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

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 uninitialized data
- 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 execv 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 is done, 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 shared version of the page. 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 systems, `vfork()` is the same as `fork()`.

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: [1]

> 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.[2]

Application usage

Example in C

```c
#include <stdio.h>   /* printf, stderr, fprintf */
#include <unistd.h>  /* _exit, fork */
#include <stdlib.h>  /* exit */
#include <errno.h>   /* errno */

int main(void)
{
   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 == 0)
   {
      /* Child process:
       * When fork() returns 0, we are in
       * the child process.
       * Here we count up to ten, one each second.
       */
      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 if (pid > 0)
   {
      /* Parent process:
       * When fork() returns a positive number, we are in the parent process
       * (the fork return value is the PID of the newly-created child process).
       * Again we count up to ten.
       */
      int i;
      for (i = 0; i < 10; i++)
      {
         printf("parent: %d\n", i);
         sleep(1);
      }
      exit(0);
   }
   else
   {
      /* Error:
       * When fork() returns a negative number, an error happened
       * (for example, number of processes reached the limit).
       */
      fprintf(stderr, "can't fork, error %d\n", errno);
      exit(1);
   }
}
```

Example in Perl

```perl
#!/usr/bin/perl

$pid = fork();    #Declare fork
if ($pid == 0) {  #Jump into the Child process
    for ($j = 0; $j < 10; $j++) {
        print "child: $j\n";
        sleep 1;
    }
    exit(0);  #Exit the fork [child process]
} elsif ($pid > 0) {
    for ($i = 0; $i < 10; $i++) {
        print "parent: $i\n";
        sleep 1;
    }
    exit(0);  #Exit parent
}
```

See also

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

References

1. ^ VFORK (http://www.linuxmanpages.com/man2/vfork.2.php)
2. ^ UNIX Specification Version 2, 1997 http://www.opengroup.org/pubs/online/7908799/xsh/vfork.html

External links

- `fork (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

Retrieved from "http://en.wikipedia.org/wiki/Fork_(operating_system)"
Categories: Process (computing) | POSIX standards

non-profit organization.