fork (system call)

In computing, particularly in the context of the Unix operating system and its workalikes, **fork** is an operation whereby a process creates a copy of itself. It is an interface which is required for compliance with the POSIX and Single UNIX Specification standards. It is usually implemented as a C standard library wrapper to the fork, clone, or other system calls of the kernel. Fork is the primary method of process creation on Unix-like operating systems.

Overview

In multitasking operating systems, processes (running programs) need a way to create new processes, e.g. to run other programs. Fork and its variants are typically the only way of doing so in Unix-like systems. For a process to start the execution of a different program, it first forks to create a copy of itself. Then, the copy, called the "child process", calls the exec system call to overlay itself with the other program: it ceases execution of its former program in favor of the other.

The fork operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process. In modern UNIX variants that follow the virtual memory model from SunOS-4.0, copy-on-write semantics are implemented and the physical memory need not be actually copied. Instead, virtual memory pages in both processes may refer to the same pages of physical memory until one of them reads from such a page: then it is copied. This optimization is important in the common case where fork is used in conjunction with exec to execute a new program: typically, the child process performs only a small set of actions before it ceases execution of its program in favour of the program to be started, and it requires very few, if any, of its parent's data structures.

When a process calls fork, it is deemed the parent process and the newly created process is its child. After the fork, both processes not only run the same program, but they resume execution as though both had called the system call. They can then inspect the call's return value to determine their status, child or parent, and act accordingly.

History

One of the earliest references to a fork concept appeared in *A Multiprocessor System Design* by Melvin Conway, published in 1962.[1] Conway's paper motivated the implementation by L. Peter Deutsch of fork in the GENIE time-sharing system, where the concept was borrowed by Ken Thompson for its earliest appearance[2] in Research Unix.[3][4] Fork later became a standard interface in POSIX.[5]

Communication

The child process starts off with a copy of its parent's file descriptors.[5] For interprocess communication, the parent process will often create one or several pipes, and then after forking the processes will close the ends of the pipes that they do not need.[6]

Variants

Vfork

Vfork is a variant of fork with the same calling convention and much the same semantics, but only to be used in restricted situations. It originated in the 3BSD version of Unix,[7][8][9] the first Unix to support virtual memory. It was standardized by POSIX, which permitted vfork to have exactly the same behavior as fork, but was marked obsolescent in the 2004 edition[10] and was replaced by posix_spawn() (which is typically implemented via vfork) in subsequent editions.

When a vfork system call is issued, the parent process will be suspended until the child process has either completed execution or been replaced with a new executable image via one of the "exec" family of system calls. The child borrows the memory management unit setup from the parent and memory pages are shared among the parent and child process with no copying done, and in particular with no copy-on-write semantics;[10] hence, if the child process makes a modification in any of the shared pages, no new page will be created and the modified pages are visible to the parent process too. Since there is absolutely no page copying involved (consuming additional memory), this technique is an optimization over plain fork in full-copy environments when used with exec. In POSIX, using vfork for any purpose except as a prelude to an immediate call to a function from the exec family (and a select few other operations) gives rise to undefined behavior.[10] As with vfork, the child borrows data structures rather than copying them. vfork is still faster than a fork that uses copy on write semantics.

System V did not support this function call before System VR4 was introduced, because the memory sharing that it causes is error-prone:

> *Vfork* does not copy page tables so it is faster than the System V *fork* implementation. But the child process executes in the same physical address space as the parent process (until an *exec* or *exit*) and can thus overwrite the parent's data and stack. A dangerous situation could arise if a programmer uses *vfork* incorrectly, so the onus for calling *vfork* lies with the programmer. The difference between the System V approach and the BSD approach is philosophical: Should the kernel hide idiosyncrasies of its implementation from users, or should it allow sophisticated users the opportunity to take advantage of the implementation to do a logical function more efficiently?
>
> —Maurice J. Bach[11]

Similarly, the Linux man page for vfork strongly discourages its use:[7]

> It is rather unfortunate that Linux revived this specter from the past. The BSD man page states: "This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2)."

Other problems with `vfork` include deadlocks that might occur in multithreaded programs due to interactions with dynamic linking.[12] As a replacement for the `vfork` interface, POSIX introduced the `posix_spawn` family of functions that combine the actions of fork and exec. These functions may be implemented as library routines in terms of `fork`, as is done in Linux,[12] or in terms of `vfork` for better performance, as is done in Solaris,[12][13] but the POSIX specification notes that they were "designed as kernel operations", especially for operating systems running on constrained hardware and real-time systems.[14]

While the 4.4BSD implementation got rid of the vfork implementation, causing vfork to have the same behavior as fork, it was later reinstated in the NetBSD operating system for performance reasons.[8]

Some embedded operating systems such as uClinux omit fork and only implement vfork, because they need to operate on devices where copy-on-write is impossible to implement due to lack of a memory management unit.

Rfork

The Plan 9 operating system, created by the designers of Unix, includes fork but also a variant called "rfork" that permits fine-grained sharing of resources between parent and child processes, including the address space (except for a stack segment, which is unique to each process), environment variables and the filesystem namespace;[15] this makes it a unified interface for the creation of both processes and threads within them.[16] Both FreeBSD[17] and IRIX adopted the rfork system call from Plan 9, the latter renaming it "sproc".[18]

Clone

`clone` is a system call in the Linux kernel that creates a child process that may share parts of its execution context with the parent. Like FreeBSD's rfork and IRIX's sproc, Linux's clone was inspired by Plan 9's rfork and can be used to implement threads (though application programmers will typically use a higher-level interface such as pthreads, implemented on top of clone). The "separate stacks" feature from Plan 9 and IRIX has been omitted because (according to Linus Torvalds) it causes too much overhead.[18]

Forking in other operating systems

In the original design of the VMS operating system (1977), a copy operation with subsequent mutation of the content of a few specific addresses for the new process as in forking was considered risky. Errors in the current process state may be copied to a child process. Here, the metaphor of process spawning is used: each component of the memory layout of the new process is newly constructed from scratch. The spawn metaphor was later adopted in Microsoft operating systems (1993).

The POSIX-compatibility component of VM/CMS (OpenExtensions) provides a very limited implementation of fork, in which the parent is suspended while the child executes, and the child and the parent share the same address space.[19] This is essentially a *vfork* labelled as a *fork*. (This applies to the CMS guest operating system only; other VM guest operating systems, such as Linux, provide standard fork functionality.)

Application usage

The following variant of the "Hello, World!" program demonstrates the mechanics of the `fork` system call in the C programming language. The program forks into two processes, each deciding what functionality they perform based on the return value of the fork system call. Boilerplate code such as header inclusions has been omitted.

```c
int main(void)
{
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

What follows is a dissection of this program.

```c
   pid_t pid = fork();
```

The first statement in `main` calls the `fork` system call to split execution into two processes. The return value of `fork` is recorded in a variable of type `pid_t`, which is the POSIX type for process identifiers (PIDs).

```c
    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
```

Minus one indicates an error in `fork`: no new process was created, so an error message is printed.

If `fork` was successful, then there are now two processes, both executing the `main` function from the point where `fork` has returned. To make the processes perform different tasks, the program must branch on the return value of `fork` to determine whether it is executing as the *child* process or the *parent* process.

```c
    else if (pid == 0) {
       printf("Hello from the child process!\n");
       _exit(EXIT_SUCCESS);
    }
```

In the child process, the return value appears as zero (which is an invalid process identifier). The child process prints the desired greeting message, then exits. (For technical reasons, the POSIX `_exit` function must be used here instead of the C standard `exit` function.)

```c
    else {
       int status;
       (void)waitpid(pid, &status, 0);
    }
```

The other process, the parent, receives from `fork` the process identifier of the child, which is always a positive number. The parent process passes this identifier to the `waitpid` system call to suspend execution until the child has exited. When this has happened, the parent resumes execution and exits by means of the `return` statement.

See also

- Fork bomb
- Fork–exec
- exit (system call)
- spawn (computing)
- wait (system call)

References

1. Nyman, Linus (25 August 2016). "Notes on the History of Fork and Join". *IEEE Annals of the History of Computing*. **38** (3): 84–87. doi:10.1109/MAHC.2016.34 (https://doi.org/10.1109%2FMAHC.2016.34).
2. "s3.s from Research UNIX" (https://github.com/dspinellis/unix-history-repo/blob/Research-PDP7-Snapshot-Development/s3.s#L43-L70). *GitHub*. 1970.
3. Ken Thompson and Dennis Ritchie (3 November 1971). "SYS FORK (II)" (https://www.bell-labs.com/usr/dmr/www/pdfs/man21.pdf) (PDF). *UNIX Programmer's Manual*. Bell Laboratories.
4. Ritchie, Dennis M.; Thompson, Ken (July 1978). "The UNIX Time-Sharing System" (https://www.bell-labs.com/usr/dmr/www/cacm.pdf) (PDF). *Bell System Tech. J.* **57** (6). AT&T: 1905–1929. doi:10.1002/j.1538-7305.1978.tb02136.x (https://doi.org/10.1002%2Fj.1538-7305.1978.tb02136.x). Retrieved 22 April 2014.
5. `fork (https://www.opengroup.org/onlinepubs/9699919799/functions/fork.html)` – System Interfaces Reference, The Single UNIX Specification, Version 4 from The Open Group
6. `pipe (https://www.opengroup.org/onlinepubs/9699919799/functions/pipe.html)` – System Interfaces Reference, The Single UNIX Specification, Version 4 from The Open Group
7. `vfork(2) (https://manned.org/vfork.2)` – Linux Programmer's Manual – System Calls
8. "NetBSD Documentation: Why implement traditional vfork()" (http://www.netbsd.org/docs/kernel/vfork.html). *NetBSD Project*. Retrieved 16 October 2013.

9. "vfork(2)". *UNIX Programmer's Manual, Virtual VAX-11 Version*. University of California, Berkeley. December 1979.
10. vfork (https://www.opengroup.org/onlinepubs/009695399/functions/vfork.html) – System Interfaces Reference, The Single UNIX Specification, Version 3 from The Open Group
11. Bach, Maurice J. (1986). *The Design of The UNIX Operating System*. Prentice–Hall. pp. 291–292. Bibcode:1986duos.book.....B (https://ui.adsabs.harvard.edu/abs/1986duos.book.....B).
12. Nakhimovsky, Greg (May 2006). "Minimizing Memory Usage for Creating Application Subprocesses" (https://web.archive.org/web/20190922113430/https://www.oracle.com/technetwork/server-storage/solaris10/subprocess-136439.html). *Oracle Technology Network*. Oracle Corporation. Archived from the original (http://www.oracle.com/technetwork/server-storage/solaris10/subprocess-136439.html) on Sep 22, 2019.
13. The OpenSolaris posix_spawn() implementation
14. posix_spawn (https://www.opengroup.org/onlinepubs/9699919799/functions/posix_spawn.html) – System Interfaces Reference, The Single UNIX Specification, Version 4 from The Open Group
15. fork(2) (https://9p.io/magic/man2html/2/fork) – Plan 9 Programmer's Manual, Volume 1
16. intro(2) (https://9p.io/magic/man2html/2/intro) – Plan 9 Programmer's Manual, Volume 1
17. rfork(2) (https://www.freebsd.org/cgi/man.cgi?query=rfork&sektion=2) – FreeBSD System Calls Manual
18. Torvalds, Linus (1999). "The Linux edge" (https://archive.org/details/isbn_9781565925823). *Open Sources: Voices from the Open Source Revolution*. O'Reilly. ISBN 978-1-56592-582-3.
19. "z/VM > z/VM 6.2.0 > Application Programming > z/VM V6R2 OpenExtensions POSIX Conformance Document > POSIX.1 Conformance Document > Section 3. Process Primitives > 3.1 Process Creation and Execution > 3.1.1 Process Creation" (http://www-01.ibm.com/support/knowledgecenter/SSB27U_6.2.0/com.ibm.zvm.v620.dmsp0/hcsp0c0022.htm%23wq47?lang=en). IBM. Retrieved April 21, 2015.

-