

Join the Stack Overflow Community

Stack Overflow is a community of 6.2 million programmers, just like you, helping each other.
Join them; it only takes a minute:


[Sign up](#)

Differences between exec and fork

```

36  if (dev.isBored() || job.sucks()) {
37      searchJobs({flexibleHours: true, companyCulture: 100});
38  }
39  A career site that's by developers, for developers.

```



[Get started](#)

What are the differences between `fork` and `exec` ?

c `exec` `fork`

edited May 19 '14 at 9:04



David Guyon

1,137 11 30

asked Oct 31 '09 at 3:47



Sashi

838 4 10 12

73 ...and we'd like google to link to here next time someone googles this... so its a valid question. – [Polaris878](#) Oct 31 '09 at 4:29

1 Meh. Google has this covered. Why clog it up with more results just so we can say that there's another SO answer that ranks? – [Justin Niessner](#) Oct 31 '09 at 4:38

3 A good, detailed summary of fork, exec, and other process control functions is at yolinux.com/TUTORIALS/ForkExecProcesses.html – [Jonathan Fingland](#) Oct 31 '09 at 4:38

6 @Justin, because we want SO to become *the* place to go for programming questions. – [paxdiablo](#) Jan 27 '10 at 21:35

3 @Polaris878: oh, it does now! :D – [Janusz Lenar](#) Jan 31 '12 at 8:46

7 Answers

The use of `fork` and `exec` exemplifies the spirit of UNIX in that it provides a very simple way to start new processes.

The `fork` call basically makes a duplicate of the current process, identical in *almost* every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

The new process (child) gets a different process ID (PID) and has the the PID of the old process (parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they can tell which is which by the return code of `fork` - the child gets 0, the parent gets the PID of the child. This is all, of course, assuming the `fork` call works - if not, no child is created and the parent gets an error code.

The `exec` call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point.

So, `fork` and `exec` are often used in sequence to get a new program running as a child of a current process. Shells typically do this whenever you try to run a program like `find` - the shell forks, then the child loads the `find` program into memory, setting up all command line arguments, standard I/O and so forth.

But they're not required to be used together. It's perfectly acceptable for a program to `fork` itself without `exec` ing if, for example, the program contains both parent and child code (you need to be careful what you do, each implementation may have restrictions). This was used quite a lot (and still is) for daemons which simply listen on a TCP port and fork a copy of themselves to process a specific request while the parent goes back to listening.

Similarly, programs that know they're finished and just want to run another program don't need to `fork`, `exec` and then `wait` for the child. They can just load the child directly into their

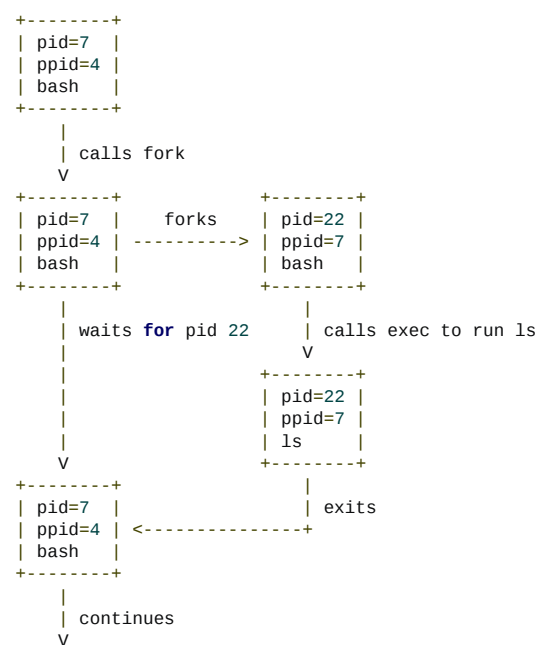
process space.

Some UNIX implementations have an optimized `fork` which uses what they call copy-on-write. This is a trick to delay the copying of the process space in `fork` until the program attempts to change something in that space. This is useful for those programs using only `fork` and not `exec` in that they don't have to copy an entire process space.

If the `exec` is called following `fork` (and this is what happens mostly), that causes a write to the process space and it is then copied for the child process.

Note that there is a whole family of `exec` calls (`execl`, `execle`, `execve` and so on) but `exec` in context here means any of them.

The following diagram illustrates the typical `fork/exec` operation where the `bash` shell is used to list a directory with the `ls` command:



edited Nov 1 '09 at 12:16

answered Oct 31 '09 at 4:31



paxdiablo

491k

118

973

1422

```
36 if (dev.isBored() || job.sucks()) {
37     searchJobs({flexibleHours: true, companyCulture: 100});
38 }
39 A career site that's by developers, for developers.
```

[Get started](#)

`fork()` splits the current process into two processes. Or in other words, your nice linear easy to think of program suddenly becomes two separate programs running one piece of code:

```
int pid = fork();

if (pid == 0)
{
    printf("I'm the child");
}
else
{
    printf("I'm the parent, my child is %i", pid);
    // here we can kill the child, but that's not very parently of us
}
```

This can kind of blow your mind. Now you have one piece of code with pretty much identical state being executed by two processes. The child process inherits all the code and memory of the process that just created it, including starting from where the "`fork()`" call just left off. The only difference is the `fork()` code return to tell you if you are the parent or the child. If you are the parent, the return is the id of the child.

`exec` is a bit easier to grasp, you just tell `exec` to execute a process using the target executable and you don't have two processes running the same code or inheriting the same state. Like @Steve Hawkins says, `exec` can be used after you `fork` to execute in the current process the target executable.

edited Oct 31 '09 at 13:26

answered Oct 31 '09 at 3:51



Doug T.

41k

16

93

165

there's also the condition when `pid < 0` and the `fork()` call failed – [Jonathan Finland](#) Oct 31 '09 at 4:37

- 3 That doesn't blow my mind at all :-). One piece of code being executed by two processes happens every time a shared library or DLL is being used. – [paxdiablo](#) Oct 31 '09 at 5:01

I think some concepts from "[Advanced Unix Programming](#)" by [Marc Rochkind](#) were helpful in understanding the different roles of `fork()` / `exec()`, especially for someone used to the Windows `CreateProcess()` model:

A *program* is a collection of instructions and data that is kept in a regular file on disk. (from 1.1.2 Programs, Processes, and Threads)

In order to run a program, the kernel is first asked to create a new *process*, which is an environment in which a program executes. (also from 1.1.2 Programs, Processes, and Threads)

It's impossible to understand the `exec` or `fork` system calls without fully understanding the distinction between a process and a program. If these terms are new to you, you may want to go back and review Section 1.1.2. If you're ready to proceed now, we'll summarize the distinction in one sentence: A process is an execution environment that consists of instruction, user-data, and system-data segments, as well as lots of other resources acquired at runtime, whereas a program is a file containing instructions and data that are used to initialize the instruction and user-data segments of a process. (from 5.3 `exec` System Calls)

Once you understand the distinction between a program and a process, the behavior of `fork()` and `exec()` function can be summarized as:

- `fork()` creates a duplicate of the current process
- `exec()` replaces the program in the current process with another program

(this is essentially a simplified 'for dummies' version of [paxdiablo's much more detailed answer](#))

answered Apr 7 '10 at 19:48



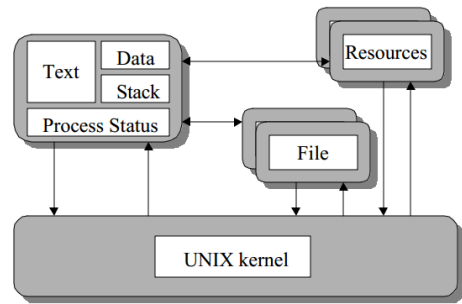
[Michael Burr](#)

240k 31 357 577

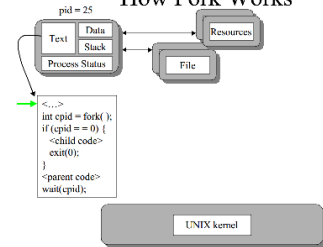
Warren Young gave apt description.

fork creates a copy of a calling process. generally follows the structure

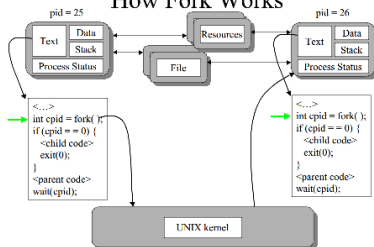
A Unix Process



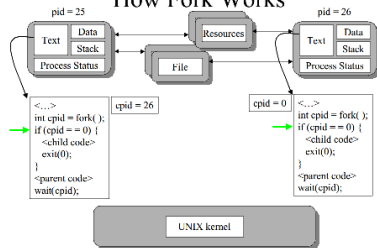
How Fork Works



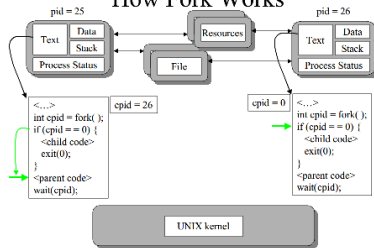
How Fork Works



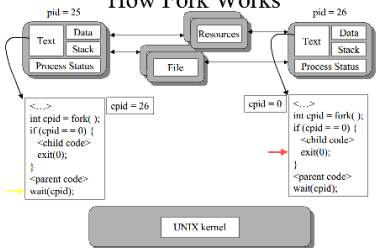
How Fork Works



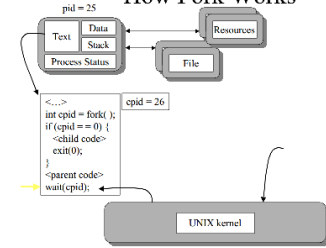
How Fork Works



How Fork Works



How Fork Works



```
int cpid = fork( );
```

```
if (cpid == 0)
```

```
{
```

```
  //child code
```

```
  exit(0);
```

```
}
```

```
//parent code
```

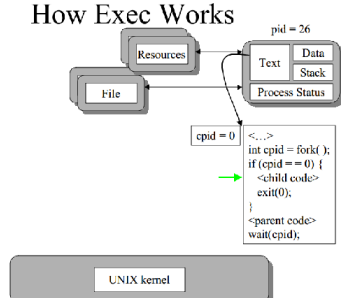
```
wait(cpid);
```

```
// end
```

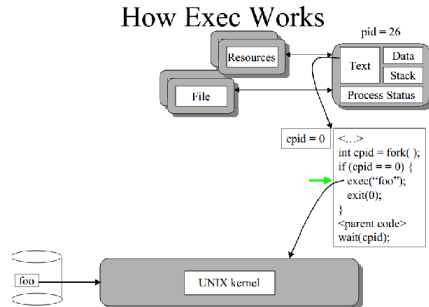
(for child process text(code),data,stack is same as calling process) child process executes code in if block.

EXEC replaces the current process with new process's code,data,stack. generally follows the structure

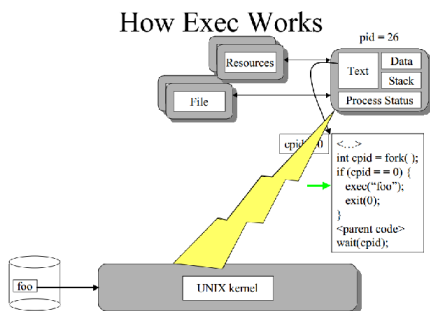
How Exec Works



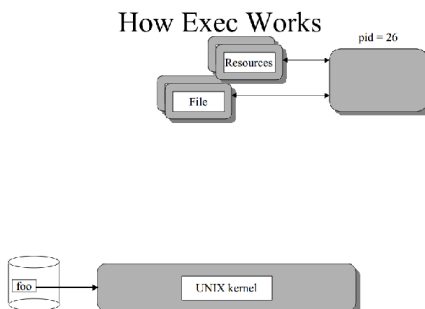
How Exec Works



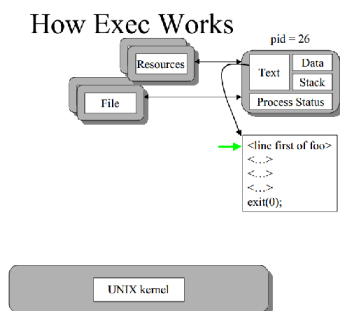
How Exec Works



How Exec Works



How Exec Works



```
int cpid = fork( );

if (cpid == 0)
{
    //child code

    exec(foo);

    exit(0);
}

//parent code

wait(cpid);

// end
```

(after `exec` call unix kernel clears the child process text,data,stack and fills with foo process related text/data) thus child process is with different code (foo's code {not same as parent})

edited Jun 2 '13 at 1:41

answered May 22 '13 at 3:23



Sandesh Kobal
308 3 5

They are use together to create a new child process. First, calling `fork` creates a copy of the current process (the child process). Then, `exec` is called from within the child process to "replace" the copy of the parent process with the new process.

The process goes something like this:

```
child = fork(); //Fork returns a PID for the parent process, or 0 for the child,
or -1 for Fail

if (child < 0) {
    std::cout << "Failed to fork GUI process...Exiting" << std::endl;
    exit (-1);
}
```

```

} else if (child == 0) { // This is the Child Process
    // Call one of the "exec" functions to create the child process
    execvp (argv[0], const_cast<char**>(argv));
} else { // This is the Parent Process
    //Continue executing parent process
}

```

edited Oct 31 '09 at 4:49

answered Oct 31 '09 at 3:54



Josh Lee

83.6k 16 178 207



Steve Hawkins

773 1 10 21

IIUC, that may not work from processes with GUI? – [moala](#) Jan 5 '12 at 12:54

In 7th line it is mentioned that exec() function creates the child process.. Is it really so because fork() has already created the child process and exec() call just replaces the program of the new process just created – [cbinder](#) May 25 '14 at 2:54

fork() creates a copy of the current process, with execution in the new child starting from just after the fork() call. After the fork(), they're identical, except for the return value of the fork() function. (RTFM for more details.) The two processes can then diverge still further, with one unable to interfere with the other, except possibly through any shared file handles.

exec() replaces the current process with a new one. It has nothing to do with fork(), except that an exec() often follows fork() when what's wanted is to launch a different child process, rather than replace the current one.

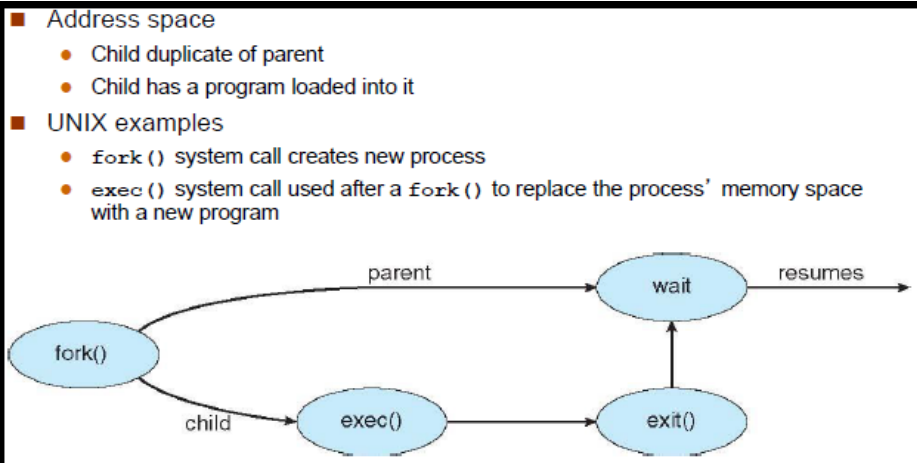
edited Jul 8 '12 at 9:04

answered Oct 31 '09 at 3:54



Warren Young

24.3k 4 58 80



fork() :

It creates a copy of running process. The running process is called **parent process** & newly created process is called **child process**. The way to differentiate the two is by looking at the returned value:

1. fork() returns the process identifier (pid) of the child process in the parent
2. fork() returns 0 in the child.

exec() :

It initiates a new process within a process. It loads a new program into the current process, replacing the existing one.

fork() + exec() :

When launching a new program is to firstly fork() , creating a new process, and then exec() (i.e. load into memory and execute) the program binary it is supposed to run.

```

int main( void )
{
    int pid = fork();
    if ( pid == 0 )
    {
        execvp( "find", argv );
    }

    //Put the parent to sleep for 2 sec, let the child finished executing
    wait( 2 );

    return 0;
}

```

edited Nov 27 '15 at 7:44

answered Nov 27 '15 at 7:32

