CSI 402 – Lecture 15

(Unix – Processes and Pipes)

Processes and Pipes

Ref: Chapter 7 of [HGS].

Pipe Mechanism in Unix:

- Allows a user to build complex commands by chaining together simple commands.
- Eliminates the need for temporary files.

Example 1: The following sequence of shell commands counts the number of lines containing the string `"this"` in the file `"file.txt"`.

```
% grep  this  file.txt  >  temp
% wc  -l  temp
% rm  temp
```

Processes and Pipes (continued)

Example 1 using pipes:

```
% grep this file.txt | wc -l
```

Notes:

- **Pipe:** A first-in first-out buffer with a "read-end" and a "write-end".
- To make the pipe in Example 1 to work:
  - The `stdout` for the `grep` program should be redirected to the write-end of pipe.
  - The `stdin` of the `wc` program should be redirected to the read-end of pipe.

General Information about Pipes

- There is *no* external file associated with a pipe; it is just a temporary internal buffer of limited capacity.
- The kernel blocks a process which tries to read from an empty pipe or which wants to write to a full pipe.
- The kernel also provides the synchronization needed among processes that use a pipe.
- A pipe can be used only between a process which created the pipe and the children of that process. (This restriction does *not* apply to "named pipes".)

System call `pipe`

- **Prototype:** `int pipe (int filedes[2])`
- **Header:** `<unistd.h>`
- `filedes`: A two-element array of type `int`; the function fills in the two entries of this array.
- Each array element stores a file descriptor.
- `filedes[0]`: File descriptor for the read-end of the pipe (i.e., the descriptor to be used with the `read` system call).
- `filedes[1]`: File descriptor for the write-end of the pipe. (i.e., the descriptor to be used with the `write` system call).
- The function returns `-1` if the call fails.

System call pipe (continued)

Sample code segment:

```
int  pd[2];  char  x[10], y[10];

if (pipe(pd) == -1) {
   fprintf(stderr, "Failed pipe.\n"); exit(1);
}

strcpy(x, "012345678");

write(pd[1], x, 10);

read(pd[0], y, 10);

printf("%s\n", y);
```

System call `dup2`

- **Prototype:** `int dup2 (int fd1, int fd2)`
- **Header:** `<unistd.h>`.
- An existing file descriptor `fd1` is duplicated as file descriptor `fd2`.
- If the file (or device) corresponding to descriptor `fd2` is open, then it is closed.
- The duplication and closing are carried out as an **atomic** step.

Using dup2 for Redirection

(a) Redirecting `stdin` to a file:

```
int fd;
if ((fd = open("x.dat", O_RDONLY)) == NULL) {
    .
    .  /* Code for error exit. */
}
if (dup2(fd, STDIN_FILENO) == -1) {
    .
    .  /* Code for error exit. */
}
close(fd);  /* Important! */
```

Note: At the end of the above segment (assuming that dup2 call was successful), all read operations to `stdin` will be redirected to the file `"x.dat"`.

Using dup2 ... (continued)

(b) Redirecting `stdout` to a pipe:

```
int pd[2];
if (pipe(pd) == -1) {
    .
    .  /* Code for error exit. */
}
if (dup2(pd[1], STDOUT_FILENO) == -1) {
    .
    .  /* Code for error exit. */
}
close(pd[1]);  /* Important! */
```

Note: At the end of the above segment (assuming that dup2 call was successful), all write operations to `stdout` will be redirected to the write-end of the pipe.

Program example: Handout 15.1.

Non-blocking Read/Write with Pipes

Reading Assignment: Program on pages 170–171 of [HGS].

- **Recall:** Read/write operation on a pipe may block the process.
- The `fcntl` system call can be used to make read/write non-blocking.

Sample Code Segment:

```
if (fcntl(pd[0],F_SETFL, O_NONBLOCK) == -1) {
    .
    .  /* Code for error exit. */
}
```

Non-blocking Read/Write ... (continued)

Notes:

- **Non-blocking read from a pipe:** If pipe is empty, the `read` call returns immediately with value `-1` and `errno` is set to `EAGAIN`.
- **Non-blocking write to a pipe:** If pipe is full, the `write` call returns immediately with value `-1` and `errno` is set to `EAGAIN`.

Program Example: Handout 15.2.

Named Pipes (or FIFOs)

- Unnamed pipes can only be used between processes which have an ancestral relationship.
- Unnamed pipes are temporary; they need to be created every time and are destroyed when the corresponding processes exit.
- Named pipes (FIFOs) overcome both of these limitations.

Named Pipes (continued)

System call `mkfifo`:

- **Prototype:**

```
int mkfifo (const char *path,  mode_t  mode)
```

- **Headers:** `<sys/types.h>` and `<sys/stat.h>`.
- `path`: Name of the FIFO to be created.
- `mode`: File permissions for the FIFO; will be modified using `umask`.
- Returns `-1` if the call fails and 0 otherwise.

Named Pipes (continued)

Notes:

- The `open` system call is used to obtain a file descriptor for a FIFO.
- Reading data from a FIFO and writing data to a FIFO can be done using `read` and `write` system calls.
- Non-blocking read/write on a FIFO can be done by using `O_NONBLOCK` in the call to `open`.

Program Example: Handout 15.3.

Handling Multiple Pipes/FIFOs

- A server may communicate with two more clients through two or more separate pipes/FIFOs.
- At any time, data may be available from more than one pipe/FIFO.
- The server may want to wait until data is available from at least one of the pipes/FIFOs.
- **A more general situation:** A server may want to monitor a set of file descriptors.
- System call `select` handles such situations.

Handling Multiple Pipes/FIFOs (continued)

Representing File Descriptor Sets:

- Usual bit representation for sets (one bit per element).
- If a bit is 1 (i.e., the bit is ON), then the process is "interested" in that file descriptor; otherwise, the bit is 0 (OFF).

(a) Data Type fd_set:

- To be used for representing and manipulating a set of file descriptors.
- **Header:** `<sys/time.h>`

Handling Multiple Pipes/FIFOs (continued)

(b) Permitted Operations:

- Initialize all bits to 0.

```
void  FD_ZERO (fd_set *f);
```

- Set a specific bit to 1.

```
void  FD_SET (int fd, fd_set *f);
```

- Set a specific bit to 0.

```
void  FD_CLR (int fd, fd_set *f);
```

- Check whether a specific bit is ON.

```
int  FD_ISSET (int fd, fd_set *f);
```

Handling Multiple Pipes/FIFOs (continued)

Structure timeval:

- **Struct Definition:**

```
struct timeval {
    long tv_sec;     /* Seconds. */
    long tv_usec;    /* Micro-seconds. */
};
```

- Used in `select` system call to specify how long the process should wait.
- The micro-seconds part can be used to specify fraction of a second.

System call `select`

- **Prototype:**

```
int select (int nfds,  fd_set *rfds,  fd_set *wfds,
            fd_set *efds,  struct timeval *tout)
```

- **Header:** `<sys/time.h>`
- `nfds`: The set of file descriptors to be monitored will be a subset of $\{0, 1, \ldots, \texttt{nfds} - 1\}$.
- For `nfds`, the symbolic constant `FD_SETSIZE` defined in `<sys/time.h>` may also be used.
- `rfds`: Pointer to the set of file descriptors which should be monitored for **reading**.
- `wfds`: Pointer to the set of file descriptors which should be monitored for **writing**.
- `efds`: Pointer to the set of file descriptors which should be monitored for **errors**.

System call `select` (continued)

- `tout`:
  - Pointer to the `timeval` structure. (Memory for the structure must have been allocated by the caller.)
  - Specifies the timeout time.
  - If this pointer is `NULL`, the call waits until an event of interest occurs for any of the file descriptors.
- The call to `select` returns 0 if timeout occurs and `-1` on error; otherwise, returns the number of file descriptors for which events of interest occurred.
- **Important Note:** The bit masks pointed to by `rfds`, `wfds` and `efds` are all modified by the call. (So, a user should keep a copy of the original masks.)

Program Example: Handout 15.4.