Process API

Cat / Yukio Nozawa
RG KUMO

Notes

All codes I wrote here do not work as standalone; they require lots of Omajinai's(Magic words) we currently don't need to care of. The snippets are just to help understand how each system call works.

3 unbreakable philosophies of Unix processes

1 . A process can create its "child" processes, which are almost the exact clones of the parent

2 . Parent process can wait for child or children to finish their tasks

3. Children can be transformed into something completely different after being cloned

Philosophy 1

Use fork system call to create a child process

The created process is a clone of the parent,

but doesn't start with the main entry point

It starts from the next line of fork

The fork system call

pid_t process_id=fork();

fork returns an pid_t typed integer value to the variable called process_id

Process ID is an identifier of integer which differentiates each process

Return value	Meaning
-1	System couldn't create a child process
1 or higher	System created a process whose process ID was process_ID.You're the parent of that process.System created a process whose process ID was process_ID. You're the parent of that process.
0	You're a child process which has been created just now

Understand with a code

```
int main(){//The parent starts from here
        pid_t process_id=fork();//create a child
        //Child starts from the if below
        if(process_ID==-1){//failed to fork
                printf("Couldn't Create a process¥n");
        }
        if(process_ID==0){//Only the child executes this block
                printf("Hello, I'm a child.¥n");
}
        if(process_ID>0){//Only the parent executes this block
                printf("Hello, I'm the parent.¥n");
        }
//Parent and child execute below
return 0;//exit
}
```

Philosophy 2

Imagine you are a CPU

You're assigned 2 tasks, but you're alone

How will you do these?

Do task1 then do task2?

Do 50% of task 1, 50% of task2, then another 50% of task 1, and the last remained 50% of task 2?

Do task1 for 0.1 seconds, then do task 2 for 0.1 seconds and so on?

Do accordingly?

You don't know because you're a human!

Philosophy 2

The previous example shows that we don't know and can't control how multiple programs are executed

Talking about the first C code, We don't know which will be displayed first, parent or child

Program which returns different results is not good

What we can do is to have parent wait for a child

Philosophy 2

Use wait system call to wait for a child

Once wait is called, the program stops until the child process changes its status

If you're creating multiple children, you'd better use waitpid because it can tell which child to wait for

The wait system call

pid_t process_id=wait(NULL);

Returns the process ID of which the status has changed

The first argument can be int*(A pointer of int) and it can retrieve what status the process has changed to

Understand with a code

```
int main(){//The parent starts from here
		pid_t process_id=fork();//create a child
		//Child starts from the if below
		if(process_ID==-1){//failed to fork
				printf("Couldn't Create a process¥n");
		}
		if(process_ID==0){//Only the child executes this block
				printf("Hello, I'm a child.¥n");
}
		if(process_ID>0){//Only the parent executes this block
wait(NULL);//We created only 1 child, so the return value isn't necessary. Also, we assume that the process changed to "exit" status, so don't care about the status and set null pointer to the first argument.
				printf("Hello, I'm the parent.¥n");//This line is executed after making sure that the child exited, in other words, it finished printing "I'm a child"
		}
//Parent and child execute below
return 0;//exit
}
```

Philosophy 3

We can now create processes, but still cannot run programs that other people have developed

Fork always clones the parent which can never accomplish what we want to do

But everyone does

There should be something

Philosophy 3

A process has its own program code, data segment, heap, stack and ETC

But just dismiss such a troublesome fact for now

Think that a process is a dish with all the technical stuff on it

We should be able to wash a dish and put new ingredients on it

Use one of the exec-family system calls to do that

The exec-family system calls

replaces the entire process image to a different one

There are execl, execlp, execle, execv, execvp, and execvpe

The way to pass commandline parameters and retrieve environment variables are different from each other

exec-family system calls never returns unless an error occures

For details, go to the man page

Understand with a code

```
int main(){//The parent starts from here
		pid_t process_id=fork();//create a child
		//Child starts from the if below
		if(process_ID==-1){//failed to fork
				printf("Couldn't Create a process¥n");
		}
		if(process_ID==0){//Only the child executes this block
				printf("Hello, I'm a child. I'll check who you are.¥n");
				char* args[2];//make space to store 2 pointers of char, the number of elements should accordingly be adjusted E.G. "gcc -c test.c" requires 4
				args[0]=strdup("whoami");//create the string "whoami" and store its pointer to the first element of args. If you don't understand pointers, just think that you stored "whoami" into args[0]
				args[1]=NULL;//whoami doesn't take any arguments, so null pointer here. For those who doesn't understand pointers, it's like args[1]="" in this case, but thinking so is very dangerous
				execvp(args[0], args);//replace the process image and transform into whoami! The replaced process starts execution from the main entry point as usual.
				printf("Something happened? Error? I don't know¥n");//This shouldn't be executed because the exec isn't supposed to return
}
		if(process_ID>0){//Only the parent executes this block
wait(NULL);//We created only 1 child, so the return value isn't necessary. Also, we assume that the process changed to "exit" status, so don't care about the status and set null pointer to the first argument.
				printf("Hello, I'm the parent.¥n");//This line is executed after making sure that the child exited, in other words, it finished printing who you were
		}
//Parent and child execute below
return 0;//exit
}
```

Why so complex?

The parent/child, clone and other aspects of Unix process API are odd and sometimes difficult to understand

But the API model enables really flexible I/O redirection and signal handling

I/O redirection

Achieved by closing a certen I/O file descriptor then immediately opening another file descriptor to which you want to redirect

Example:

(at the child process before exec)

Close the standard output (file descriptor 1)

Now, 1 is blank

Open a file with fopen, pipe, socket or whatever

System searches an available file descriptor number from 0

0 is currently standard input

1 is blank

Found! Let's use this file descriptor!

Redirection completed

Signal handling

Processes can communicate each other by signals

The change of status "a child exited" is also a signal

the wait system call was waiting for a signal to come

There are lots of signals and they're fundamental to more advanced system programming