

master ▾

MIT6.S081 / lec01-introduction-and-examples /  
1.9-exec-and-wait-systemcall.md

Go to file

...



huihongxiao GitBook: [master] 30 p...



Latest commit bc9c619 on Apr 24

History

1 contributor

85 lines (55 sloc) | 10.2 KB



Raw

Blame



## 1.9 exec, wait系统调用

在接下来我展示的一个例子中，会使用echo，echo是一个非常简单的命令，它接收任何你传递给它的输入，并将输入写到输出。

```
$ echo a b c
a b c
```

我为你们准备了一个文件名为exec的代码，

```
[crash] cat -n exec.c
 1
 2 // exec.c: replace a process with an executable file
 3
 4 #include "kernel/types.h"
 5 #include "user/user.h"
 6
 7 int
 8 main()
 9 {
10     char *argv[] = { "echo", "this", "is", "echo", 0 };
11     exec("echo", argv);
12     printf("exec failed!\n");
13     exit(0);
14 }
[crash]
```

代码会执行exec系统调用，这个系统调用会从指定的文件中读取并加载指令，并替代当前调用进程的指令。从某种程度上来说，这样相当于丢弃了调用进程的内存，并开始执行新加载的指令。所以第12行的系统调用exec会有这样的效果：操作系统从名为echo的文件中加载指令到当前的进程中，并替换了当前进程的内存，之后开始执行这些新加载的指令。同时，你可以传入命令行参数，exec允许你传入一个命令行参数的数组，这里就是一个C语言中的指针数组，在上面代码的第10行设置好了一个字符指针的数组，这里的字符指针本质就是一个字符串（string）。

所以这里等价于运行echo命令，并带上“this is echo”这三个参数。所以当我运行exec文件，

```
$ exec  
this is echo
```

我可以看到“this is echo”的输出。即使我运行了exec程序，exec程序实际上会调用exec系统调用，并用echo指令来代替自己，所以这里是echo命令在产生输出。

有关exec系统调用，有一些重要的事情，

1. exec系统调用会保留当前的文件描述符表单。所以任何在exec系统调用之前的文件描述符，例如0，1，2等。它们在新的程序中表示相同的东西。
2. 通常来说exec系统调用不会返回，因为exec会完全替换当前进程的内存，相当于当前进程不复存在了，所以exec系统调用已经没有地方能返回了。

所以，exec系统调用从文件中读取指令，执行这些指令，然后就没有然后了。exec系统调用只会当出错时才会返回，因为某些错误会阻止操作系统为你运行文件中的指令，例如程序文件根本不存在，因为exec系统调用不能找到文件，exec会返回-1来表示：出错了，我找不到文件。所以通常来说exec系统调用不会返回，它只会在kernel不能运行相应的文件时返回。

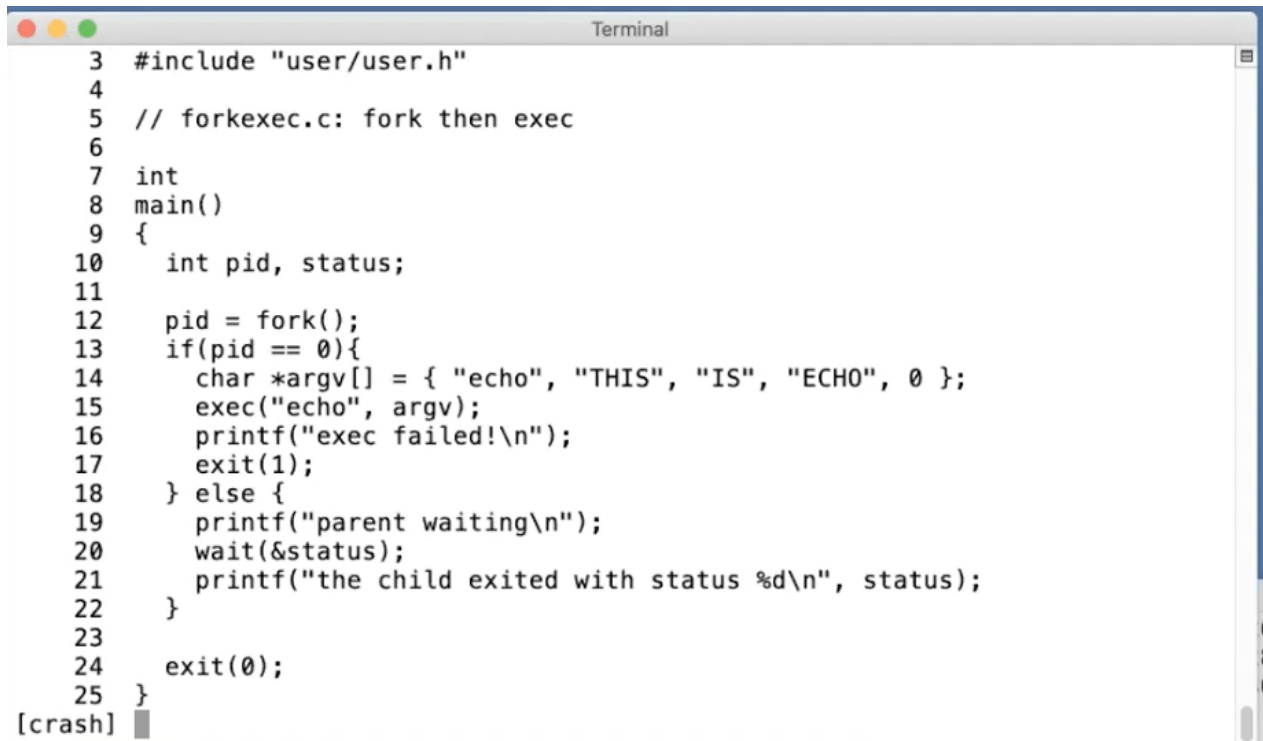
有关exec有什么问题吗？

学生提问：argv中的最后一个0是什么意思？

Robert教授：它标记了数组的结尾。C是一个非常低阶（接近机器语言）的编程语言，并没有一个方法来确定一个数组究竟有多长。所以为了告诉内核数组的结尾在哪，我们将0作为最后一个指针。argv中的每一个字符串实际上是一块包含了数据的内存指针，但是第5个元素是0，通常来说指针0是一个NULL指针，它只表明结束。所以内核中的代码会遍历这里的数组，直到它找到了值为0的指针。

好的，这就是一个程序如何用文件中的另一个程序来替代自己。实际上，当我们在Shell中运行类似于“echo a b c”的指令，或者ls，或者任何命令，我们不会想要代替Shell进程，所以我们不会希望Shell执行exec系统调用。如果我们这么做了，这里会用echo指令来替代Shell进程，当echo退出了，一切就结束了。所以我们不想要echo替代Shell。实际上，Shell会执行fork，之后fork出的子进程再调用exec系统调用，这是一个非常常见的Unix程序调用风格。对于那些想要运行程序，但是还希望能拿回控制权的场景，可以先执行fork系统调用，然后在子进程中调用exec。

这里有一个简单的例子，来演示fork/exec程序。



```
3 #include "user/user.h"
4
5 // forkexec.c: fork then exec
6
7 int
8 main()
9 {
10     int pid, status;
11
12     pid = fork();
13     if(pid == 0){
14         char *argv[] = { "echo", "THIS", "IS", "ECHO", 0 };
15         exec("echo", argv);
16         printf("exec failed!\n");
17         exit(1);
18     } else {
19         printf("parent waiting\n");
20         wait(&status);
21         printf("the child exited with status %d\n", status);
22     }
23
24     exit(0);
25 }
[crash]
```

在这个程序中的第12行，调用了fork。子进程从第14行开始，我们在子进程中与前一个程序一样调用exec。子进程会用echo命令来代替自己，echo执行完成之后就退出。之后父进程重新获得了控制。fork会在父进程中返回大于0的值，父进程会继续在第19行执行。

Unix提供了一个wait系统调用，如第20行所示。wait会等待之前创建的子进程退出。当我在命令行执行一个指令时，我们一般会希望Shell等待指令执行完成。所以wait系统调用，使得父进程可以等待任何一个子进程返回。这里wait的参数status，是一种让退出的子进程以一个整数（32bit的数据）的格式与等待的父进程通信方式。所以在第17行，exit的参数是1，操作系统会将1从退出的子进程传递到第20行，也就是等待的父进程处。&status，是将status对应的地址传递给内核，内核会向这个地址写入子进程向exit传入的参数。

Unix中的风格是，如果一个程序成功的退出了，那么exit的参数会是0，如果出现了错误，那么就会像第17行一样，会向exit传递1。所以，如果你关心子进程的状态的话，父进程可以读取wait的参数，并决定子进程是否成功的完成了。

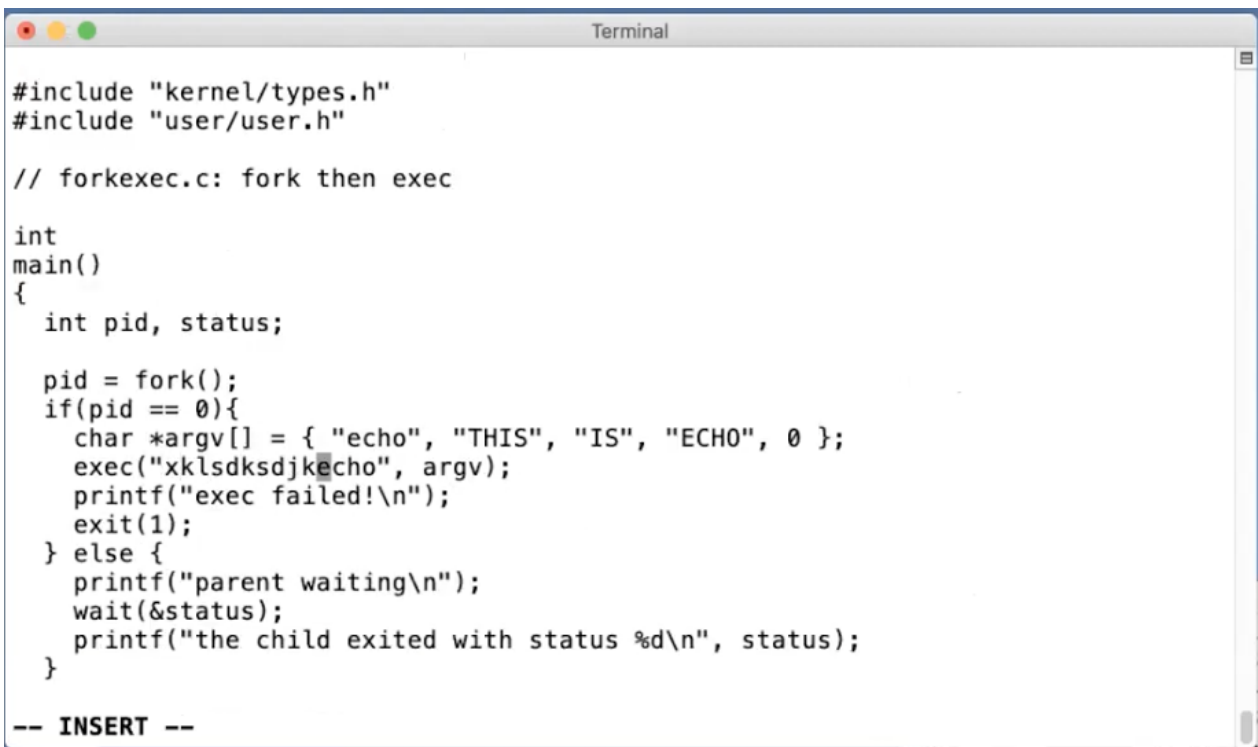
学生提问：有关第15行的exec系统调用，在刚刚提到exec会完全走到echo程序，而不会返回到fork出的子进程中，所以代码有可能走到底16，17行吗？

Robert教授：对于上面例子中的exec，代码不会走到16，17行，因为这里就是调用了echo。但是，如果我修改代码，那就有可能走到那两行了。首先，我先运行一下原始版本的程序

```
$ forkexec
parent waiting
THIS IS ECHO
the child exited with status 0
```

可以看出，程序执行了echo，并传入了相应的参数。同时子进程以状态0退出，表明echo成功的退出了，并且父进程在等待子进程。

接下来，我修改一下代码。这次我将会运行一个不存在的指令，

A screenshot of a terminal window titled "Terminal" showing the source code of a C program named forkexec.c. The code includes kernel/types.h and user/user.h. It defines a main function that forks a child process. The child process attempts to execute a command using exec, with arguments "echo", "THIS", "IS", "ECHO", and a null terminator. If exec fails, it prints "exec failed!\n" and exits with status 1. The parent process prints "parent waiting\n", waits for the child to finish, and then prints "the child exited with status %d\n", status). At the bottom of the terminal, the text "-- INSERT --" is visible.

```
Terminal

#include "kernel/types.h"
#include "user/user.h"

// forkexec.c: fork then exec

int
main()
{
    int pid, status;

    pid = fork();
    if(pid == 0){
        char *argv[] = { "echo", "THIS", "IS", "ECHO", 0 };
        exec("xklsdksdjkecho", argv);
        printf("exec failed!\n");
        exit(1);
    } else {
        printf("parent waiting\n");
        wait(&status);
        printf("the child exited with status %d\n", status);
    }
}

-- INSERT --
```

为了让修改生效，我需要退出QEMU，并重建所有的东西以使得我的修改能够被编译。之后我再运行forkexec，

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ forkexec
parent waiting
exec failed!
the child exited with status 1
```

这一次，因为我们想要执行的指令并不存在，`exec`系统调用会返回，我们可以看到“`exec failed!`”的输出，同时`exit(1)`的参数1，传递给了父进程，父进程会打印出子进程的退出码。所以，`exec`系统调用只会在出错的时候返回给调用进程。

这里有一些东西需要注意，实际上我认为你们很多人已经注意到了，这里是一个常用的写法，先调用`fork`，再在子进程中调用`exec`。这里实际上有些浪费，`fork`首先拷贝了整个父进程的，但是之后`exec`整个将这个拷贝丢弃了，并用你要运行的文件替换了内存的内容。某种程度上来说这里的拷贝操作浪费了，因为所有拷贝的内存都被丢弃并被`exec`替换。在大型程序中这里的影响会比较明显。如果你运行了一个几G的程序，并且调用`fork`，那么实际就会拷贝所有的内存，可能会要消耗将近1秒钟来完成拷贝，这可能会是个问题。

在这门课程的后面，你们会实现一些优化，比如说`copy-on-write fork`，这种方式会消除`fork`的几乎所有的明显的低效，而只拷贝执行`exec`所需要的内存，这里需要很多涉及到虚拟内存系统的技巧。你可以构建一个`fork`，对于内存实行`lazy`拷贝，通常来说`fork`之后立刻是`exec`，这样你就不用实际的拷贝，因为子进程实际上并没有使用大部分的内存。我认为你们会觉得这将是一个有趣的实验。

学生提问：为什么父进程在子进程调用`exec`之前就打印了“`parent waiting`”？

Robert教授：这里只是巧合。父进程的输出有可能与子进程的输出交织在一起，就像我们之前在`fork`的例子中看到的一样，只是这里正好没有发生而已。并不是说我们一定能看到上面的输出，实际上，如果看到其他的输出也不用奇怪。我怀疑这里背后的原因是，`exec`系统调用代价比较高，它需要访问文件系统，访问磁盘，分配内存，并读取磁盘中`echo`文件的内容到分配的内存中，分配内存又可能需要等待内存释放。所以，`exec`系统调用背后会有很多逻辑，很明显，处理这些逻辑的时间足够长，这样父进程可以在`exec`开始执行`echo`指令之前完成输出。这样说得通吧？

学生提问：子进程可以等待父进程吗？

Robert教授：Unix并没有一个直接的方法让子进程等待父进程。`wait`系统调用只能等待当前进程的子进程。所以`wait`的工作原理是，如果当前进程有任何子进程，并且其中一个已经退出了，那么`wait`会返回。但是如果当前进程没有任何子进程，比如在这个简单的例子中，如果子进程调用了`wait`，因为子进程自己没有子进程了，所以`wait`会立即返回-1，表明出现错误了，当前的进程并没有任何子进程。

简单来说，不可能让子进程等待父进程退出。

学生提问：当我们说子进程从父进程拷贝了所有的内存，这里具体指的是什么呢？是不是说子进程需要重新定义变量之类的？

Robert教授：在编译之后，你的C程序就是一些在内存中的指令，这些指令存在于内存中。所以这些指令可以被拷贝，因为它们就是内存中的字节，它们可以被拷贝到别处。通过一些有关虚拟内存的技巧，可以使得子进程的内存与父进程的内存一样，这里实际就是将父进程的内存镜像拷贝给子进程，并在子进程中执行。

实际上，当我们在看C程序时，你应该认为它们就是一些机器指令，这些机器指令就是内存中的数据，所以可以被拷贝。

学生提问：如果父进程有多个子进程，wait是不是会在第一个子进程完成时就退出？这样的话，还有一些与父进程交错运行的子进程，是不是需要多个wait来确保所有的子进程都完成？

Robert教授：是的，如果一个进程调用fork两次，如果它想要等两个子进程都退出，它需要调用wait两次。每个wait会在一个子进程退出时立即返回。当wait返回时，你实际上没有必要知道哪个子进程退出了，但是wait返回了子进程的进程号，所以在wait返回之后，你就可以知道是哪个子进程退出了。