

Fork (operating system)

From Wikipedia, the free encyclopedia

In computing, when a process **forks**, it creates a copy of itself. More generally, a fork in a multithreading environment means that a thread of execution is duplicated, creating a child thread from the parent thread.

Under Unix and Unix-like operating systems, the parent and the child processes can tell each other apart by examining the return value of the `fork()` system call. In the child process, the return value of `fork()` is 0, whereas the return value in the parent process is the PID of the newly created child process.

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, though if copy-on-write semantics are implemented actual physical memory may not be assigned (i.e., both processes may share the same physical memory segments for a while). Both the parent and child processes possess the same code segments, but execute independently of each other.

Contents

- 1 Importance of forking in Unix
- 2 Process Address Space
- 3 Fork and page sharing
- 4 Vfork and page sharing
- 5 MMUless systems
- 6 Forking in other operating systems
- 7 Application usage
 - 7.1 Example in C
 - 7.2 Example in Perl
 - 7.3 Example in Python
- 8 See also
- 9 References
- 10 External links

Importance of forking in Unix

Forking is an important part of Unix, critical to the support of its design philosophy, which encourages the development of filters. In Unix, a filter is a (usually small) program that reads its input from `stdin`, and writes its output to `stdout`. A pipeline of these commands can be strung together by a shell to create new commands.

For example, one can string together the output of the `find(1)` command and the input of the `wc(1)` command to create a new command that will print a count of files ending in ".cpp" found in the current directory and any subdirectories, as follows:

```
$ find . -name "*.cpp" -print | wc -l
```

In order to accomplish this, the shell forks itself, and uses pipes, a form of interprocess communication, to tie the output of the `find` command to the input of the `wc` command. Two child processes are created, one for each command (`find` and `wc`). These child processes are overlaid with the code associated with the programs they are intended to execute, using the `exec(3)` family of system calls (in the above example, `find` will overlay the first child process, and `wc` will overlay the second child process, and the shell will use pipes to tie the output of `find` with the input of `wc`).

More generally, forking is also performed by the shell each time a user issues a command. A child process is created by forking the shell, and the child process is overlaid, once again by `exec`, with the code associated with the program to be executed.

Process Address Space

Whenever an executable file is executed, it becomes a process. An executable file contains binary code grouped into a number of blocks called segments. Each segment is used for storing a particular type of data. A few segment names of a typical ELF executable file are listed below.

- `text` – Segment containing executable code
- `.bss` – Segment containing data initialized to zero
- `data` – Segment containing initialized data
- `symtab` – Segment containing the program symbols (e.g., function name, variable names, etc.)
- `interp` – Segment containing the name of the interpreter to be used

The `readelf` command can provide further details of the ELF file. When such a file is loaded in the memory for execution, the segments are loaded in memory. It is not necessary for the entire executable to be loaded in contiguous memory locations. Memory is divided into equal sized partitions called pages (typically 4KB). Hence when the executable is loaded in the memory, different parts of the executable are placed in different pages (which might not be contiguous). Consider an ELF executable file of size 10K. If the page size supported by the OS is 4K, then the file will be split

into three pieces (also called frames) of size 4K, 4K, and 2K respectively. These three frames will be accommodated in any three free pages in memory.

Fork and page sharing

When a `fork()` system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the OS for the child process. But this is not needed in certain cases. Consider the case when a child executes an "`exec`" system call (which is used to execute any executable file from within a C program) or exits very soon after the `fork()`. When the child is needed just to execute a command for the parent process, there is no need for copying the parent process' pages, since `exec` replaces the address space of the process which invoked it with the command to be executed.

In such cases, a technique called copy-on-write (COW) is used. With this technique, when a fork occurs, the parent process's pages are not copied for the child process. Instead, the pages are shared between the child and the parent process. Whenever a process (parent or child) modifies a page, a separate copy of that particular page alone is made for that process (parent or child) which performed the modification. This process will then use the newly copied page rather than the shared one in all future references. The other process (the one which did not modify the shared page) continues to use the original copy of the page (which is now no longer shared). This technique is called copy-on-write since the page is copied when some process writes to it.

Vfork and page sharing

`vfork` is another UNIX system call used to create a new process. 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 `execve()` family of system calls. Even in `vfork`, the pages are shared among the parent and child process. But `vfork` does not mandate copy-on-write. 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 highly efficient when a process needs to execute a blocking command using the child process.

On some implementations, `vfork()` is the same as `fork()`. ^[1]

The `vfork()` function differs from `fork()` only in that the child

process can share code and data with the calling process (parent process). This speeds cloning activity significantly, at a risk to the integrity of the parent process if `vfork()` is misused.

The use of `vfork()` for any purpose except as a prelude to an immediate call to a function from the `exec` family, or to `_exit()`, is not advised. In particular the Linux man page for `vfork` strongly discourages its use: [2]

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)`."

The `vfork()` function can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to call `exec`, the data space copied from the parent to the child by `fork()` is not used. This is particularly inefficient in a paged environment, making `vfork()` particularly useful. Depending upon the size of the parent's data space, `vfork()` can give a significant performance improvement over `fork()`.

The `vfork()` function can normally be used just like `fork()`. It does not work, however, to return while running in the child's context from the caller of `vfork()` since the eventual return from `vfork()` would then return to a no longer existent stack frame. Care must also be taken to call `_exit()` rather than `exit()` if `exec` cannot be called, since `exit()` flushes and closes standard I/O channels, thereby damaging the parent process's standard I/O data structures. (Even with `fork()`, it is still incorrect to call `exit()`, since buffered data would then be flushed twice.)

If signal handlers are invoked in the child process after `vfork()`, they must follow the same rules as other code in the child process. [3][1]

MMUless systems

On several embedded devices, there is no Memory Management Unit, which is a requirement for implementing the copy-on-write semantics prescribed by `fork()`. If the system has some other mechanism for per-process address spaces, such as a segment register, copying the entire process memory to the new process achieves the desired effect, however this is a costly operation and most likely unneeded given that the new process almost immediately replaces the process image in most instances.

If all processes share a single address space, then the only way `fork()` could be implemented would be to swap the memory pages along with the rest of the task context switch. Rather than doing that, embedded operating systems such as uClinux usually omit `fork()` and only implement `vfork()`; part of the work porting to such a platform involves rewriting code to use the latter.

Forking in other operating systems

The fork mechanism (1969) in Unix and Linux maintains implicit assumptions on the underlying hardware: linear memory and a paging mechanism that enable an efficient, memory copy operation of a contiguous address range. In the original design of the VMS (now OpenVMS) 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. From a software-engineering viewpoint this latter approach would be considered more clean and safe, but the fork mechanism is still predominant due to its efficiency. The spawn (computing) metaphor was later adopted in Microsoft operating systems (1993).

Application usage

The `fork()` system call takes no argument and returns a process ID, which is usually an integer value. The returned process ID is of the type `pid_t`, which has been defined in the header file, `sys/types.h`.

The purpose of `fork()` system call is to create a new process, which becomes the child process of caller, after which both, the parent and child processes, will execute the code following the `fork()` system call. Hence, it is important to distinguish between parent and child process. This can be done by testing the return value of `fork()` system call.

- If `fork()` returns a negative value, it indicates that the creation of the process was unsuccessful.
- `fork()` returns a zero to the newly created child process.
- `fork()` returns a positive value, the process ID of the child process, to the parent.^[4]

Example in C

```
#include <stdio.h>    /* printf, stderr, fprintf */
```

```
#include <sys/types.h> /* pid_t */
#include <unistd.h> /* _exit, fork */
#include <stdlib.h> /* exit */
#include <errno.h> /* errno */

int main(void)
{
    int pid_t,pid;

    /* Output from both the child and the parent process
     * will be written to the standard output,
     * as they both run at the same time.
     */
    pid = fork();
    if (pid == -1)
    {
        /* Error:
         * When fork() returns -1, an error happened
         * (for example, number of processes reached the limit).
         */
        fprintf(stderr, "can't fork, error %d\n", errno);
        exit(EXIT_FAILURE);
    }

    if (pid == 0)
    {
        /* Child process:
         * When fork() returns 0, we are in
         * the child process.
         */
        int j;
        for (j = 0; j < 10; j++)
        {
            printf("child: %d\n", j);
            sleep(1);
        }
        _exit(0); /* Note that we do not use exit() */
    }
    else
    {
        /* When fork() returns a positive number, we are in the pa
         * (the fork return value is the PID of the newly created
         * Again we count up to ten.
         */
        int i;
        for (i = 0; i < 10; i++)
        {
            printf("parent: %d\n", i);
        }
    }
}
```

```
        sleep(1);
    }
    exit(0);
}
return 0;
}
```

Example in Perl

```
#!/usr/bin/env perl -w
use strict;

if (fork) { # Parent process
    foreach my $i (0 .. 9) {
        print "Parent: $i\n";
        sleep 1;
    }
}
else { # Child process
    foreach my $i (0 .. 9) {
        print "Child: $i\n";
        sleep 1;
    }
    exit(0); # End the forked process
}

exit(0); # End the parent process.
```

Example in Python

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os, time

def createDaemon():
    """
        This function create a service/Daemon that will execute a
    """

    try:
        # Store the Fork PID
        pid = os.fork()

        if pid > 0:
```

```
        print 'PID: %d' % pid
        os._exit(0)

except OSError, error:
    print 'Unable to fork. Error: %d (%s)' % (error.errno, error)
    os._exit(1)

doTask()

def doTask():
    """
        This function create a task that will be a daemon
    """

    # Open the file in write mode
    file = open('/tmp/tarefa.log', 'w')

    while True:
        print >> file, time.ctime()
        file.flush()
        time.sleep(2)

    # Close the file
    file.close()

if __name__ == '__main__':

    # Create the Daemon
    createDaemon()
```

See also

- Child process
- Parent process
- Fork bomb
- Fork-exec
- Exec
- Exit
- Wait

References

- ^a ^b The Open Group Base Specifications Issue 6
(<http://www.opengroup.org/onlinepubs/009695399/utilities/pax.html>) -
POSIX.1-2001 (*IEEE Std 1003.1*) Copyright © 2001-2004 The IEEE and The
Open Group (<http://www.opengroup.org/>)

2. ^ VFORK (<http://www.linuxmanpages.com/man2/vfork.2.php>)
3. ^ UNIX Specification Version 2, 1997 <http://www.opengroup.org/pubs/online/7908799/xsh/vfork.html>
4. ^ "CSL Website" (<http://www.csl.mtu.edu/cs4411/www/NOTES/process/fork/create.html>) . <http://www.csl.mtu.edu/cs4411/www/NOTES/process/fork/create.html>.

External links

- [fork](http://www.opengroup.org/onlinepubs/9699919799/functions/fork.html) (<http://www.opengroup.org/onlinepubs/9699919799/functions/fork.html>) : create a new process - System Interfaces Reference, The Single UNIX® Specification, Issue 7 from The Open Group
- [Lightwolf](http://lightwolf.sourceforge.net) (<http://lightwolf.sourceforge.net>) , A library that implements thread forking for Java
- [NetBSD: Why implement traditional vfork\(\)](http://www.netbsd.org/docs/kernel/vfork.html) (<http://www.netbsd.org/docs/kernel/vfork.html>)

Retrieved from "[http://en.wikipedia.org/w/index.php?title=Fork_\(operating_system\)&oldid=476681335](http://en.wikipedia.org/w/index.php?title=Fork_(operating_system)&oldid=476681335)"
Categories: [Process \(computing\)](#) | [C POSIX library](#)

-
- This page was last modified on 13 February 2012 at 17:44.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.