

The Shell, Processes and Basic Inter Process Communication (IPC) with System Calls

Michael Jantz

Dr. Prasad Kulkarni

Ishrak Hayet

The Shell

- Shell is a program that lets users communicate with the Operating System
- Typically, in a shell, users are given access to a command line interface (CLI) through which users can input textual commands
- The shell accepts the command, interprets it and executes it
 - The command will be an executable program that either comes with the OS or is created by the user
 - The shell usually looks for the program from the entries in the system's PATH environment variable (you can check using: 'echo \$PATH' on your terminal)
 - If the program is not in any of the directories of the PATH variable, then we have to input the command using its full path
 - If the command is in the current directory, we can use './<command>'

Processes

- Processes are nothing but programs that are being executed
- Processes are created in the following ways
 - When we execute a program, a process is created
 - A child process can be created from within another process using `fork()` system call (more on this later)

IPC at a Glance

- Operating Systems provide mechanisms for processes to cooperate and communicate with one another through Inter Process Communication (IPC)
- Pipes are one such mechanism to provide unidirectional communication channel between two processes
- We will work with pipes in today's lab

A Few System Calls

- System calls are special functions that are invoked through the kernel space
- We will use the following system calls for today's lab
 - fork - used to create child processes
 - waitpid - used when a parent process blocks itself until a specific child process exits
 - pipe - used to create a pipe for IPC
 - dup2 - used to duplicate a file descriptor into another file descriptor
 - execl - used to execute a specific program with arguments just like we would do on a shell
 - read - used to read from a file
 - write - used to write to a file

fork

- fork is a system call that creates a child process
 - Declaration: `pid_t fork(void);`¹
 - Invocation: `pid_val = fork();`
- The process that calls fork is the parent process
- The child process is created by duplicating the parent process
- Starting from the line after the fork call, all the remaining lines of code are executed for both the parent and child processes

¹ <https://www.man7.org/linux/man-pages/man2/fork.2.html>

fork (continued)

- Since the remaining lines of code are executed for both parent and child processes, we can use the return value from `fork` to identify the code sections that we want to run for either the parent or child process
 - Return value (`pid_val == 0`): child process section
 - When the child process looks at the `pid_val`, it finds 0
 - Return value (`pid_val > 0`): parent process section
 - When the parent process looks at the `pid_val`, it finds the unique process id of the child process (useful to keep track of the child processes)
 - Return value (`pid_val == -1`): error
 - When an error occurs, the *errno*¹ value is set and can be used by *perror*² to print error message
- The parent and child processes run in separate memory spaces
- The child inherits “copies” of parent’s attributes (e.g. *file descriptors*)

1) <https://man7.org/linux/man-pages/man3/errno.3.html>

2) <https://linux.die.net/man/3/perror>

waitpid

- waitpid is a system call that lets a parent process wait for the completion of a child process
 - Declaration: `pid_t waitpid(pid_t pid, int *status, int options);`¹
 - Invocation (for this lab): `waitpid(pid_val, NULL, 0);`
- Waiting for the child process prevents “orphan” and “zombie” processes (both waste system resources)
 - Orphan process:
 - When parent process finishes execution before child, the child becomes orphan process
 - Orphan process is adopted by the init or system daemon process
 - Zombie process:
 - When child process finishes execution before parent, the child becomes zombie process
 - If the parent process “waits” to read the exit status of the child, the child process is reaped from the process entry table and prevents the child from remaining a zombie process
 - The waitpid system call lets a parent process read the exit status of finished child processes and reaps off zombie processes

¹ <https://linux.die.net/man/2/waitpid>

File Descriptors

- A file descriptor is a unique, non-negative integer used to identify an open file
- File descriptors can be used with open, close, read and write system calls
- Some special file descriptors:
 - ¹STDIN_FILENO – a file descriptor for the standard input (keyboard)
 - ²STDOUT_FILENO – a file descriptor for the standard output (computer display)
 - Pipe file descriptors – two file descriptors for a pipe's read and write end (more on these later)
- Inside the PCB of a process, there is a file descriptor table which:
 - Keeps a mapping of file descriptors (used by the process) to actual files on the system
 - Is inherited by the children of the process

pipe

- Pipe is a “unidirectional” data channel that can be used by a process to communicate with other processes
 - Declaration: *int pipe(int pipefd[2]);*¹
 - Invocation:
int fd[2];
pipe(fd);
- fd[0] is the read end of the pipe
- fd[1] is the write end of the pipe
- **Close pipe ends when they are no longer needed (very important):**
close(fd[0]);
close(fd[1]);

¹ <https://man7.org/linux/man-pages/man2/pipe.2.html>

dup2

- dup2 is a system call that copies one file descriptor into another
 - Declaration: *int dup2(int oldfd, int newfd);*¹
 - Invocation: *dup2(fd_one, fd_two);*
- Can be used for I/O redirection
 - Input redirection:
 - *dup2(pipe_read_end, STDIN_FILENO);*
 - Input of the process is taken from the pipe
 - Output redirection:
 - *dup2(pipe_write_end, STDOUT_FILENO);*
 - Output of the process is sent to the pipe

¹ <https://man7.org/linux/man-pages/man2/dup.2.html>

exec

- `exec` represents a group of system calls that can be used to execute external programs just like we would from a terminal
- We will use *execl* for today's lab
 - Declaration: *int execl(const char* pathname, const char* arg, ..., (char*)NULL);*¹
 - Invocation: *execl(program_path, variable_number_of_args, (char*)NULL);*
- *exec* system calls replace the calling process's image by a new image
- So, lines of code after a successful `exec` call inside a "specific process" will not be executed

¹ <https://man7.org/linux/man-pages/man3/exec.3.html>

read

- read is a system call that is used to read from a file descriptor
 - Declaration: *ssize_t read(int fd, void *buf, size_t count);*¹
 - Invocation:

```
char buf[n];  
int numOfBytesRead = read(fd, buf, sizeof(buf));
```
- Reading from pipes:
 - Reading from a pipe will block the caller and read as long as any process has open write descriptors for that pipe
 - If all processes close the write end of a specific pipe, reading from that pipe will no longer block and instead return 0

¹ <https://www.man7.org/linux/man-pages/man2/read.2.html>

write

- Write is a system call that is used to write to a file descriptor
 - Declaration: *ssize_t write(int fd, void *buf, size_t count);*¹
 - Invocation:

```
char buf[n];  
int numOfBytesWritten = write(fd, buf, sizeof(buf));
```
- Writing to pipes:
 - If the read end of a pipe has been closed by all processes, writing to that pipe will result in SIGPIPE signal and the process trying to write to that pipe will be terminated

¹ <https://www.man7.org/linux/man-pages/man2/write.2.html>

Understanding Processes and IPC through the Shell

Example Shell Commands (Redirections)

- Some common bash operators:

- `foo < in.txt` – redirect standard input of program `foo` to `in.txt`.

- `foo > out.txt` – redirect standard output of program `foo` to `out.txt`.

- `foo >> out.txt` – redirect standard output of program `foo` to be appended to the file `out.txt`.

- `foo | bar` – redirect standard output of program `foo` to be the standard input to program `bar`.

- Bash is also a scripting programming language (complete with variables and if and while statements) that can be used to script the OS.

Utility Programs

- Any Unix distribution comes with several utility programs for interacting with the OS.
 - grep – search for strings in a file
 - find – find a particular file
 - du – Determine the disk usage of files and directories
 - ls – List files and their permissions
- Many, many more. Proficiency with these basic tools will make you a much more effective developer on your platform.
- Unix provides a manual (accessible from the shell) that documents the use and syntax of each core utility. e.g:
 - man grep

finder.sh

```
find $1 -name '*'.[ch] | xargs grep -c $2 | sort -t : +1.0 -2.0  
--numeric --reverse | head --lines=$3
```

- find \$1 -name '*'.[h]—Find files with .h extensions under the directory given by the first argument.
- xargs grep -c \$2— Search the set of files on standard input for the string given by the second argument. -c says that instead of printing out each usage in each file, give me the number of times \$2 is used in each file.
- sort— Sort standard input and print the sorted order to standard output. -t : +1.0 -2.0 says sort using the second column on each line (delimited by the ':' character) as a key. --numeric says to sort numerically (as opposed to alphabetically). --reverse says sort in reverse order.
- head— print only the first *n* lines of standard input. --lines=\$3 lets us set the number of lines with the third argument.

How finder.sh Works in the Shell

- When the user types this command at a shell, the shell parses the input, and issues system calls to create the processes and set up the pipes between these processes.
 - In this lab, we will implement what the shell would typically perform when given a command like this.
- Although to save time, our implementation will only work for pipelines of length 4 as opposed to arbitrarily long pipelines (as a shell would handle).

Getting Started

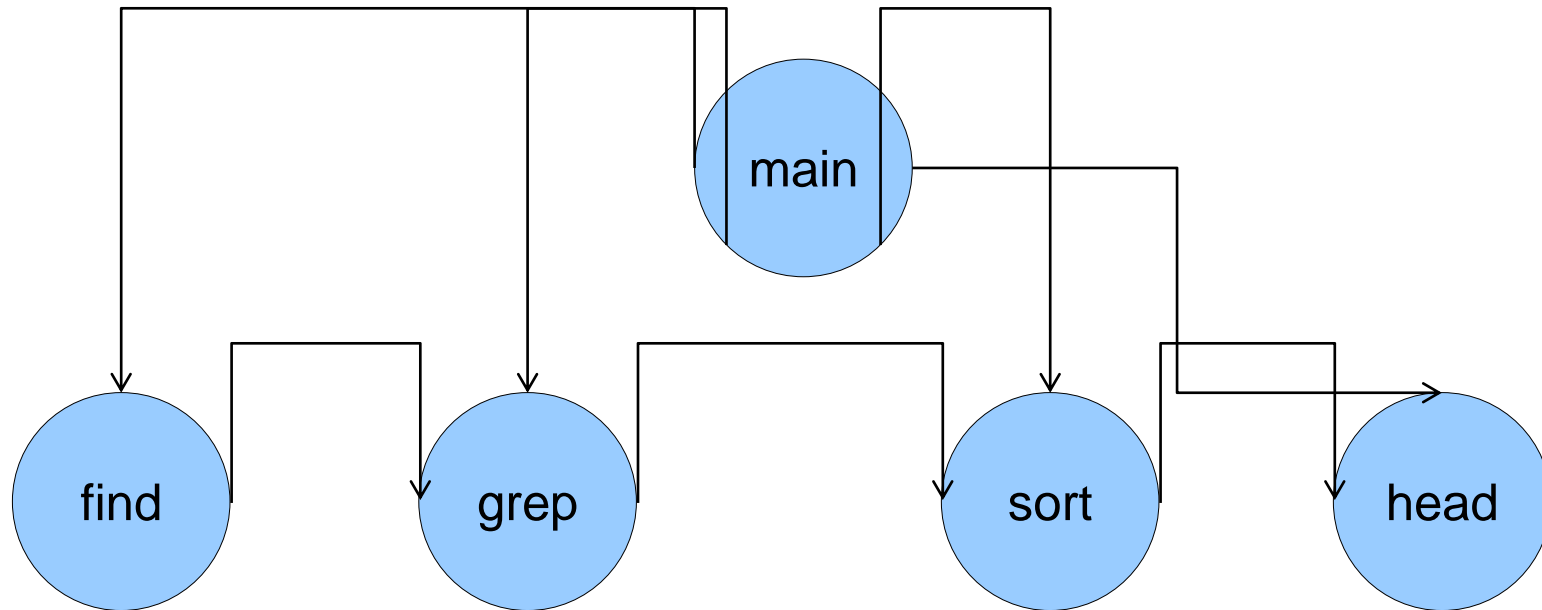
- The first thing to notice after untarring the tar file is the Makefile:
 - Notice the variables DIR, STR, and NUM_FILES and the command under the 'find' target.
 - Test the command. In this lab's directory, do:
 - `bash> make find`
- Should see the output as described two slides back.
- The goal of this lab is to write a program – `finder.c` – that produces the same output as the `finder.sh` command.

finder.c

- As it is given, this program is a skeleton for a four stage pipeline.
- All it currently does is start a process, which forks off four children (which do nothing), waits for them to finish, and exits.

finder.c (cont.)

- We want a program that forks off four children, sets up pipes between these children, and executes the appropriate command with each child:



Step 1 - Constructing the Pipeline

- Go ahead and try to setup the pipeline using the `pipe()` and `dup2()` system calls.
- Remember to `close()` unused file descriptors for each process.
- You may want to experiment with pipes using the `pipe.c` program first.
- Try to connect the processes in `pipe.c` so that the file read in the first process is written down a pipe which is read from in the second process.

Step 2 - Adding exec

- When you are confident your pipeline is working correctly, all that is left is to tell each process to exec the appropriate binary (e.g. find, grep, sort etc.)
- exec replaces the current process image with a new process image specified by a binary file name and arguments.
- Once the image is replaced, you have no control over what the process does (which is why it is recommended that you test the pipeline well before this step).

Example

```
if (pid_1 == 0) {
    /* First Child */
    char cmdbuf[BSIZE];
    bzero(cmdbuf, BSIZE);
    sprintf(cmdbuf, "%s %s -name '*'\"'.[ch]", FIND_EXEC, argv[1]);

    /* set up pipes */
    ...

    if ( (execl(BASH_EXEC, BASH_EXEC, "-c", cmdbuf, (char *) 0)) < 0) {
        fprintf(stderr, "\nError execing find. ERROR#%d\n", errno);
        return EXIT_FAILURE;
    }
}
```

Finishing Up

- After you have each completed your implementation, compile the finder program and run the test code:
 - `bash> make test`
- If the diff line does not produce an error, your implementation is correct.