## Resumo

fazer um fork é criar dentro do processo pai um processo filho que executa uma tarefa de processamento.

<https://www.quora.com/What-does-it-mean-to-fork-a-process>

When a process uses fork(), it creates a duplicate copy of itself and this duplicates becomes the child of the process. The fork() is implemented using clone() system call in linux which returns twice from kernel.

* A non-zero value(Process ID of child) is returned to the parent.
* A value of zero is returned to the child.
* In case the child is not created successfully due to any issues like low memory, -1 is returned to the fork().

Let’s understand this with an example:

1. pid = fork();
2. if(pid < 0) {
3. //child was not created successfully
4. return 1;
5. }
6. else if(pid == 0) {
7. // This is the child process
8. // Child process code goes here
9. }
10. else {
11. // Parent process code goes here
12. }
13. printf("This is code common to parent and child")

In the example, we have assumed that exec() is not used inside the child process.

But a parent and child differs in some of the PCB(process control block) attributes. These are:

1. PID - Both child and parent have a different Process ID.
2. Pending Signals - The child doesn’t inherit Parent’s pending signals. It will be empty for the child process when created.
3. Memory Locks - The child doesn’t inherit its parent’s memory locks. Memory locks are locks which can be used to lock a memory area and then this memory area cannot be swapped to disk.
4. Record Locks - The child doesn’t inherit its parent’s record locks. Record locks are associated with a file block or an entire file.
5. Process resource utilisation and CPU time consumed is set to zero for the child.
6. The child also doesn’t inherit timers from the parent.

**But what about the child memory? Is a new address space created for a child**?

The answers in no. After the fork(), both parent and child share the memory address space of parent. In linux, these address space are divided into multiple pages. Only when the child writes to one of the parent memory pages, a duplicate of that page is created for the child. This is also known as copy on write(Copy parent pages only when the child writes to it).

Let’s understand copy on write with an example.

1. int x = 2;
2. pid = fork();
3. if(pid == 0) {
4. x = 10;
5. // child is changing the value of x or writing to a page
6. // One of the parent stack page will contain this local variable. That page will be duplicated for child and it will store the value 10 in x in duplicated page.
7. }
8. else {
9. x = 4;
10. }

**But why is copy on write necessary?**

A typical process creation takes place through fork()-exec() combination. Let’s first understand what exec() does.

Exec() group of functions replaces the child’s address space with a new program. Once exec() is called within a child, a separate address space will be created for the child which is totally different from the parent’s one.

If there was no copy on write mechanism associated with fork(), duplicate pages would have created for the child and all the data would have been copied to child’s pages. Allocating new memory and copying data is a very expensive process(takes processor’s time and other system resources). We also know that in most cases, the child is going to call exec() and that would replace the child’s memory with a new program. So the first copy which we did would have been a waste if copy on write was not there.

1. pid = fork();
2. if(pid == 0) {
3. execlp("/bin/ls","ls",NULL);
4. printf("will this line be printed"); // Think about it
5. // A new memory space will be created for the child and that memory will contain the "/bin/ls" program(text section), it's stack, data section and heap section
6. else {
7. wait(NULL);
8. // parent is waiting for the child. Once child terminates, parent will get its exit status and can then continue
9. }
10. return 1; // Both child and parent will exit with status code 1.

Usually, a process creation involves a combination of fork(), exec(), wait() and exit() calls.

**Why does parent waits for a child process?**

1. The parent can assign a task to it’s child and wait till it completes it’s task. Then it can carry some other work.
2. Once the child terminates, all the resources associated with child are freed except for the process control block. Now, the child is in zombie state. Using wait(), parent can inquire about the status of child and then ask the kernel to free the PCB. In case parent doesn’t uses wait, the child will remain in the zombie state.

**Why is exec() system call necessary?**

It’s not necessary to use exec() with fork(). If the code that the child will execute is within the program associated with parent, exec() is not needed.

But think of cases when the child has to run multiple programs. Let’s take the example of shell program. It supports multiple commands like find, mv, cp, date etc. Will be it right to include program code associated with these commands in one program or have child load these programs into the memory when required?

It all depends on your use case. You have a web server which given an input x that returns the 2^x to the clients. For each request, the web server creates a new child and asks it to compute. Will you write a separate program to calculate this and use exec()? Or you will just write computation code inside the parent program?

If you find my explanation useful, you can also watch my videos on process creation using fork(), exec(), wait() and exit().