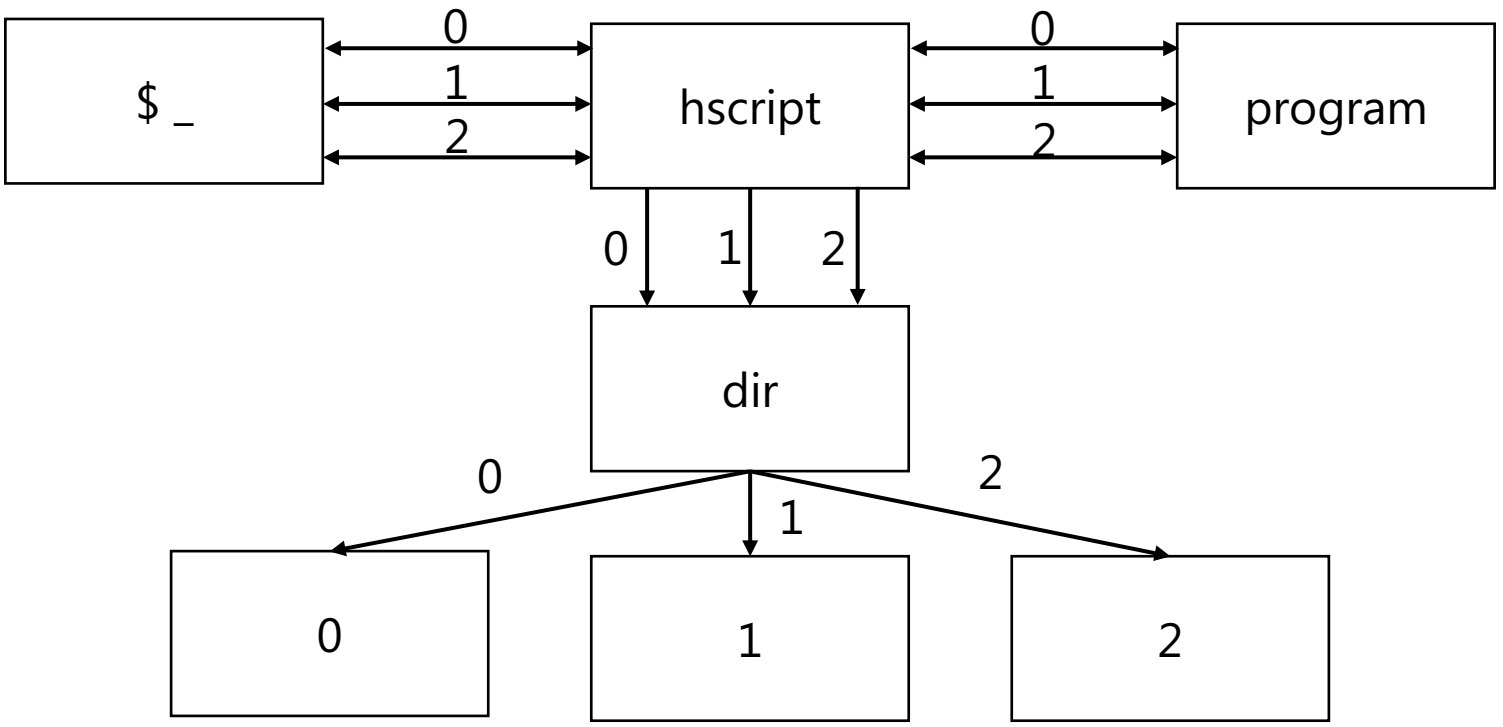
Project 4

Overview

- script-like program
- “script” is a tool that logs all activity on the terminal while running and writes to an output file
- Our version only logs activity for one program
- You will develop:
  - **hscript.c**
  - **Makefile**
    - **all** directive
    - **clean** directive
  - Any helper files needed

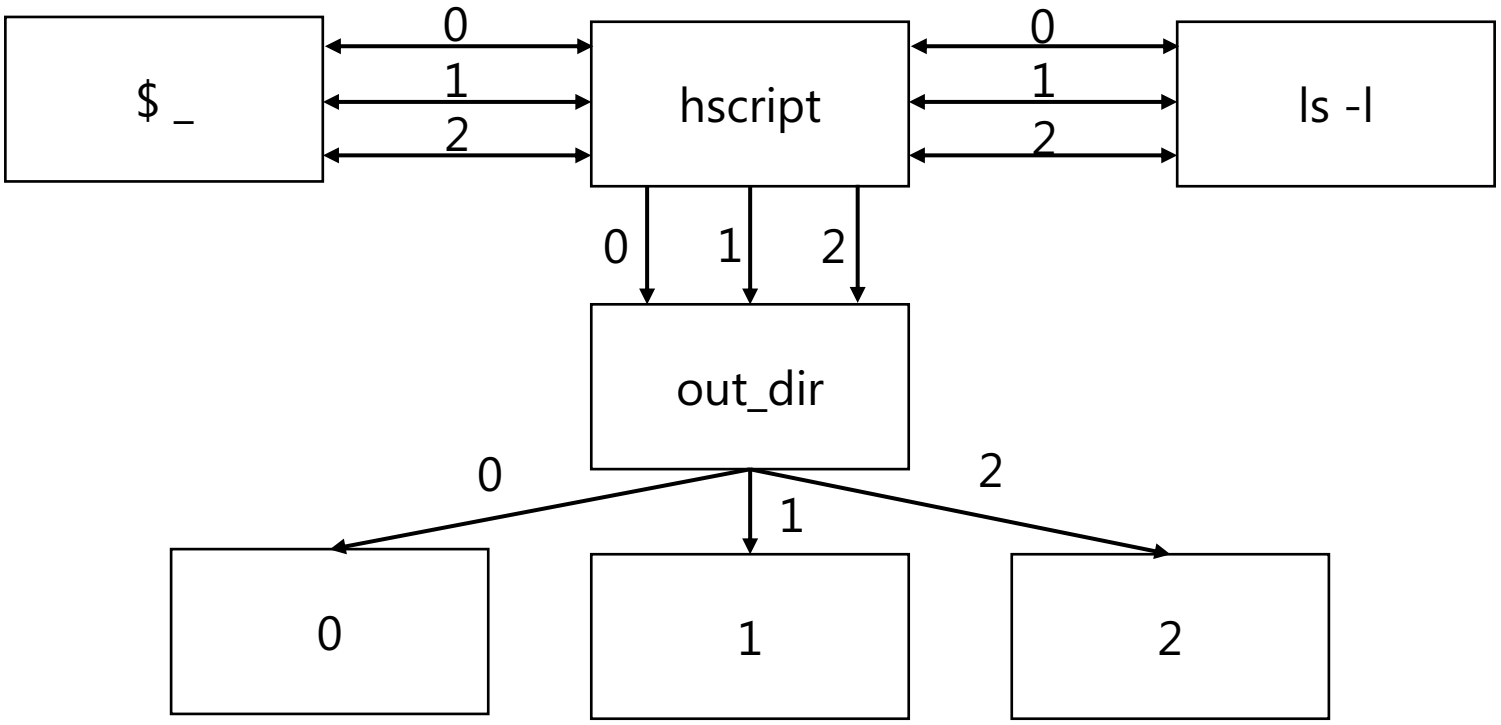
Architecture

$ ./hscript program <args> dir



Architecture

$ ./hscript ls -l out_dir



Syscalls

- Syscalls needed:
  - select()
  - wait()
  - signal()/sigaction()
  - dup()
  - fork()
  - close()
  - execv()/execvp()
  - pipe()
  - mkdir()
  - stat()
  - open()

Syscall Review

open(), pipe(), close(), fork(), exec(), wait(), signal()

open()

- Calling convention:

```
open(path, flags, mode);
```

- Returns an int file descriptor or -1 on error
- Ex usage:

```
int fd;
fd = open(“my_file”, O_RDONLY | O_TRUNC | O_CREAT, 0600);
```

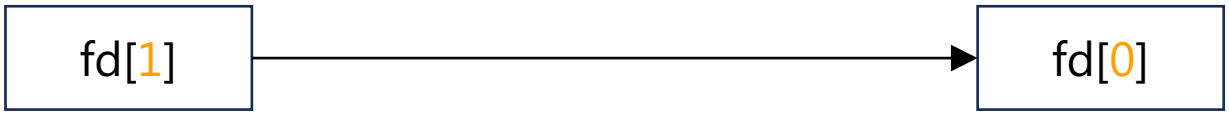
pipe()

- Calling convention:

```
pipe(pipe_fd_arr[2]);
```

- Returns 0 on success and -1 on error
- **Requires int array of size 2 as an argument**
- Ex usage:

```
int fd[2];
pipe(fd);
```



close()

- Calling convention:

```
close(fd);
```

- Returns 0 on success and -1 on error
- **Releases fd back into free pool**
- Ex usage:

```
int fd;
fd = open("my_file", O_RDONLY | O_TRUNC | O_CREAT, 0600);
close(fd);
```

fork()

- Calling convention:

```
fork();
```

- Returns 0 if the child, >0 if parent, and -1 if error
- **Makes a copy of parent's binary/program state and creates a new process**
- Ex usage:

```
int is_parent = fork();
if(is_parent) {
        //Parent
} else {
        //Child
}
```

execvp()

- Calling convention:

```
execvp(program, arguments);
```

- Returns -1 on error, does not return on success
- **Overwrites current process's binary/program state with the one given by "program"and passes in the argument list "arguments"**
- Ex usage:

```
char *program = "ls";
char *arguments = {"-a", NULL}; // Must be NULL terminated
execvp(program, arguments); //executes 'ls -a'
```

wait()

- Calling convention:

```
wait(status);
```

- Returns -1 on error and PID of terminated child on success
- **Waits for a child process to terminate then resumes execution**
- Ex usage:

```
int status;
if(fork()) //Parent
        wait(&status);
else //Child
        exit(0);
```

signal()

- Calling convention:

```
signal(signal_to_handle, handler);
```

- Returns SIG_ERR on error and previous value of signal handler on success
- **Allows the program to catch and handle signals sent to it by other processes**
- Ex usage:

```
void handler(){ //does something }
int main() {
        signal(SIGTERM, handler); // Allowed but not portable
        signal(SIGSEGV, SIG_IGN);
}
```

sigaction()

- Calling convention:

```
sigaction(signal_to_handle, new_action, old_action);
```

- Returns 0 on success and -1 on error
- **Allows the program to catch and handle signals sent to it by other processes**
- Ex usage:

```
void handler(){ //does something }
int main() {
        struct sigaction my_sigact_struct;
        my_sigact_struct.sa_handler = handler;

        sigaction(SIGTERM, &my_sigact_struct, NULL); // Portable
}
```

New Syscalls

mkdir(), stat(), select()

mkdir()

- Calling convention:

```
mkdir(name, mode);
```

- Returns 0 on success and -1 on error
- **Attempts to create directory and location specified by “name” with permissions “mode”**
- Ex usage:

```
char *dir = “/home/user/new_dir”;

// Creates directory new_dir at /home/user/ with 0700 mode
mkdir(dir, 0700);
```

stat()

- Calling convention:

```
stat(file, statbuf);
```

- Returns 0 on success and -1 on error
- **Writes information about “file” into statbuf**
- Ex usage:

```
struct stat statbuf;
char *my_file = “/home/user/a_file”;
if(stat(my_file, &statbuf) == -1)
        // ERROR
else
        // Info stored in statbuf
```

select()

- Calling convention:

```
select(num_fds, read_fds, write_fds, except_fds, timeout);
```

- Returns number of file descriptors contained in the three fd sets and -1 on error
- **Allows program to wait for a file descriptor(s) to become ready for I/O operation**
- Used in conjunction with
  - FD_CLR
  - FD_ISSET
  - FD_SET
  - FD_ZERO

select() Usage

```
int fd1[2], fd2, fd3;
pipe(fd1);
fd2 = open(“file”, O_RDONLY);
fd3 = open(“other_file”, O_WRONLY);
fd_set read_set;
fd_set write_set;
FD_ZERO(&read_set); // Must do this first to clear set
FD_ZERO(&write_set);
FD_SET(fd2, &read_set); // Add fd2 to read_set
FD_SET(fd1[0], &read_set);
FD_SET(fd3, &write_set); // Add fd3 to write_set
FD_SET(fd1[1], &write_set);

int ndfs = fd3 + 1; // must be highest numbered file descriptor of all sets plus 1
if (select(ndfs, &read_set, &write_set, NULL, NULL) > 0) { // No exception_set or timeout
             // One of the file descriptors is ready for I/O, test which one
             if(FD_ISSET(fd1[0], &read_set)) {} // if fd1[0] is ready
             if(FD_ISSET(fd[1], &write_set)) {} // if fd[1] is ready
             if(FD_ISSET(fd3, &write_set)) {}   // if fd3 is ready
             if(FD_ISSET(fd2, &read_set) {}     // if fd2 is ready
}
```