实验3 进程控制与线程创建

实验目的

1. 掌握Linux中如何加载子进程自己的程序；

2. 掌握父进程通过创建子进程完成某项任务的方法；

3. 掌握系统调用exit()和\_exit()调用的使用；

4.掌握Linux线程的基本概念和使用。

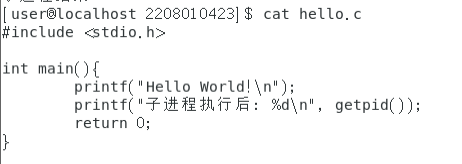
实验内容

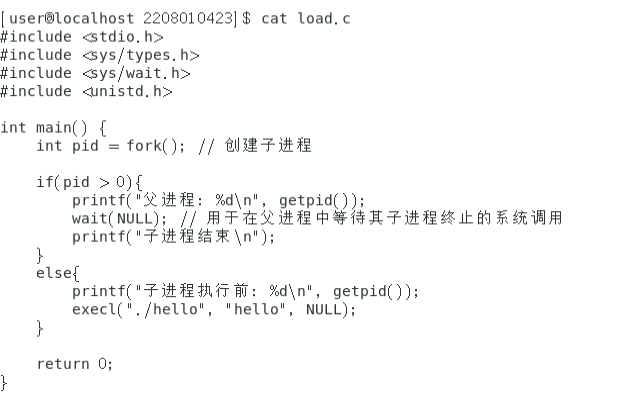
1. 设计两个程序hello.c和load.c，hello.c的功能是输出“hello world！”；load.c完成的功能是：创建一个子进程，子进程执行hello.c编译后的程序，父进程输出一行字符串（内容自定）后等待子进程结束（wait函数）。使用execl（）加载子进程的程序。子进程在execl前后各输出一次进程的pid，分析结果。

**要求：1.写出程序代码及运行截图**

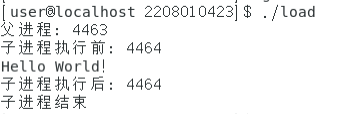
1. **总结exec系列函数的作用**

**程序代码 :**





**运行结果 :**



运行结果先是父进程打印自己的 pid，然后 wait(NULL) 等待子进程结束。

子进程先打印自己的 pid, 然后执行 execl() 函数，运行 hello.c, 打印 Hello World!. 并输出子进程 id，**可以注意到进程 id 并不会改变。**

**exec系列函数的作用**

execl : 执行指定路径的程序，参数以可变参数列表的形式提供。

execle : 类似 execl, 但可以传递环境变量数组

execlp : 类似于 execl, 但可以根据环境变量查找可执行文件

execv : 执行指定路径程序，参数以指针数组的形式提供

execve : 类似 execv, 可以传递环境变量数组

execvp : 类似 execv, 可以根据环境变量查找可执行文件。

exec 系列函数不会创建新进程，只是替换现有进程内容 ； 成功执行时，不返回 ，失败时返回 -1 并设置 errno ；进程号不变。

2．运行以下程序，注意程序的执行结果和printf函数（以行为单位进行缓冲的）的关系。

#include<stdlib.h>

#include <stdio.h>

#include <unistd.h>

int main()

{ printf("this is first line ");

sleep(2);

printf("this is the second line ");

sleep(2);

printf("this is the third line");

return 0;

}

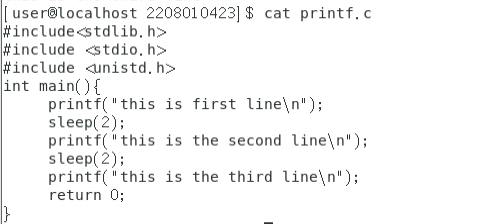
实验步骤：

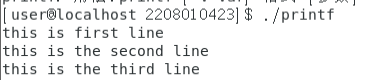
（1）运行以上程序，查看运行结果



等待 6s 之后，再全部输出。

1. 在每条printf语句后都加上\n，再运行以上程序，观察两种结果的不同，分析printf缓冲的作用。





每隔两秒输出一行语句。

**关于 printf 缓冲的作用 :**

行缓冲机制，只有遇到 \n 或者缓冲区满了才会输出。

对于本程序，第一种写法，没有 \n, 所以三个输出最后才输出 ；第二种写法，每个语句都有 \n, 所以执行之后就会输出。

**要求：写出程序代码及运行截图**

3. 运行以下程序，分析程序执行结果。

#include<stdlib.h>

#include <stdio.h>

#include <unistd.h>

main()

{ int p;

printf("This is the fisrt line");

p=fork();

if (p>0)

{ wait(0);

printf("this is parent ");

}

else{

printf("this is child first\n");

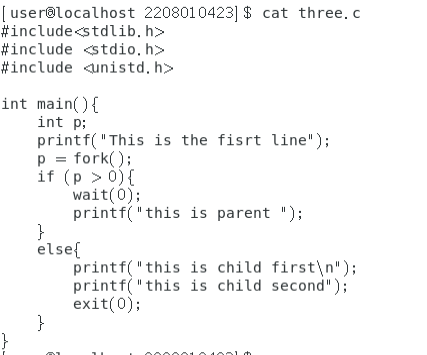
printf("this is child second");

exit(0);

}

}

实验步骤：





1. 分析程序的执行结果。若第一条printf语句后加上\n结果如何？

**分析程序运行结果 ：**

fork() 复制了整个进程，包括未刷新的标准 I/O 缓冲区。

第一次 fork 的时候，均未输出，缓冲区都有 This is the fisrt line.

来到父进程，等待子进程结束。

子进程遇到 printf("this is child first\n"); ，缓冲区刷新输出

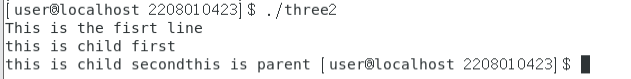
This is the fisrt linethis is child first

然后缓冲区装入 this is child second, exit 之后输出。

来到父进程，缓冲区装入 "this is parent ", 程序结束之后一并输出

This is the fisrt linethis is parent。

**第一条语句加上 \n：**

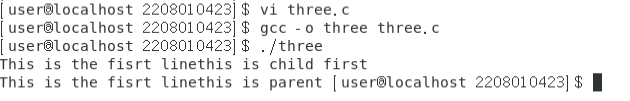


先直接输出 "This is the fisrt line\n"。

然后 fork(), 后面来到父进程，wait 子进程执行。

子进程输出 "this is child first\n", 然后在输出 this is child second, 之后父进程输出 this is child first\n

1. **若exit(0)换为\_exit(0)结果会怎样？**



缓冲区装入 This is the fisrt line, 然后被 fork 一次。

父进程等待子进程执行，子进程遇到 printf("this is child first\n"); 刷新 I/O,

此时输出 This is the fisrt linethis is child first

然后子进程缓冲区装入 "this is child second", 之后遇到 \_exit(0), 不刷新缓冲，不再输出。

然后来到父进程，缓冲区装入 this is parent , 输出This is the fisrt linethis is parent, 程序结束。

1. 分析printf和fork函数，printf和\_exit函数一起使用时特点。

**printf**

printf 的输出通常先存储在缓冲区中，并不会立刻输出到屏幕。缓冲区在以下情况下会刷新并输出内容：遇到换行符 \n（对于行缓冲的标准输出） ；缓冲区满了。进程正常结束（调用 exit() 时会自动刷新缓冲区）；调用 fflush(stdout) 以手动刷新缓冲区。

**fork 函数**

fork() 创建一个子进程，子进程是父进程的一个副本，继承了父进程的代码和数据，包括未刷新到屏幕的标准 I/O 缓冲区。

如果 fork() 前的 printf 还未刷新缓冲区，则子进程和父进程会共享同一个缓冲区内容，这会导致缓冲区内容可能被多次输出。

**printf和\_exit函数一起使用时特点。**

如果在 printf 后直接调用 \_exit()，并且 printf 不输出换行符，由于 \_exit() 不会刷新缓冲区，缓冲区中的内容将不会被输出到屏幕。

1. 运行以下程序，回答问题。

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

void \*run(void \*ptr){

pid\_t pid;pthread\_t tid;

pid = getpid(); //获取进程id

tid = pthread\_self(); //获取线程id

printf("In thread,process id:%lu,thread id:%lu\n",(unsigned long)pid,(unsigned long)tid);

return 0;

}

int main(){

pthread\_t tid[3],mtid;

pid\_t pid;int i;

int ret=0;

for (i=0;i<3;i++){

ret=pthread\_create(&tid[i],NULL,run,NULL);

if(ret) {

printf("create thread failed ");

return 0;

}

}

for(i=0;i<3;i++) pthread\_join(tid[i],NULL);

pid = getpid(); //获取进程id

mtid = pthread\_self(); //获取线程id

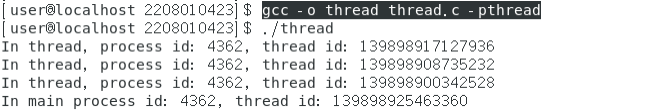
printf("In main process id:%lu,thread id:%lu\n",(unsigned long)pid,(unsigned long)mtid);

return 0;

}

1. 编辑程序thread.c，编译运行，运行结果是？

编译时使用如下命令：gcc -o thread thread.c -pthread



1. 根据程序结果分析，程序中的进程和线程有何关系？

可以从运行结果看出，四个 process id 都是相同的，所以每个线程共享同一个进程的进程ID (pid)。 而每个线程有不同的线程ID (tid)，这些线程ID是线程的唯一标识符，用来区分不同的线程。

**要求：写出运行截图及结果分析**