**5: Interlude: proces API**

libro:

interludes: intermedios

we discuss process creation in U NIX systems. U NIX presents one of the most intriguing ways to create a new process with a pair of system calls: fork() and exec().A third routine, wait(), can be used by a process wishing to wait for a process it has created to

complete.

5.1 The fork() System Call

The fork() system call is used to create a new process.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char \*argv[]) {

printf("hello world (pid:%d)\n", (int) getpid()); //imprime junto con el identificador de proceso(PID))

int rc = fork(); //el proceso llama al fork

if (rc < 0) {

// fork failed

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)

When you run this program (called p1.c), you’ll see the following:

prompt> ./p1

hello world (pid:29146)

hello, I am parent of 29147 (pid:29146)

hello, I am child (pid:29147)

prompt>

-in U NIX systems, the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running.

-The process calls the fork() system call, which the OS provides as a way to create a new process. The odd part: the process that is created is an (almost) exact copy of the calling process. That means that to the OS, it now looks like there are two copies of the program p1 running, and both are about to return from the fork() system call. The newly-created process (called the child, in contrast to the creating parent) doesn’t start running at main(), like you might expect (note, the “hello, world” message only got printed out once); rather, it

just comes into life as if it had called fork() itself.

#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)

-the child isn’t an exact copy. Although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns

to the caller of fork() is different. Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero.

-the output (of p1.c) is not deterministic. When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. In our example (above), the parent did and thus printed out its message first. In other cases, the opposite might happen, as we show in this output trace:

prompt> ./p1

hello world (pid:29146)

hello, I am child (pid:29147)

hello, I am parent of 29147 (pid:29146)

prompt>

The CPU scheduler, determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first. This nondeterminism, as it turns out, leads to some interesting problems, particularly in multi-threaded programs.

-the parent process calls wait() to delay its execution until the child finishes executing. When the child is done, wait() returns to the parent.

5.2 The wait() System Call

Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with the wait() system call (or its more complete sibling waitpid())

-Adding a wait() call to the code above makes the output deterministic.

prompt> ./p2

hello world (pid:29266)

hello, I am child (pid:29267)

hello, I am parent of 29267 (rc\_wait:29267) (pid:29266)

prompt>

-There are a few cases where wait() returns before the child exits

5.3 Finally, The exec() System Call

A final and important piece of the process creation API is the exec()

system call 3 . This system call is useful when you want to run a program

that is different from the calling program

-On Linux, there are six variants of exec(): execl, execlp(), execle(), execv(), execvp(), and execvpe(). Read the man pages to learn more.

#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)

-often you want to run a different program; exec() does just that (Figure 5.3).

the child process calls execvp() in order to run the program wc, which is the word counting program. In fact, it runs wc on the source file p3.c, thus telling us how many lines, words, and bytes are found in the file:

prompt> ./p3

hello world (pid:29383)

hello, I am child (pid:29384)

29 107 1030 p3.c

hello, I am parent of 29384 (rc\_wait:29384) (pid:29383)

prompt>

The fork() system call is strange; its partner in crime, exec(), is not so normal either. What it does: given the name of an executable (e.g., wc), and some arguments (e.g., p3.c), it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of

the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the argv of that process. Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc). After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

5.4: Why? Motivating The API

why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of fork() and exec() is essential in building a U NIX shell, because it lets the shell run code after the call to fork() but before the call to exec(); this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

The shell is just a user program . It shows you a prompt and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls fork() to create a new child process to run the command,

calls some variant of exec() to run the command, and then waits for the command to complete by calling wait(). When the child completes, the shell returns from wait() and prints out a prompt again, ready for your next command.

The separation of fork() and exec() allows the shell to do a whole bunch of useful things rather easily.

prompt> wc p3.c > newfile.txt

the output of the program wc is redirected into the output file newfile.txt

Here is the output of running the p4.c program:

prompt> ./p4

prompt> cat p4.output

32 109 846 p4.c

prompt>

#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

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"); // arg: file to count

myargs[2] = NULL; // mark 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

Beyond fork(), exec(), and wait(), there are a lot of other interfaces for interacting with processes in U NIX systems. For example, the kill() system call is used to send signals to a process, including directives to pause, die, and other useful imperatives. For convenience, in most U NIX shells, certain keystroke combinations are configured to

deliver a specific signal to the currently running process; for example, control-c sends a SIGINT (interrupt) to the process (normally terminating it) and control-z sends a SIGTSTP (stop) signal thus pausing the process in mid-execution (you can resume it later with a command, e.g., the fg built-in command found in many shells).

-process groups

-signal() system call to “catch”various signals; doing so ensures that when a particular signal is delivered to a process, it will suspend its normal execution and run a particular piece of code in response to the signal.

5.6 Useful Tools

-ps command allows you to see which processes are running; read the man pages for some useful flags to pass to ps

-top is also quite helpful, as it displays the processes of the system and how

much CPU and other resources they are eating up.

-**The command kill can be used to send arbitrary signals to processes, as can the slightly more user friendly killall. Be sure to use these carefully; if you accidentally kill your window manager, the computer you are sitting in front of may become quite difficult to use.**

-A system generally needs a user who can administer the system, and is not limited in the way most users are. In U NIX -based systems, these special abilities are given to the superuser (sometimes called root)

5.7: Summary

A SIDE : K EY P ROCESS API T ERMS

• Each process has a name; in most systems, that name is a number known as a process ID (PID).

• The fork() system call is used in U NIX systems to create a new process. The creator is called the parent; the newly created process is called the child. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.

• The wait() system call allows a parent to wait for its child to complete execution.

• The exec() family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.

• A U NIX shell commonly uses fork(), wait(), and exec() to launch user commands; the separation of fork and exec enables features like input/output redirection, pipes, and other cool features, all without changing anything about the programs being run.

• Process control is available in the form of signals, which can cause jobs to stop, continue, or even terminate.

• Which processes can be controlled by a particular person is encapsulated in the notion of a user; the operating system allows multiple users onto the system, and ensures users can only control their own processes.

• A superuser can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.

**Teorico notas:**

API: funciones de procesos??

Fork

file p1→p ver de que se trata el archivo

pront: lo que me permite escribir

fork→ divide el programa – divide en dos entonces va haciendo un arbolito binario

poun couter: es independiente y no podes decir nada a cerca de la velocidad de esos core??

bomba fork: perdes el control de la maquina: solo te quedaria rebootearla

main(){

while(1)

{

fork();

}

}

join en unix: wait → el padre espera hasta que el hijo termine

ls: genera un fork que en el programa hijo genera un que…?????

fig 5.3:

#include

#include

#include

#include

#include

<stdio.h>

<stdlib.h>

<unistd.h>

<string.h>

<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"); ///file print f escribi un archivo, escribi el error al standard error

exit(1); ///llamada a sistema

} else if (rc == 0) { // child (new process) ///soy el hijo

printf("hello, I am child (pid:%d)\n", (int) getpid());

char \*myargs[3];

myargs[0] = strdup("wc"); // program: "wc" (word count) //necesito crear memoria

myargs[1] = strdup("p3.c"); // argument: file to count

myargs[2] = NULL; // marks end of array //fin de los argumentos

//ACA yo soy el hijo cree el proceso pero todavia no ejecute el programa y puedo hacer dos cosas: 1-2

//ya estoy en el proceso nuevo pero todavia no ejecute el programa

///cada proceso que ejecuta tiene un entorno

///1: ambiente→man environment(environ)

///2: /// tocar fd

//close(1);

//open(“./salida.txt”, O\_CREAT| O\_WRONLY| o\_TRUNC, S\_IRWXU);

execvp(myargs[0], myargs); // runs word count ///llamada a sistema myargs es TODOS los argumentos

//execvp cambia la ejecucion del programa

printf("this shouldn’t print out"); ///si funciona el execvp esta linea no existe

} else {

// parent goes down this path (main)

int rc\_wait = wait(NULL); // paddre el hijo se juntan en el wait que es el joint y siguen

printf("hello, I am parent of %d (rc\_wait:%d) (pid:%d)\n",

rc, rc\_wait, (int) getpid());

}

return 0;

}

en la terminal::

embudo:

wc p1.c>salida.txt

\*importante empezar a usar las aapi del so, para eso leer las man page

Practico de virtualizacion de cpu: podemos hacer1-10y el 12a-c (D no)

Nota(necesario p/hacer los ejercios 1-2): hay dos diferencias entre el tiempo de procesamiento

1-wall time(reloj de pared): real time

2-cpu time(system time): tiempo que pasa el proceso ejecutando en la cpu

sch: permite conectar con otras maquinas