PURDUE
UNIVERSITY®

CS 25200: Systems Programming

Lecture 14: Shell Executor, Processes, and Signals

Prof. Turkstra

Voting

- Out of state IDs are not accepted
- Federal IDs are fine (passport, military, etc)
- IDs must have an expiration date
  - First year this is enforced
- You can get a replacement Purdue ID for free October 21 – 25
  - Otherwise it is $10

Final Exam

- Tuesday, December 10 1pm – 3pm, LILY 1105

Midterm Exam

- Thursday, October 17 8pm - 10pm WALC 1055
- Make up lecture on Monday 11/25?
  - Week of Thanksgiving break

Lecture 13

- Shell executor
- exit() vs _exit()
- Processes
- Signals

Another look at ls_grep.c

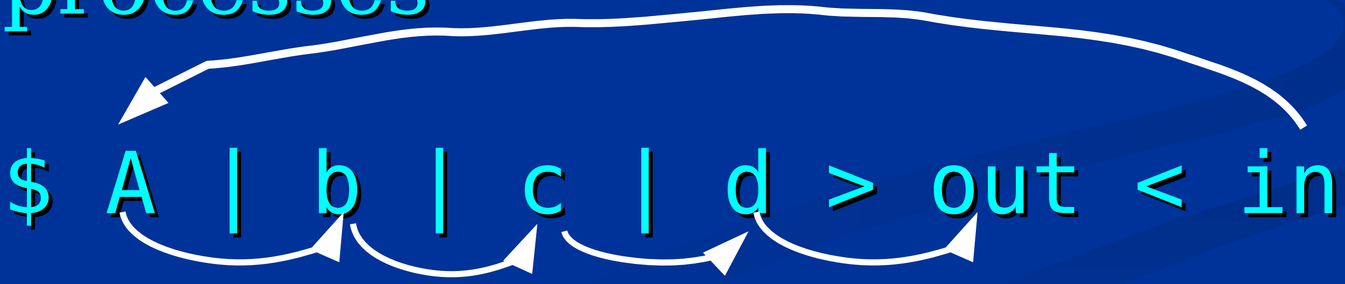
- Included in Lab 3’s example directory

FD_CLOEXEC

- Causes file descriptor to be closed any time a call is made to a function in the exec() family
- Flag is stored in the file descriptor table, not the file object
  - Why?

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



```
$ A | b | c | d > out < in
```

```
execute() {
  // save in/out
  int default_in = dup(0);
  int default_out = dup(1);

  // set the initial input
  int input_fd;
  if (input_file) {
    input_fd = open(input_file,......);
  }
  else {
    // Use default input
    input_fd = dup(default_in);
  }

  int ret;
  int output_fd;
  for (i = 0; i < num_single_commands; i++) {
    // redirect input
    dup2(input_fd, 0);
    close(input_fd);

    // setup output
    if (i == num_single_commands – 1){
      // Last single command
      if (output_file) {
        output_fd = open(output_file,......);
      }
      else {
        // Use default output
        output_fd = dup(default_out);
      }
    }
    else {
      // Not last single command
      // Create pipe
      int pipe_fds[2];
      pipe(pipe_fds);
      output_fd = pipe_fds[1];
      input_fd = pipe_fds[0];
    }

    // Redirect output
    dup2(output_fd,1);
    close(output_fd);

    // Create child process
    ret = fork();
    if (ret == 0) {
      close(default_in);
      close(default_out);
      execvp(scmd[i].args[0], scmd[i].args);
      perror(“execvp”);
      exit(1);
    }
  }
  // restore in/out defaults
  dup2(default_in,0);
  dup2(default_out,1);
  close(default_in);
  close(default_out);

  if (!background) {
    // Wait for last command
    waitpid(ret, NULL,0 );
  }

} /* execute() */
```

exit() and _exit()

- void exit(int status) – cause normal process termination
  - All atexit() and on_exit() registered functions are executed
    - In reverse order
  - stdio streams are flushed and closed
  - tmpfile() files are removed
- void _exit(int status) – terminate the process immediately
  - fds still closed

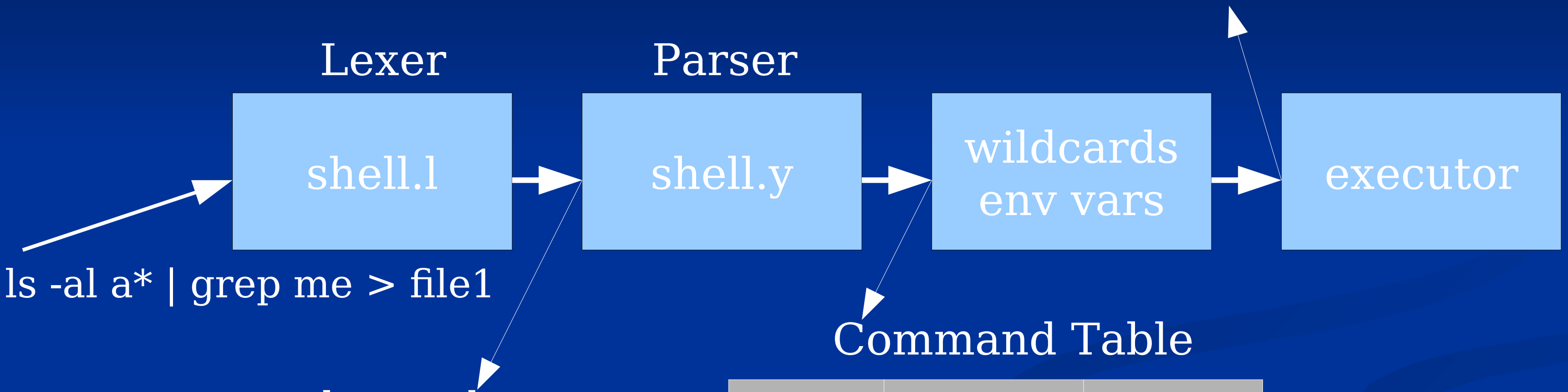
Shell strategy notes

- Remember, input_fd tracks the input for the next command
  - Will either be the input file fd (for the first command) or pipe_fds[0]
- Example only handles pipes and input/output redirection
- Have to handle stderr redirection
  - Applies to all processes
- Also have to handle the "append" cases

Shell

Final Command Table

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



Lexer

shell.l

Parser

shell.y

wildcards
env vars

executor

ls -al a* | grep me > file1

<ls> <-al>
<a*> <PIPE>
<grep> <me>
<GREAT>
<file1>

Command Table

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

Program vs. process

- A program is an executable file that contains a set of instructions
  - Usually stored on disk or other secondary storage
- A process is a program in execution
  - It resides, at least partially, in memory

Processes

- Programs may have multiple processes or instances running
  - E.g., multiple instances of Bash
- All processes have a parent
  - Except init, pid 1
- Remember ps?

Processes

- top, ps (-e, -ax, -f, -u)

Process properties

- PID: Process ID, index into process table
- Command/program name
- Arguments
- Environment variables
- Current working directory
- User ID
- stdin / stdout / stderr

Process ID

- Uniquely identifies running process
- Initial process (init, systemd) has PID 1
- PIDs assigned in ascending order
- Wrap around when limit is reached
- System call to get pid:

```
pid_t getpid();
```

Command and arguments

- Every process has a command name and 0 or more arguments
- Arguments are passed to main
  int main(int argc, char **argv);
  - argc: number of args
  - argv: arguments (argv[0] is the command name)

printargs.c

```
int main(int argc, char **argv) {
  for (int i = 0; i < argc; i++) {
    printf(“argv[%d]=\”%s\”\n”, i, argv[i]);
  }
}
```

Environment variables

- Array of strings, A=B, inherited from the parent process
- E.g.
  - PATH=/bin:/usr/bin – directories to search for commands
  - USER=<login> - username
  - HOME=/homes/turkstra
- Can modify .login or .bashrc
  - Aliases, etc too

Manipulating

- `export A=B`
  - All children will also see the change
- `A=B`
  - Only the current process will get it
- For example,
  `export PATH=$PATH:~/bin`
- Can run env or export to view current environment

In C

```
extern char **environ;

int main(int argc, char **argv) {
  int i = 0;
  while (environ[i] != NULL) {
    printf(“%s\n”, environ[i]);
    i++;
  }
}
```

Current directory

- Sometimes called working directory, current working director, or present working directory
  - $ pwd
- Every process has a current directory
  - Really just an inode

- Used to resolve relative paths
- Relative paths do not begin with /
  /root/hello.c – absolute
  hello.c – relative
  ../src/hello.c – relative
  src/hello.c – relative
  ~/src/hello.c – relative
- Change current directory: `$ cd dir`
- System call:

```
int chdir(const char *path);
int fchdir(int fd);
```

User identifier

- Processes have an effective user ID (euid)
  - Files created are owned by it
  - Most access checks use it
- …and a real uid (ruid)
  - Inherited from parent
  - Impacts signal sending/receiving

- Processes running as root can change their UID using:
  ```
  int setuid(uid_t uid);
  ```
  - It’s permanent.
  - E.g.: OpenSSH runs as root, but setuid()s to the connecting user after fork.
- Or…
  ```
  int seteuid(uid_t uid);
  ```
  - Allows setuid-root processes to temporarily drop root privileges
- Remember sudo and su?

Signals

- One form of inter-process communication (IPC)
- Asynchronous mechanism for the OS to communicate with a running process
- Processes can register signal handlers to perform certain actions for certain signals
- Signals are similar to interrupts

Some signals

- SIGHUP: Hangup
- SIGINT: Terminal interrupt
- SIGBUS: BUS error
- SIGKILL: Kill (cannot be ignored)
- SIGSEGV: Segmentation violation
- SIGTERM: Termination
- SIGCHLD: Child process has stopped, exited, or changed
- SIGUSR1, SIGUSR2, etc

Handling signals

- sighandler_t signal(int signum,
  sighandler_t handler)
- int sigaction(int signum,
  const struct sigaction *act,
  struct sigaction *oldact);

```
struct sigaction {
         void     (*sa_handler)(int);
         void     (*sa_sigaction)(int, siginfo_t *, void *);
         sigset_t   sa_mask;
         int        sa_flags;
         void     (*sa_restorer)(void);
     };
```

Flags

- SA_RESTART: Resume the function after a signal is handled properly
- Instead of returning EINTR
- SA_NOCLDSTOP: Only deliver SIGCHLD on termination, not stopping
- SA_ONSTACK: Use the signal stack
  - Must set it up first

Signals and lex

- Lex's scanner uses getc() to read from fd 0
- getc() is built on top of the read() system call
- Many blocking system calls will return if a signal is received
  - And set errno to EINTR
- What happens when we get SIGINT (or SIGCHLD)?
  - getc() returns -1!
- How do we stop it?

Keeping lex alive

- …use SA_RESTART

```
struct sigaction signal_action;

signal_action.sa_handler = sig_int_handler;
sigemptyset(&signal_action.sa_mask);
signal_action.sa_flags = SA_RESTART;
int error = sigaction(SIGINT,
                     &signal_action, NULL);
if (error) {
  perror("sigaction");
  exit(-1);
}
```

Questions?