Which file in kernel specifies fork(), vfork()… to use sys_clone() system call

unix.stackexchange.com/questions/87551/which-file-in-kernel-specifies-fork-vfork-to-use-sys-clone-system-call

When ltrace is used for tracing the system calls, I could see that fork() uses sys_clone() rather than sys_fork(). But I couldnt find the linux source where it is defined.

My program is

```
#include<stdio.h>
main()
{
        int pid,i=0,j=0;
        pid=fork();
        if(pid==0)
                printf("\nI am child\n");
        else
                printf("\nI am parent\n");

}
```

And ltrace output is

```
SYS_brk(NULL)
= 0x019d0000
SYS_access("/etc/ld.so.nohwcap", 00)
= -2
SYS_mmap(0, 8192, 3, 34, 0xffffffff)
= 0x7fe3cf84f000
SYS_access("/etc/ld.so.preload", 04)
= -2
SYS_open("/etc/ld.so.cache", 0, 01)
= 3
SYS_fstat(3, 0x7fff47007890)
= 0
SYS_mmap(0, 103967, 1, 2, 3)
= 0x7fe3cf835000
SYS_close(3)
= 0
SYS_access("/etc/ld.so.nohwcap", 00)
= -2
SYS_open("/lib/x86_64-linux-gnu/libc.so.6", 0, 00)
= 3
SYS_read(3, "\177ELF\002\001\001", 832)
= 832
SYS_fstat(3, 0x7fff470078e0)
= 0
SYS_mmap(0, 0x389858, 5, 2050, 3)
= 0x7fe3cf2a8000
SYS_mprotect(0x7fe3cf428000, 2097152, 0)
= 0
SYS_mmap(0x7fe3cf628000, 20480, 3, 2066, 3)
```

```
= 0x7fe3cf628000
SYS_mmap(0x7fe3cf62d000, 18520, 3, 50, 0xffffffff)
= 0x7fe3cf62d000
SYS_close(3)
= 0
SYS_mmap(0, 4096, 3, 34, 0xffffffff)
= 0x7fe3cf834000
SYS_mmap(0, 4096, 3, 34, 0xffffffff)
= 0x7fe3cf833000
SYS_mmap(0, 4096, 3, 34, 0xffffffff)
= 0x7fe3cf832000
SYS_arch_prctl(4098, 0x7fe3cf833700, 0x7fe3cf832000, 34, 0xffffffff)
= 0
SYS_mprotect(0x7fe3cf628000, 16384, 1)
= 0
SYS_mprotect(0x7fe3cf851000, 4096, 1)
= 0
SYS_munmap(0x7fe3cf835000, 103967)
= 0
__libc_start_main(0x40054c, 1, 0x7fff47008298, 0x4005a0, 0x400590 <unfinished ...>
fork( <unfinished ...>
SYS_clone(0x1200011, 0, 0, 0x7fe3cf8339d0, 0)
= 5967
<... fork resumed> )
= 5967
puts("\nI am parent" <unfinished ...>
SYS_fstat(1, 0x7fff47008060)
= 0
SYS_mmap(0, 4096, 3, 34, 0xffffffff
)                                                        = 0x7fe3cf84e000
I am child
SYS_write(1, "\n", 1
)                                                                       = 1
SYS_write(1, "I am parent\n", 12)
= -512
--- SIGCHLD (Child exited) ---
SYS_write(1, "I am parent\n", 12I am parent
)                                                           = 12
<... puts resumed> )
= 13
SYS_exit_group(13 <no return ...>
+++ exited (status 13) +++
```

asked Aug 21 '13 at 2:46

user3539
1,48372235

2 Answers 2

up vote 26 down vote accepted

The `fork()` and `vfork()` wrappers in glibc are implemented via the `clone()` system call. To better understand the relationship between `fork()` and `clone()`, we must consider the relationship between processes and threads in Linux.

Traditionally, `fork()` would dublicate all the resources owned by the parent process and assign the copy to the child process. This approach incurs considerable overhead, which all might be for nothing if the child immediately calls `exec()`. In Linux, `fork()` utilizes *copy-on-write* pages to delay or altogether avoid copying the data that can be shared between the parent and child processes. Thus, the only overhead that is incurred during a normal `fork()` is the copying of the parent's page tables and the assignment of a unique process descriptor struct, `task_struct`, for the child.

Linux also takes an exceptional approach to threads. In Linux, threads are merely ordinary processes which happen to share some resources with other processes. This is a radically different approach to threads compared to other operating systems such as Windows or Solaris, where processes and threads are entirely different kinds of beasts. In Linux, each thread has an ordinary `task_struct` of its own that just happens to be setup in such a way that it shares certain resources, such as an address space, with the parent process.

The `flags` parameter of the `clone()` system call includes a set of flags which indicate which resources, if any, the parent and child processes should share. Processes and threads are both created via `clone()`, the only difference is the set of flags that is passed to `clone()`.

A normal `fork()` could be implemented as:

```
clone(SIGCHLD, 0);
```

This creates a task which does not share any resources with its parent, and is set to send the `SIGCHLD` termination signal to the parent when it exits.

In contrast, a task which shares the address space, filesystem resources, file descriptors and signal handlers with the parent, in other words a *thread*, could be created with:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

`vfork()` in turn is implemented via a separate `CLONE_VFORK` flag, which will cause the parent process to sleep until the child process wakes it via a signal. The child will be the sole thread of execution in the parent's namespace, until it calls `exec()` or exits. The child is not allowed to write to the memory. The corresponding `clone()` call could be as follows:

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0)
```

The implementation of `sys_clone()` is architecture specific, but the bulk of the work happens in `do_fork()` defined in `kernel/fork.c`. This function calls the static `clone_process()`, which creates a new process as a copy of the parent, but does not start it yet. `clone_process()` copies the registers, assigns a PID to the new task, and either dublicates or shares appropriate parts of the process environment as specified by the clone `flags`. When `clone_process()` returns, `do_clone()` will wake the newly created process and schedule it to run.

up vote 2 down vote
The component responsible for translating userland system call functions to kernel system calls under Linux is the libc. In GLibC, the NPTL library redirects this to the `clone(2)` system call.

answered Aug 21 '13 at 3:30

Ignacio Vazquez-Abrams
30.6k66078

Your Answer

Not the answer you're looking for? Browse other questions tagged linux linux-kernel system-calls trace ltrace or ask your own question.