Chapter 5 Process API

5.1 The fork() System Call

Process creation on Unix: is a bit special.

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

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
                rc, (int) getpid());
    }
    return 0;
}

```

Figure 5.1: **Calling `fork()` (`p1.c`)**

Process Identifier (PID)

- Used to name the process

- Is useful if you want to do something with the process, such as stop it or suspend it.

The fork() creates a new process

- The strange part: exact copy of the calling process.
- Looks like two copies of p1, both are about to return from the fork() system call.

The creating process is called **parent**

The new process is called **child**.

- It does not start running at main()
- Instead it starts its life as if it had called fork() itself.

The child (is not an exact copy of the parent); it has its own

- Copy of the address space (memory)
- Registers, PC, SP etc.

The value returned from fork() is different for parent and child:

- Child: 0
- Parent: gets the PID of the newly created process (child)

This allows us to determine if we are the child or parent by checking the return value from fork().

The p1.c program is not deterministic.

- We don't know whether the child or the parent runs first

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

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        int rc_wait = wait(NULL);
        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
                rc, rc_wait, (int) getpid());
    }
    return 0;
}

```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

- Child runs first, then the parent and vice versa.
- This non-determinism is due to the scheduler — and we cannot usually make strong assumptions about what it will do.

5.2 The Wait() system call

Sometimes useful to wait for the child to finish —> use wait().

The parent process calls wait() to delay its execution until the child finishes.

When the child is done, wait() returns to the parent (with the child's PID as the return code)

Q: Why will this make the output deterministic?

A: Yes, The child will always print first.

Why? Two cases:

1. Child runs and prints first
2. Parent runs first, but waits for the child to finish.

Only when the child has finished, will the parent print.

5.3 Finally, the exec() system call

fork() is only useful if you want to run multiple copies of the same program

This is where exec() is useful.

The p3.c program runs the wc command using execvp().

% wc = word count

% wc -l = line count

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

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {              // parent goes down this path (main)
        int rc_wait = wait(NULL);
        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
                rc, rc_wait, (int) getpid());
    }
    return 0;
}

```

Figure 5.3: **Calling `fork()`, `wait()`, And `exec()` (`p3.c`)**

exec() does:

- Load code and static data from executable file (wc)
- Overwrites current code segment and static data of p3
- Heap and stack re-initialized
-

exec() does not create a new process

- It transforms the currently running program, which was formerly p3
- Into a different running program, in this case wc.
-

5.4 Why? Motivating the API

Why this odd interface for something as simple as creating a process

- Turns out: separation of fork() and exec() is essential for building a Unix shell
- Let's shell code run after the call to fork(), but before the call to exec()
  - This can alter the environment of the "about-to-be-run" program
  - Enables many useful features

[Pipes and Redirection]

Shell:

- Shows you a prompt
- Wait for you to type something into the prompt
- Type a command (an executable program with arguments)
  - Shell figures out where the executable is in the filesystem
    - ./p1
    - PATH is being searched for executable files when typed at the shell prompt
    - Avoid put the current directory (that is the dot .) in the PATH because it can cause problems…
  - Calls fork() - to create a child process
  - Calls exec() - to run the command
  - Calls wait() - to wait for the command to finish
  - When returns the shell prints another prompt

Separating fork() and exec() allows redirection…

% wc p3.c > newfile.txt

Output from wc command is redirected from stdout to newfile.txt

Shell accomplishes this as follows:

- When child is created, before calling exec()
  - The shell closes stdout and opens newfile.txt
  - Any subsequent output by the child will be transparently rerouted to the newfile.txt instead of the screen
  - Avoiding to send the output from wc to stdout — it is only sent newfile.txt

Pipes: pipe() system call

- Output of one process is connected to an in-kernel pipe (i.e., queue), and
- Input of another process is connected to the same pipe.
-

Can create chains of commands that are strung together:

% grep ssh *.md | wc -l

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

int main(int argc, char *argv[]) {
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {              // parent goes down this path (main)
        int rc_wait = wait(NULL);
    }
    return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4.c)**

5.5 Process Control and Users

- signal() system call: can send signals to process
  - Suspend process: CTRL-Z (Signal: SIGTSTP)
  - Resume process with fg
  - Interrupt signal: (CTRL-C (Signal: SIGINT)
    - Normally terminate the process
- Can write handler to catch signals, and do special processing, e.g., save application state.
- Multiuser system: Not everyone can send signals to all processes
  - Only the user owning a process can send signals to it
  - And the admin/super/root user can of course send signals to all processes

TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "**Get it right**. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful. Here, the UNIX designers simply got it right. And because Lampson so often "got it right", we name the law in his honor.

HotOS'19: A fork() in the road.

Abstract:

The received wisdom suggests that Unix's unusual combination of fork() and exec() for process creation was an inspired design. In this paper, we argue that fork was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability. We catalog the ways in which fork is a terrible abstraction for the modern programmer to use, describe how it compromises OS implementations, and propose alternatives.

As the designers and implementers of operating systems, we should acknowledge that fork's continued existence as a first-class OS primitive holds back systems research, and deprecate it. As educators, we should teach fork as a historical artifact, and not the first process creation mechanism students encounter.

The paper instead suggest using posix_spawn():

man posix_spawn()