PURDUE UNIVERSITY®

CS 25200: Systems Programming

Lecture 13: Files, fork(), and pipe()s

Prof. Turkstra

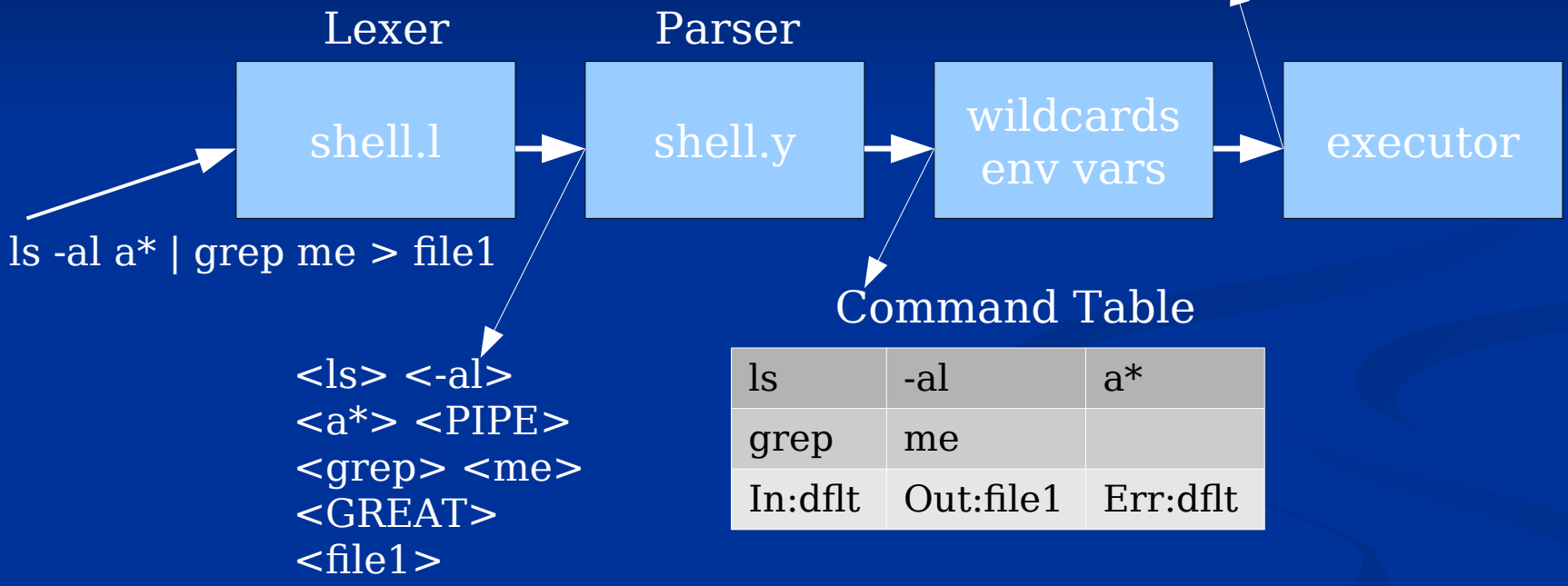
Lecture 13

- File table and descriptors
- fork() and exec()
- fd manipulation
- pipe()s

Shell

Final Command Table

ls	-al	aab	aaa
grep	me		
In:dflt	Out:file1	Err:dflt	



Command Table

ls	-al	a*
grep	me	
In:dflt	Out:file1	Err:dflt

I/O redirection - reading

- To redirect input for a program or command,

  `< file` `n < file` n is the file descriptor

  `<< file` `n << file` n is the file descriptor
- Example:

```
mail jeff@purdue.edu < my_document
```

I/O redirection - writing

- We can redirect the output from a program or command too!
  - `> file` and `n > file` Redirect output to `file`
  - `>> file` and `n >> file` Appends output to `file`
  - `>| file` and `n >| file` Overrides the noclobber option, if set
  - `>& number` Redirects the output to file descriptor `number`

More

- Redirect output and error to different files…

```
$ command > out.txt 2> err.txt
```

- Redirect stdout to out.txt, and stderr to stdout

```
$ command > out.txt 2>&1
```

I/O redirection - pipes

- Pipes enable a series of programs to work together
  `command_1 | command_2 | … | command_n`
- Functions a lot like `>` except stdout from `command_n-1` is redirected to stdin of `command_n`.
- Example:
  ```
  $ ls -l | wc -l
  46
  ```
  counts how many lines of text `ls` just output

System calls

- Really, we're just juggling file descriptors

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

File descriptors

- Open files, pipes, network sockets are referred to by an integer value called the file descriptor or fd
- This value is an index into a file descriptor table maintained in the kernel
  - Cannot be directly manipulated by a process

fd

- File descriptors can be viewed in a number of ways

```
$ lsof
$ ls /proc/PID/fd
```

- Most processes will have three default fds:
  0 – stdin, 1 – stdout, and 2 – stderr
  - Dictated by POSIX

File descriptor table

fd	flags	file pointer
0		
1	FD_CLOEXEC	
...		
n		

Open file object (file_t)

access mode	O_WRONLY
status flags	O_APPEND|O_SYNC
offset	52
reference count	1
i-node pointer	

- fcntl() can be used to manipulate fd flags and status flags

:-)

Open file “object”

- Holds most of a file’s state
  - Pointer to an inode (really a vnode)
  - Access mode (O_RDONLY, O_RDWR, O_WRONLY)
  - Status flags (O_ASYNC, O_APPEND, O_NONBLOCK, etc)
  - Offset – where the next read or write operation will commence
  - Reference count – similar to inodes

open() system call

```
int open(const char *pathname, int flags[, mode_t mode]);
```

- Flags includes:
  - Access mode (O_RDONLY, O_WRONLY, O_RDRWR) – required
  - File creation flags (O_CLOEXEC, O_CREAT, O_TRUNC, etc) – optional
  - File status flags (O_APPEND, O_SYNC, O_NONBLOCK, etc)
- Mode is your usual file creation mode

close() system call

```
int close(int fd);
```

- Decrements the reference count for the appropriate open file object
- Object is reclaimed if reference count == 0
- Returns -1 on error and sets errno
- Failing to close() fds results in a file descriptor leak
  - Arguably worse than a memory leak

errno

- When system call wrappers return -1, they usually set a global variable errno.
  - #include <errno.h>

fork() “system call”

```
pid_t fork(void);
```

- Wrapper for the clone system call
  - Don’t worry about this too much
- The only way to create a new process in *nix
- Creates an identical copy of the currently running process
  - Copy-on-write optimization avoids the overhead of duplicating memory

fork() it

- New process is a child of the parent process
- Open file descriptor table is copied
- Open file objects are shared
  - Reference count is increased by one

After fork()

Parent fdtable

fd	flags	file pointer
0		
1	FD_CLOEXEC	
...		
n		

Child fdtable

fd	flags	file pointer
0		
1	FD_CLOEXEC	
...		
n		

Open file object (file_t)

access mode	O_WRONLY
status flags	O_APPEND|O_SYNC
offset	52
reference count	2
i-node pointer	

:-)

Shared file_t

- We can use our shared file objects to establish a communication channel between the parent and child
  - Or even among multiple children
- Note: Solaris doesn't share the file position
  - This can cause headaches between platforms

Executing something else

- What if what we want to execute is not part of our program?
  - What if it is somewhere else?
- execve()!

execvp() “system call”

```
int execvp(const char *file, char *const argv[]);
```

- Replaces the current process image with a new process image
  - Really wraps execve()
- execvp() even searches $PATH for the executable, if no path provided
- Remember, argv must end with a NULL!
- Successful execve()’s never return

```
int main() {
  // Create a new process
  int ret = fork();
  if (ret == 0) {
    // Child process: execute “ls –al”
    char * const argv[3] = { “ls”, “-al”, NULL };
    execvp(argv[0], argv);
    // There was an error
    perror(“execvp”);
    exit(1);
  }
  else if (ret < 0) {
    // There was an error in fork
    perror(“fork”);
    exit(2);
  }
  else {
    // This is the parent process
    // ret is the pid of the child
    // Wait until the child exits
    waitpid(ret, NULL,0);
  } // end if
  exit(0); // No error
}// end main
```

wait() and friends

```
int waitpid(pid_t pid, int *wstatus, int options);
```

- Waits for changes in a child's state
  - Child terminates
  - Stopped by signal
  - Resumed by a signal
- No wait? ZOMBIES!
- Block until child changes state or interrupt handler caller
  - More later

Our shell

```
void execute_command()
{
  int ret;
  for (int i = 0; i < num_single_commands; i++) {
    ret = fork();
    if (ret == 0) { //child
      execvp(command[i]->arguments[0],
             command[i]->arguments);
      perror("execvp");
      exit(1);
    }
    else if (ret < 0) {
      perror("fork");
      return;
    }
  // Parent shell continue
  } // for
  if (!background) { // wait for last process
    waitpid(ret, NULL);
  }
}// execute
```

dup2() system call

```
int dup2(int oldfd, int newfd);
```

- Creates a copy of the file descriptor using the provided newfd
  - newfd will be silently closed if it is already open!
  - And it's atomic!

File descriptor table

fd	flags	file pointer
0		
1	FD_CLOEXEC	
2	FD_CLOEXEC	
n		

Open file object (file_t)

access mode	O_WRONLY
status flags	O_APPEND|O_SYNC
offset	52
reference count	2
i-node pointer	

:-)

- dup2(2, 1);

Redirecting stdout

```
int main(int argc, char **argv)
{
  // Create a new file
  int fd = open(“myoutput.txt”, O_CREAT|O_WRONLY|O_TRUNC,
                0664);
  if (fd < 0) {
    perror(“open”);
    exit(1);
  }
  // Redirect stdout to file
  dup2(fd,1);
  close(fd); // fd no longer needed.

  // Now printf that prints to stdout, will write to
  // myoutput.txt
  printf(“Hello world\n”);
}
```

dup() system call

```
int dup(int oldfd);
```

- Creates a copy of the file descriptor using the next available fd
- Handy if you want to “save” an fd for some reason
  - Hmmmm

pipe() system call

```
int pipe(int pipefd[2], int flags);
```

- Creates a unidirectional data channel
- Two file descriptors
  - pipefd[0]: read end
  - pipefd[1]: write end
- There is kernel buffering
- Flags are optional
  - O_NONBLOCK, O_CLOEXEC, etc
- Solaris has bidirectional pipes

Pipe dream

File descriptor table

fd	flags	file pointer
...		
3		
4		
n		

Open file object (file_t)

access mode	O_RDONLY
status flags	
offset	0
reference count	1
i-node pointer	

Open file object (file_t)

access mode	O_WRONLY
status flags	
offset	0
reference count	1
i-node pointer	

:-)

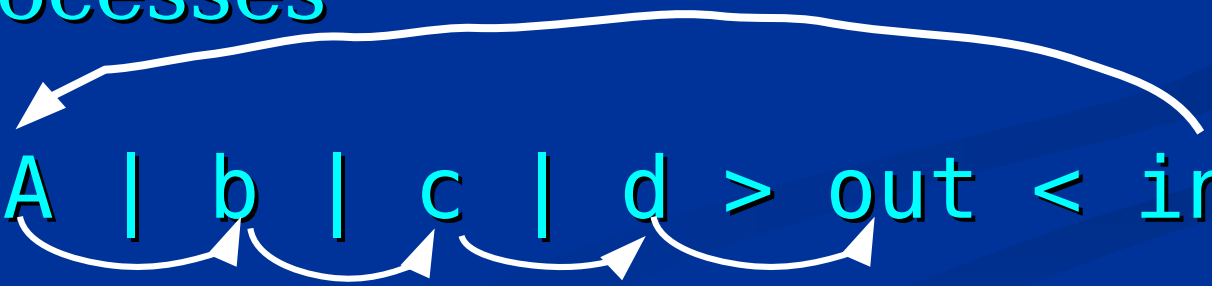
```
int fds[2];
pipe(fds);
fdpipe[0] == 3
fdpipe[1] == 4
```

A look at ls_grep.c

- Included in Lab 3's example directory

Executor

- Parent process sets up all piping and redirection before forking children
- Children inherit redirections
- Parent must save original fds and restore them
- stderr should be the same for all processes



```
execute() {
  //save input/output fds for later
  int tmpin = dup(0);
  int tmpout = dup(1);
  //set the initial input
  int fdin;
  if (infile) {
    fdin = open(infile,......);
  }
  else {
    // Use default input
    fdin = dup(tmpin);
  }
  int ret;
  int fdout;
  for (i = 0; i < num_single_commands; i++) {
    //redirect input
    dup2(fdin, 0);
    close(fdin);
    //setup output
    if (i == num_single_commands - 1){
      // Last single command
      if (outfile){
        fdout = open(outfile,......);
      }
      else {
        // Use default output
        fdout = dup(tmpout);
      }
    }
```

```
    else {
      // Not last single command
      //create pipe
      int fdpipe[2];
      pipe(fdpipe);
      fdout = fdpipe[1];
      fdin = fdpipe[0];
    }// if/else
    // Redirect output
    dup2(fdout,1);
    close(fdout);
    // Create child process
    ret = fork();
    if (ret == 0) {
      execvp(single_cmd[i]->args[0], single_command[i]->args);
      perror("execvp");
      exit(1);
    }
  } // for
  //restore in/out defaults
  dup2(tmpin,0);
  dup2(tmpout,1);
  close(tmpin);
  close(tmpout);
  if (!background) {
    // Wait for last command
    waitpid(ret, NULL, 0);
  }
} // execute
```

Questions?