实验2 进程控制

汪楚文 2018202114 04/12/2020

实验概述

[实验目的]

·熟悉fork,execve,exit等系统调用的使用

·通过编写程序理解Linux进程生命周期

[基本要求]

编写Linux环境下C程序，使用fork,exec,exit系统调用

[具体要求]

·程序调用fork创建子进程（20分）

·父子进程至少有一个调用exec族系统调用执行其他程序（20分）

·调用exit终止进程（10分）

·代码有注释（10分），提交实验报告（实验报告要求见3）

[进一步要求]

·使用wait系统调用代替示例代码exec.c中的sleep系统调用（10分）

·在调用exec时使用自己编写的C程序或者shell程序（20分）

·将实验1与实验2结合（10分）

[实验报告要求]

详见文件README.txt

【重要】本实现的全部要求都已实现

实验内容

[实验环境]

Ubuntu X86_64

[实验思路]

根据实验要求，本c程序的设计如下：

一.fork()创建子进程

使用pid=fork()来创建子进程，并根据fork()的返回值来判断此时正在runing的进程是父进程还是子进程，进而做相应的处理。返回值小于0则输出"fork error“

二.exit(0)来结束进程

通过exit()来退出进程，代替return。当未正常退出时c程序的main函数才会return。

三.父进程调用wait(NULL)

wait函数用于父进程和子进程的同步，让父进程等待子进程结束。参数为NULL，由于子进程退出失败时会报错，故不需要用参数&status去获得子进程终止时的返回值

四.子进程调用execve函数

通过子进程执行退出时执行execve函数（系统调用）来调用执行实验一的shell程序（mtest1.sh）

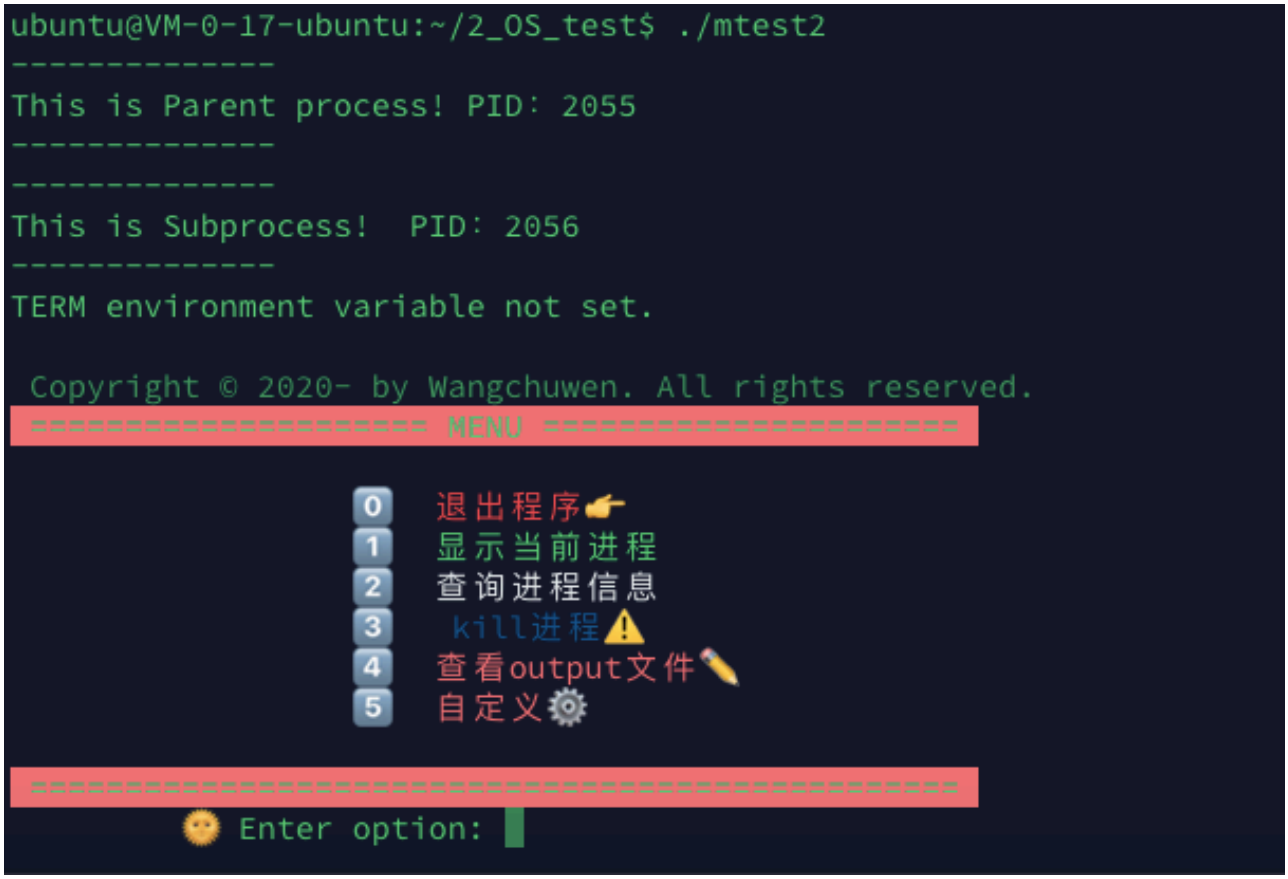
五.运行提示

通过printf("This is Parent process! PID：%d\n",getpid());和 printf("This is Subprocess! PID：%d\n",getpid());来表明此时正在执行的进程。

六.退出异常检查

当进程没有通过exit或exec来退出时，main函数return 0，将会输出"Unexpectded error happened\n"

实验结果



(1)gcc -c mtest2.c ->gcc -o mtest2 mtest2.o -> ./mtest2 编译运行mtest2.c

(2)执行父进程，如图，输出提示信息和PID,并执行wait(NULL),等待与子进程同步

(3)执行子进程，如图，输出提示信息和PID。

(4)子进程发起execve系统调用，子进程退出,调用执行mtest1.sh,如图mtest1.sh成功执行。

(5)父进程exit退出，mtest2程序终止。

实验中遇到的问题及解决办法

一.execve传入的参数不对导致实验一的mtest1.sh不能执行，并报错execve error

解决办法：
char *argv[ ]={"/bin/bash","shell",NULL};
execve("mtest1.sh",argv,NULL)
了解了execve个参数的意义后，将程序更改为上述语句，成功执行mtest1.sh

二.调用wait函数，发现不能正常编译执行。

解决办法：.c文件加上#include <sys/wait.h>头文件

三.printf("This is Parent process! PID：%d\n",pid);和 printf("This is Subprocess! PID：%d\n",pid);输出的pid总为0，不是真实的pid。

解决办法：printf("This is Parent process! PID：%d\n",getpid());
printf("This is Subprocess! PID：%d\n",getpid());
用系统调用`getpid()`来获取真实`pid`

Exec族函数

[所需头文件]

#include <unistd.h>

[函数说明]

exec函数族提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了。

[函数原型]

int execl(const char *pathname, const char *arg, ...)
int execv(const char *pathname, char *const argv[])
int execle(const char *pathname, const char *arg, ..., char *const envp[])
int execve(const char *pathname, char *const argv[], char *const envp[])
int execlp(const char *filename, const char *arg, ...)
int execvp(const char *filename, char *const argv[])

成功：函数不会返回

出错：返回-1，失败原因记录在error中

这6 个函数在函数名和使用语法的规则上都有细微的区别，下面就可执行文件查找方式、参数表传递方式及环境变量这几个方面进行比较说明。

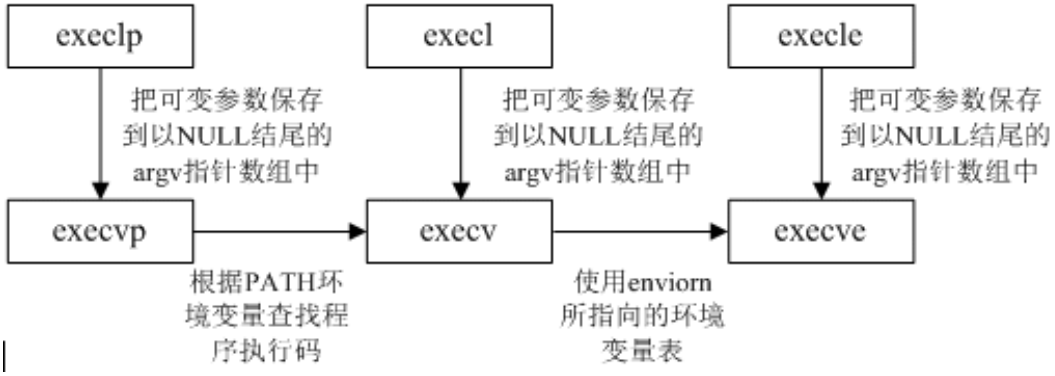
① 查找方式：上表其中前4个函数的查找方式都是完整的文件目录路径（pathname），而最后2个函数（也就是以p结尾的两个函数）可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。

② 参数传递方式：exec函数族的参数传递有两种方式，一种是逐个列举(l)的方式，而另一种则是将所有参数整体构造成指针数组(v)进行传递。

在这里参数传递方式是以函数名的第5位字母来区分的，字母为“l”（list）的表示逐个列举的方式，字母为“v”（vertor）的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是NULL。读者可以观察execl、execle、execlp的语法与execv、execve、execvp的区别。

③ 环境变量：exec函数族使用了系统默认的环境变量，也可以传入指定的环境变量。这里以“e”（environment）结尾的两个函数execle、execve就可以在envp[]中指定当前进程所使用的环境变量替换掉该进程继承的所以环境变量。

事实上，这6个函数中真正的系统调用只有execve，其他5个都是库函数，它们最终都会调用execve这个系统调用，调用关系如下图所示：

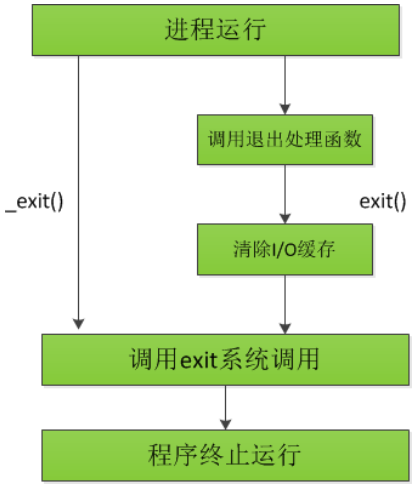


相似函数辨析

一.fork与vfork

1. fork ()子进程拷贝父进程的数据段，代码段
   vfork():子进程与父进程共享数据段
2. fork(): 父子进程的执行次序不确定
   vfork() : 保证子进程先运行，在调用exec 或exit 之前与父进程数据是共享的,在它调用exec或exit 之后父进程才可能被调度运行。
3. vfork （）保证子进程先运行，在调用exec 或exit 之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

二.exit与_exit



1._exit()执行后会立即返回给内核，而exit()要先执行一些清除操作，然后将控制权交给内核。
2.调用_exit()函数时，其会关闭进程所有的文件描述符，清理内存，以及其他一些内核清理函数，但不会刷新流（stdin 、stdout、stderr……）。exit()函数是在_exit()函数上的一个封装，它会调用_exit，并在调用之前先刷新流。
3.exit()函数与_exit()函数最大的区别就在于，exit()函数在调用exit系统之前要检查文件的打开情况，把文件缓冲区的内容写回文件。

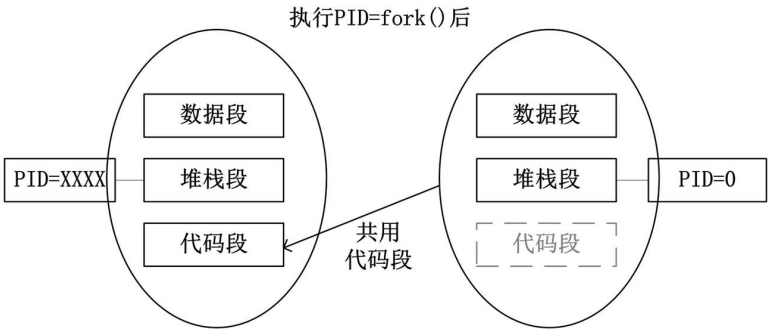
三.fork与clone

Fork：fork创造的子进程是父进程的完整副本，复制了父亲进程的资源，包括内存的内容task_struct内容
Vfork：vfork创建的子进程与父进程共享数据段,而且由vfork()创建的子进程将先于父进程运行
Clone：Linux上创建线程一般使用的是pthread库 实际上linux也给我们提供了创建线程的系统调用，就是clone，其相对较轻量级。
fork()函数复制时将父进程的所以资源都通过复制数据结构进行了复制，然后传递给子进程，所以fork()函数不带参数；clone()函数则是将部分父进程的资源的数据结构进行复制，复制哪些资源是可选择的，这个可以通过参数设定，所以clone()函数带参数。fork()可以看出是完全版的clone()，而clone()克隆的只是fork()的一部分。
对于访问次序，fork的父进程和子进程的执行顺序是不定的；vfork是子进程先执行，父进程挂起直至子进程调用exec或exit;clone可通过CLONE_VFORK参数来决定访问次序。

对fork返回两次的理解

关键注意两点：1.fork返回后，父进程或子进程的执行位置。
2.两次返回的pid存放的位置。

当程序执行到下面的语句： pid=fork();
子进程和父进程使用相同的代码段，但是会拥有各自的数据段和堆栈段。



由于在复制时复制了父进程的堆栈段，所以两个进程都停留在fork函数中，等待返回。因此fork函数会返回两次，一次是在父进程中返回，另一次是在子进程中返回，这两次的返回值是不一样的。

1）在父进程中，fork返回新创建子进程的进程ID；
2）在子进程中，fork返回0；
3）如果出现错误，fork返回一个负值。

堆栈段里有一块空间是用来存放pid变量，父子进程拥有不同的堆栈段，而内核给这两个堆栈段里的pid赋上不同的值。

之所以fork返回两次，并不是因为一个函数语句可以同时返回两个值，而是因为fork()在父进程和子进程都被执行，所以一共就被执行了两次，故有次返回。两次返回是来自两个进程（子进程和父进程），这两个进程来自同一个程序的两次执行。

五个常用系统调用

一.setsid——创建一个会话，getsid——获取会话id

a）setsid——创建一个会话
#include <unistd.h>
pid_t setsid(void);
如果调用这个函数的进程不是进程组的leader，那么就创建一个新的会话，调用者进程是这个会话的leader，进程组的leader，并且没用控制终端与之相关联。进程组id和会话id设置为这个进程的pid。这个进程是这个进程组和会话的唯一一个进程。
成功返回调用者进程的会话id，否则返回(pid_t)-1，errno被设置。

b）getsid——获取会话id
#include <unistd.h>
pid_t getsid(pid_t pid);
getsid(0)返回当前进程的会话id，getsid(p)返回进程id为p的会话id，会话id是会话leader的进程组id。
成功返回会话id，否则返回(pid_t)-1，errno被设置。

二.）mkfifo——创建一个FIFO的特殊文件（命名管道）

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
mkfifo创建一个名称为pathname指向字符串的FIFO特殊文件，mode指定FIFO文件的访问权限。
被umask修改，所以创建的文件权限是(mode & ~umask)。
一旦创建了FIFO特殊文件，任何进程可以以读写的方式打开，就像使用一般文件一样。
但它的写端和读端必须同时打开，以读的方式打开FIFO文件会阻塞直到另一个进程以写的方式
打开同一个FIFO文件，反之亦然。
执行成功返回0，错误情况下返回-1，errno被设置。

三.ptrace——进程跟踪

#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,void *addr, void *data);
ptrace系统调用提供了一个进程（tracer）可以观察和控制另一个执行的进程（tracee），并且检查和改变tracee的内存和寄存器。
一个tracee首先需要和一个tracer关联。关联和随后的命令是在一个线程中的，在多线程的进程中，每一个线程都可以单独的和一个tracer（可以不相同）关联或者不关联。因此，tracee总是意味着一个线程，而不是一个（多线程）进程。跟踪命令总是以如下的方式向tracee发送：
ptrace(PTRACE_foo, pid, ...)
这里，pid是一个相应的linux线程的id。
当被跟踪时，tracee会在每次收到信号的时候停止，即使这个信号被忽略了，一种例外是SIGKILL信号，它和平常一样。tracer将会在下次调用waitpid（或者其他相关联的wait系统调用），这可以返回一个status值，表明tracee停止的原因。这时tracer可以使用ptarce请求来检查或修改tracee，然后tracer可以在忽略tracee收到的信号（或者传递另外一个信号）的情况下继续tracee的运行。
当tracer跟踪完毕，可以让tracee继续以平常方式运行，这通过PTRACE_DETACH模式来完成。
request的值决定要完成的动作，下面简单介绍几个：
a）PTRACE_TRACEME
表示该进程被父进程所跟踪，父进程应该希望跟踪该进程。
b）PTRACE_PEEKTEXT, PTRACE_PEEKDATA
从内存地址中读取一个字（word），内存地址由addr给出。在linux中没有区分test和data地址空间，所以它们是等价的。
c）PTRACE_PEEKUSR
从USER区域中读取一个字（word），偏移量为addr。
d）PTRACE_POKETEXT, PTRACE_POKEDATA
往内存地址中写入一个字（word）。内存地址由addr给出。
e）PTRACE_POKEUSR
往USER区域中写入一个字（word）。偏移量为addr。
执行错误返回-1，errno被设置。

四.getpid，getppid——获取进程识别号

#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
获取进程标识号。这可以作为一个独一无二的临时文件名。
pid_t getppid(void);//获取父进程标示号。
本实验有用到此系统调用

五. shmget——获取共享内存

#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
shmget函数得到一个与键值key关联的共享内存标识符。
以下情况创建一个新的共享内存：
a，key的值是IPC_PRIVATE，这时shmflg的值不起作用。
b，key的值不是IPC_PRIVATE，且shmflg指定了IPC_CREAT，若key相应的共享内存不存在，就创建一个新的共享内存，如果存在则打开这个共享内存。
以上两种情况创建一个大小为size（round到PAGE_SIZE的整数倍）的共享内存。
如果shmflg同时制定了IPC_CREAT和IPC_EXCL，且key相应的共享内存存在，那么shmget执行失败，errno被设置。
shmflg的最低9个有效位用来设置共享内存的访问权限，可执行没有意义。
当一个新的共享内存创建的时候，它的内存被初始化成0，与它关联的结构体shmid_ds初始化为：
shm_perm.cuid和shm_perm.uid被设置成进程的有效用户ID(EUID)。
shm_perm.cgid和shm_perm.gid被设置成进程的有效组ID(GUID)。
shm_perm.mode的最低9个有效位被设置成shmflg的最低9个有效位。
shm_segsz被设置成size。
shm_lpid,shm_nattch,shm_atime,shm_dtime被设置成0。
shm_ctime被设置成当前时间。
执行成功返回一个有效的共享内存标识符，否则返回-1，errno被设置。

项目相关文件说明

OStest2.pdf----------实验报告

README.txt----------实验要求原始文件

mtest2.c----------c程序源代码

mtest1.sh----------实验一的shell程序，用于调用

超链接：点击此处可在github中查看项目：）