

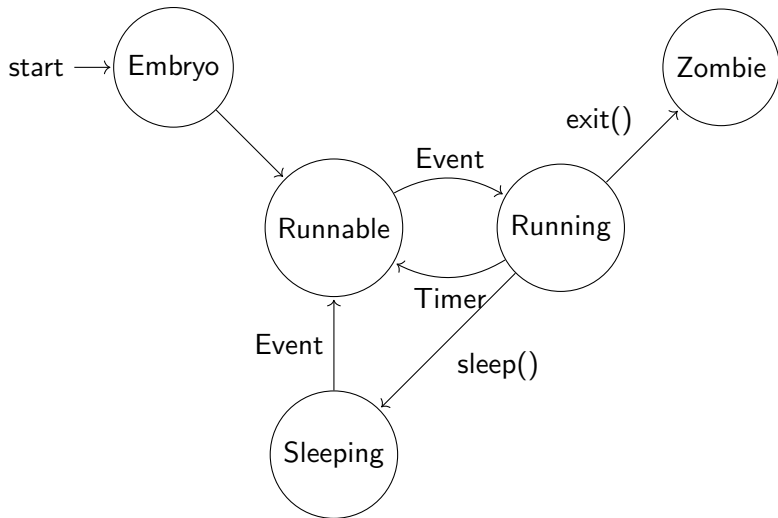
xv6©-rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)
Process programming

Carmi Merimovich

Tel-Aviv Academic College

February 19, 2017

The life of a process



All transition are due to INTERRUPTS.

Processes organization

- The processes are organized into a tree structure.
- The first process is created by the kernel as part of the initialization.
- Except for the first process, processes are created only by processes.
- The **fork()** system call creates a new process.
- The process invoking the **fork()** is called the **parent** process.
- The created process is called the **child** process.

Process creation/exiting system calls.

exit()

- Process resources are freed. (mostly)
- Process enters the ZOMBIE state.
- Children of the process are adopted by the first process.
- Process really dies when its parent **wait()**s on it.

pid=wait()

- If there are no child process returns error.
- If there are ZOMBIE children:
 - one of them (really) dies.
 - The **id** of the dead process is returned.
- If there are no ZOMBIE children:
 - wait for one of the children to become ZOMBIE.

exec(filename, argv)

- The code/data/stack of the current process is freed.
- The executable at **filename** is loaded and begins running at **main**.
- The **argv** parameter of **exec** is supplied to the new executable:
 - **main(argc, argv)**
- Nothing else changes:
 - Files open.
 - Current working directory.
 - etc.
- NOTE: This is NOT a **call**.
- The new code/data **replaces** the previous code/data.

pid=fork()

- A new child process is created.
- The parent process proceeds with the return value being the process id of the child process.
- The child process begins as a replication of its parent with one difference:
 - The return value is zero.

`pid=fork()` finegrained

Hypothetical **forking** code:

```
movl $1,%eax
int   $64
movl %eax,pid
```

Run time result:

movl \$1,%eax int \$64	
Parent continues	Child created
movl %eax,pid (pid==child process id)	movl %eax,pid (pid==0)

fork()/exec()/exit()

```
pid = fork();  
if (pid < 0) // fork failed  
    exit();  
if (pid == 0) { // Child code  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit(); // exec failed  
} else { // Parent process executes here  
  
}
```

First process and the **/init** program

The first process

- The kernel sets the initial state to:
 - cwd is `"/`.
 - No file is open.
- Sets standard input, standard output, and standard error, to the console device.
- Creates a process to run the shell (**sh**).
- Enters an infinite loop of **wait()**'s.
- It uses the system calls:
 - **open**.
 - **mknod**.
 - **dup**.
 - **fork**.
 - **exec**.
 - **exit**.

/init (1): Sets standard input and output

```
int main(void) {  
    int pid, wpid;  
  
    if (open("console", O_RDWR) < 0) {  
        mknod("console", 1, 1);  
        open("console", O_RDWR);  
    }  
    dup(0);    // stdout  
    dup(0);    // stderr
```

/init (2): Forks to shell, wait loop

```
for (;;) {
    printf(1, "init:_starting_sh\n");
    pid = fork();
    if(pid < 0) {
        printf(1, "init:_fork_failed\n");
        exit();
    }
    if (pid == 0) {
        char *argv[] = { "sh", 0 };
        exec("sh", argv);
        printf(1, "init:_exec_sh_failed\n");
        exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
        printf(1, "zombie!\n");
}
```

sh main functionality

(way, **way**, too much user mode code here to be of interest!)

sh main loop

```
while (read(0, cmd, ...) > 0) {  
    if (cmd is internal command)  
        executeInternalCmd(cmd);  
    else  
        forkExternalCmd(cmd);  
}  
exit();
```

- Internal cmd “cd” causes execution of the `chdir` system call.
- External commands are assumed to be executable files.

sh example: Simple exec

Typing:

ls

will use the following code,

```
pid = fork();  
if (pid == 0) {  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Simple exec

Typing:

ls

will use the following code, where the parent **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Simple exec

Typing:

ls

will use the following code, and the child **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Simple exec

Typing:

```
ls -l
```

will use the code,

```
pid = fork();  
if (pid == 0) {  
    char *argv[] = {"ls", "-l", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Simple exec

Typing:

```
ls -l
```

will use the code, where the parent **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    char *argv[] = {"ls", "-l", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Simple exec

Typing:

```
ls -l
```

will use the code, and the child **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    char *argv[] = {"ls", "-l", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Output redirection

Typing:

```
ls > a.txt
```

will use the code,

```
pid = fork();  
if (pid == 0) {  
    close (1);  
    open("a.txt", O_CREAT);  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Output redirection

Typing:

```
ls > a.txt
```

will use the code, where the parent **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    close (1);  
    open("a.txt", O_CREAT);  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```


sh example: Output redirection

Typing:

```
ls > a.txt
```

will use the code, and the child **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    close (1);  
    open("a.txt", O_CREAT);  
    char *argv[] = {"ls", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Output redirection

Typing:

```
ls -l > b.txt
```

will use the code,

```
pid = fork();  
if (pid == 0) {  
    close (1);  
    open("b.txt", O_CREAT);  
    char *argv[] = {"ls", "-l", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Output redirection

Typing:

```
ls -l > b.txt
```

will use the code, where the parent **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    close (1);  
    open("b.txt", O_CREAT);  
    char *argv[] = {"ls", "-l", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Output redirection

Typing:

```
ls -l > b.txt
```

will use the code, and the child **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    close (1);  
    open("b.txt", O_CREAT);  
    char *argv[] = {"ls", "-l", 0};  
    exec("ls", argv);  
    exit();  
}  
wait();
```

sh example: Input redirection

Typing:

```
sh < b.txt
```

will use the code,

```
pid = fork();  
if (pid == 0) {  
    close (0);  
    open("b.txt", O_RDONLY);  
    char *argv[] = {"sh", 0};  
    exec("sh", argv);  
    exit();  
}  
wait();
```

sh example: Input redirection

Typing:

```
sh < b.txt
```

will use the code, where the parent **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    close (0);  
    open("b.txt", O_RDONLY);  
    char *argv[] = {"sh", 0};  
    exec("sh", argv);  
    exit();  
}  
wait();
```

sh example: Input redirection

Typing:

```
sh < b.txt
```

will use the code, and the child **sh** executes:

```
pid = fork();  
if (pid == 0) {  
    close (0);  
    open("b.txt", O_RDONLY);  
    char *argv[] = {"sh", 0};  
    exec("sh", argv);  
    exit();  
}  
wait();
```

sh example: Pipe

Typing:

```
cat a.bat | sh
```

will use the code:

```
int p[2];
pipe(p);
pid = fork();
if (pid == 0) {
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"cat", 0};
    exec("cat", argv);
    exit();
}
```

```
pid = fork()
if (pid == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"sh", 0};
    exec("sh", argv);
    exit();
}
close(p[0]);
close(p[1]);
wait();
wait();
```


sh example: Pipe

Typing:

```
cat a.bat | sh
```

will use the code: where the parent **sh** executes:

```
int p[2];
pipe(p);
pid = fork();
if (pid == 0) {
    close (1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"cat", 0};
    exec("cat", argv);
    exit();
}
```

```
pid = fork()
if (pid == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"sh", 0};
    exec("sh", argv);
    exit();
}
close(p[0]);
close(p[1]);
wait();
wait();
```

sh example: Pipe

Typing:

```
cat a.bat | sh
```

will use the code: the first child **sh** executes:

```
int p[2];
pipe(p);
pid = fork();
if (pid == 0) {
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"cat", 0};
    exec("cat", argv);
    exit();
}
```

```
pid = fork()
if (pid == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"sh", 0};
    exec("sh", argv);
    exit();
}
close(p[0]);
close(p[1]);
wait();
wait();
```

sh example: Pipe

Typing:

```
cat a.bat | sh
```

will use the code: **the second child sh executes:**

```
int p[2];
pipe(p);
pid = fork();
if (pid == 0) {
    close (1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"cat", 0};
    exec("cat", argv);
    exit();
}
```

```
pid = fork()
if (pid == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    char *argv[] = {"sh", 0};
    exec("sh", argv);
    exit();
}
close(p[0]);
close(p[1]);
wait();
wait();
```