COMP20200 Unix Programming

Lecture 10

CS, University College Dublin, Ireland

Assignment 2

Implement a simple shell.

- This lecture will cover some of the things needed.

The shell

1. Print a prompt.
2. Read a line from stdin
3. Perform some parsing on string
   - Separate redirects and pipes (<>|)
   - Separate builtins (`cd`, :, $)
4. Execute command
   - System call `execve`
5. Wait for command to complete.
6. Go to step 1.

execve

A system call.

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
                  char *const envp[]);
```

From `man execve`

executes the program pointed to by filename...

execve() does not return on success, and the text, data, and stack of the calling process are overwritten by that of the program loaded.

execve

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char** environ;

int main(int argc, char **argv){
  execve("/bin/ls", argv, environ);
  printf("hello?\n");
  return 0;
}
```

system

Execute a shell command. From `man system`:

system() executes a command specified in command by calling /bin/sh -c command, and returns after the command has been completed.

```
#include <stdio.h>
#include <stdlib.h>

extern char** environ;

int main(int argc, char **argv){
  system("ls /");
  printf("hello?\n");
  return 0;
}
```

- Can use system, it simplifies running command, but relies on */bin/sh*.

execve and fork

```
#include <stdio.h> #include <stdlib.h>
#include <unistd.h> #include <sys/wait.h>

extern char** environ;

int main(int argc, char **argv){
  pid_t child_pid;
  int child_status;

  child_pid = fork();
  if(child_pid == 0) {
    sleep(10);
    execve("/bin/ls", argv, environ); //lucky: ls not using argv[0]
    printf("Unknown command\n");
    exit(0);
  } else {
    //Parent process waits for child to finish
    printf("parent waiting\n");
    wait(&child_status); // waitpid(child_pid, &child_status, 0);
  }
  printf("parent exiting\n");
  return 0; }
```

EXEC(3)

The exec() family:

`execl, execlp, execle, execv, execvp, execvpe`

- Wrapper functions for the `execve` system call.
- Example: `execvp`

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

- it takes care of:
  - PATH
  - Environment variables.

execvp example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv){

  char* this_argv[4];
  char command[] = "echo";
  char arg1[] = "hello";
  char arg2[] = "world";

  this_argv[0] = command;
  this_argv[1] = arg1;
  this_argv[2] = arg2;
  this_argv[3] = NULL;
  execvp(comand, this_argv);
  printf("Unknown command\n");
  exit(1);
}
```

Note: `this_argv` is NULL terminated.

Command is both first argument to `execvp` AND `this_argv[0]`.

Full path for `execvp` not needed.

strtok

- The strtok() function parses a string into a sequence of tokens.
- On the first call to strtok() the string to be parsed should be specified in str.
- In each subsequent call that should parse the same string, str should be NULL.

```
#include <stdio.h> #include <stdlib.h> #include <string.h>

int main(void) {
  char input_str[] = "One two three four";
  char **output_str = malloc(sizeof(char*));
  int count = 0;
  char* temp = strtok(input_str, " ");
  while (temp != NULL) {
    output_str[count] = temp;
    count++;
    output_str = realloc(output_str, (count + 1) * sizeof(char*));
    temp = strtok(NULL, " ");
  }
  return 0; }
```

strtok

From previous slide:

- Can you print the contents of `output_str` without knowing `count`?
- Hint: You could if it was NULL terminated.
- If your input is from say `getline`, you may need to trim trailing new line character.

Signals

```
man 7 signals
```

- Limited form of inter-process communication.
- Software interrupts sent to a program, indicate an important event has occurred.
- Interrupt what process is currently doing.
- Indicates to program that user wants it to do something not in usual flow control.
- Each signal is represented by an integer and a symbolic name (`/usr/include/signal.h`)
- `kill` is a utility and system call to send signal to a process
  - name is historic: can be used to kill a process, but can also send other signals.

Signals

```
$ /bin/kill -L
 1 HUP       2 INT       3 QUIT      4 ILL       5 TRAP      6 ABRT
 7 BUS       8 FPE       9 KILL     10 USR1     11 SEGV     12 USR2
13 PIPE     14 ALRM     15 TERM     16 STKFLT   17 CHLD     18 CONT
19 STOP     20 TSTP     21 TTIN     22 TTOU     23 URG      24 XCPU
25 XFSZ     26 VTALRM   27 PROF     28 WINCH    29 POLL     30 PWR
31 SYS
$
```

Some common signals:

SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C).
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + \).
SIGFPE	8	Issued if an illegal mathematical operation is attempted.
SIGKILL	9	Process must quit immediately without any clean-up.
SIGSEGV	11	Memory access violation
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default).
SIGSTOP	19	Pause the execution of the process

Signals

- Can send signals with:
  - Command kill if you know its process ID.

    ```
    $ kill <pid>
    $ man 1 kill
    ```

  - With the system call kill()

    ```
    $ man 2 kill
    ```

  - With the library wrapper raise()

    ```
    $ man 3 raise
    ```

  - With keyboard to program in focus

    ```
    Ctrl-C  Ctrl-Z  Ctrl-\
    ```

ps

List running processes ps -ely. (man ps for more options.)

```
S   UID   PID  PPID C PRI  NI     SZ TTY        TIME CMD
S     0     1     0 0  80   0    907 ?      00:00:07 init
S     0     2     0 0  80   0      0 ?      00:00:00 kthreadd
S     0     3     2 0  80   0      0 ?      00:01:14 ksoftirqd/0
S     0     6     2 0 -40   -      0 ?      00:00:00 migration/0
S     0     7     2 0 -40   -      0 ?      00:00:04 watchdog/0
S     0     8     2 0 -40   -      0 ?      00:00:00 migration/1
S     0    10     2 0  80   0      0 ?      00:00:50 ksoftirqd/1
S     0    11     2 0 -40   -      0 ?      00:00:03 watchdog/1
S     0    12     2 0  60 -20      0 ?      00:00:00 cpuset
S     0   714     1 0  80   0   1673 ?      00:00:15 sshd
S  1000  1662     1 0  80   0  29064 ?      00:05:25 xfce4-panel
S  1000  1666     1 0  80   0  27385 ?      00:01:03 xfdesktop
S  1000  4536  1662 1  80   0 151423s?      01:29:22 chromium-brow
S     0 16819     2 0  80   0      0 ?      00:00:00 kworker/0:2
R  1000 16865 24342 0  80   0    722 pts/3  00:00:00 ps
S  1000 17093  3207 0  80   0   1769 pts/7  00:00:00 bash
```

Also: top for process info with real time update, can be sorted by cpu, memory usage etc.

kill

Using kill to send signal to a process.

```
kill -<signal> PID
```

```
$ sleep 600 &
[1] 17088
$ pgrep -l sleep
17088 sleep
$ kill -2 17088
$
```

Or to send signal to all processes with string “sleep”:

```
$ pkill -2 sleep
$
```

Kill system call

Kill is also a system call. Used to send any signal to any process.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Handling Signals

- Robust program needs to handle signals.
- A way to deliver asynchronous events to application.
- Process tells kernel what to do when a signal is received
    1. Signal can be ignored (except SIGKILL and SIGSTOP)
    2. Signal can be caught. Process registers a function with kernel, which is called by kernel when signal occurs
    3. Perform default action.

Signals

Minimum version (without error checking):

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void catch_function(int signo) {
    printf("Interactive attention signal caught.\n");
}

int main(void) {
    signal(SIGINT, catch_function);

    printf("Raising the interactive attention signal.");
    raise(SIGINT) != 0; // kill(getpid(), SIGINT);

    printf("Exiting.");
    return 0;
}
```

Note order of print statments when you run this.

Signals

Same as previous example, with error checking:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void catch_function(int signo) {
    printf("Interactive attention signal caught.\n");
}

int main(void) {
    if (signal(SIGINT, catch_function) == SIG_ERR) {
        fprintf(stderr, "An error occurred while setting a signal han
        return EXIT_FAILURE;
    }
    printf("Raising the interactive attention signal.");
    if (raise(SIGINT) != 0) {
        fprintf(stderr, "Error raising the signal.\n");
        return EXIT_FAILURE;
    }
    printf("Exiting.");
    return 0;
}
```

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo) {
  if (signo == SIGINT){
    printf("\nreceived SIGINT\n");
    fflush(stdout);
  }
}

int main() {
  printf("Catching SIGINT, try Ctrl+C  ");
  fflush(stdout);

  if (signal(SIGINT, sig_handler) == SIG_ERR)
  printf("\ncan't catch SIGINT\n");
  while(1)
    sleep(1);
  return 0;
}
```

Try the above code. Also try with commenting out `fflush()`

Handling Ctrl-C

```
#include <stdio.h> #include <stdlib.h>
#include <unistd.h> #include <sys/wait.h>
#include <signal.h>
extern char** environ;

int main(int argc, char **argv){
  pid_t child_pid;
  int child_status;

  child_pid = fork();
  if(child_pid == 0) {
    sleep(10);
    execve("/bin/ls", argv, environ);
    printf("Unknown command\n");
    exit(0);
  } else {
    signal(SIGINT, SIG_IGN);
    printf("parent waiting\n");
    wait(&child_status);
    signal(SIGINT, SIG_DFL);
  }
  printf("parent exiting\n");
  return 0; }
```

Environment variables

Environment variables can be got with a call to getenv.

```
#include <stdio.h>
#include <stdlib.h>

 int main(void) {
   char* path = getenv("PATH");
   printf("Path: %s\n", path);
   return 0;
 }
```

(More on setting environment variables later in the course)