Mini-Shell

Hyo-Bong Son (proshb@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Mini Shell
Mini Shell

- A shell program executing following commands
  - An application program that runs programs on behalf of the user

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Original command</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
<td>Change directory</td>
<td>cd</td>
</tr>
<tr>
<td>cls</td>
<td>Clear the terminal screen</td>
<td>clear</td>
</tr>
<tr>
<td>dir</td>
<td>List directory contents</td>
<td>ls</td>
</tr>
<tr>
<td>copy</td>
<td>Copy files and directories</td>
<td>cp</td>
</tr>
<tr>
<td>move</td>
<td>Move files and directories</td>
<td>mv</td>
</tr>
<tr>
<td>delete</td>
<td>Delete files and directories</td>
<td>rm</td>
</tr>
<tr>
<td>exit</td>
<td>Exit the shell</td>
<td>exit</td>
</tr>
</tbody>
</table>

- How to make it?
Mini Shell

Print current "PATH>"
   ex: ~>

Get a line

Make a process

Child: Execute the cmd

Child: Exit

Input error

Parent: Wait for the child

Signal
Processes

- **An instance of a program in execution.**
  - One of the most profound ideas in computer science.
  - Not the same as “program” or “processor”.

- **Process provides each program with two key abstractions:**
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.

- **How are these illusions maintained?**
  - Process executions interleaved (multitasking).
  - Address space managed by virtual memory system.
Creating a New Process

- **pid_t fork(void)**
  
  ```c
  if (fork() == 0) {
    printf("hello from child\n");
  } else {
    printf("hello from parent\n");
  }
  ```

  The `fork()` function creates a new process (child process) that is identical to the calling process (parent process).
  - Returns 0 to the child process.
  - Returns child’s `pid` to the parent process.

  Fork is interesting (and often confusing) because it is called once but returns twice.
Fork Example (1)

- **Key points**
  - Parent and child both run same code.
    - Distinguish parent from child by return value from `fork()`
  - Start with same state, but each has private copy.
    - Share file descriptors, since child inherits all open files.

```c
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
Fork Example (2)

- Key points
  - Both parent and child can continue forking.

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
Destroying a Process

- **void exit (int status)**
  - Exits a process.
    - Normally returns with status 0
  - **atexit()** registers functions to be executed upon exit.

```c
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    fork();
    exit(0);
}
```
Zombies (1)

- **Idea**
  - When a process terminates, still consumes system resources.
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead

- **Reaping**
  - Performed by parent on terminated child.
  - Parent is given exit status information.
  - Kernel discards the terminated process.

- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then child will be reaped by `init` process.
  - Only need explicit reaping for long-running processes.
    - e.g. shells and servers
Zombies (2)

```c
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1); /* Infinite loop */
    }
}
```

- **ps** shows child processes as “defunct”
- Killing parent allows child to be reaped
Synchronizing with Children

- **pid_t wait (int *status)**
  - suspends current process until one of its children terminates.
  - return value is the **pid** of the child process that terminated.
  - if **status != NULL**, then the object it points to will be set to a status indicating why the child process terminated.

- **pid_t waitpid (pid_t pid, int *status, int options)**
  - Can wait for specific process
  - Various options
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}

HC  Bye

HP

CT  Bye
Wait Example (2)

- If multiple children completed,
  - will take in arbitrary order.
  - Can use macros **WIFEXITED** and **WEXITSTATUS** to get information about exit status.

```c
void fork10() {
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
Running New Programs (1)

- **int execl (char *path, char *arg0, ..., NULL)**
  - loads and runs executable at **path** with arguments **arg0, arg1, ...**
    - **path** is the complete path of an executable
    - **arg0** becomes the name of the process
      - Typically **arg0** is either identical to **path**, or else it contains only the executable filename from path.
      - “real” arguments to the executable start with **arg1**, etc.
    - list of args is terminated by a (char *) 0 argument.
  - returns –1 if error, otherwise doesn’t return!

- **int execv (char *path, char *argv[])**
  - argv : null terminated pointer arrays
Running New Programs (3)

- Example: running `/bin/ls`

```c
main() {
    if (fork() == 0) {
        execl("/bin/ls", "ls", "/", 0);
    }
    wait(NULL);
    printf("completed\n");
    exit();
}

main() {
    char *args[] = {"ls", "/", NULL};
    if (fork() == 0) {
        execv("/bin/ls", args);
    }
    wait(NULL);
}
```
Shell

- **Definition**
  - An application program that runs programs on behalf of the user
    - sh: Original Unix Bourne Shell
    - csh: BSD Unix C Shell
    - tcsh: Enhanced C Shell
    - bash: Bourne-Again Shell

**Execution is a sequence of read/evaluate steps**

```c
int main()
{
    char cmdline[MAXLINE];
    
    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```
Simple Shell Example (1)

```c
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */
    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        if (!bg) { /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        } else /* otherwise, don’t wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```
Mini Shell

- **A shell program executing following commands**
  - An application program that runs programs on behalf of the user

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Original command</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
<td>Change directory</td>
<td>cd</td>
</tr>
<tr>
<td>cls</td>
<td>Clear the terminal screen</td>
<td>clear</td>
</tr>
<tr>
<td>dir</td>
<td>List directory contents</td>
<td>ls</td>
</tr>
<tr>
<td>copy</td>
<td>Copy files and directories</td>
<td>cp</td>
</tr>
<tr>
<td>move</td>
<td>Move files and directories</td>
<td>mv</td>
</tr>
<tr>
<td>delete</td>
<td>Delete files and directories</td>
<td>rm</td>
</tr>
<tr>
<td>exit</td>
<td>Exit the shell</td>
<td>exit</td>
</tr>
</tbody>
</table>

- **How to make it?**
Accessing File Metadata

```c
/* statcheck.c - Querying and manipulating a file's meta data */

int main (int argc, char **argv)
{
    struct stat st;
    char *type, *readok;

    stat(argv[1], &st);
    if (S_ISREG(st.st_mode)) /* file type*/
        type = "regular";
    else if (S_ISDIR(st.st_mode))
        type = "directory";
    else
        type = "other";
    if ((st.st_mode & S_IRUSR)) /* OK to read*/
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
bass> ./statcheck statcheck.c
  type: regular, read: yes
bass> chmod 000 statcheck.c
bass> ./statcheck statcheck.c
  type: regular, read: no
bass> ./statcheck statcheck.c
  type: regular, read: yes
bass> chmod 000 statcheck.c
bass> ./statcheck statcheck.c
  type: regular, read: no
```