**实验一 进程控制实验**

**一、实验目的**

1、掌握进程的概念，明确进程和程序的区别

2、认识和了解并发执行的实质。

3、学习进程创建方法及进程树的概念。

4、学习进程映像的更换。

**二、实验内容**

1、进程的创建

编写一段程序，使用系统调用fork( )创建两个子进程，在系统中有一个父进程和两个子进程活动。让每个进程在屏幕上显示一个字符；父进程显示字符“a”，子进程分别显示字符“b” 和“c”。试观察记录屏幕上的显示结果，并分析原因。**（1分）**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid1, pid2;

pid1 = fork();

if (pid1 < 0) {

perror("fork failed");

exit(1);

}

if (pid1 == 0) {

printf("b\n");

exit(0);

}

wait(NULL);

pid2 = fork();

if (pid2 < 0) {

perror("fork failed");

exit(1);

}

if (pid2 == 0) {

printf("c\n");

exit(0);

}

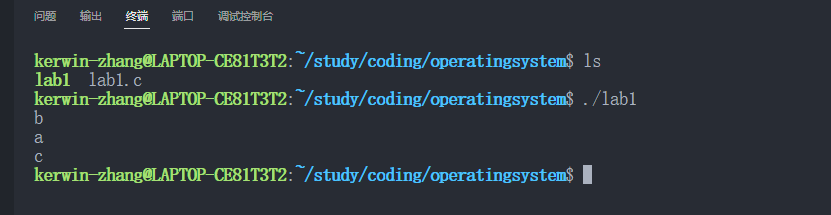
printf("a\n");

wait(NULL);

return 0;

}

执行结果：



分析：

第一次 fork() 调用创建了 pid1 子进程。在父进程中，fork() 返回子进程的 PID，因此继续执行后面的代码。在 pid1 子进程中，fork() 返回 0，然后执行 printf("b\n"); 打印字符 "b"，随后 exit(0); 终止子进程。

父进程在创建 pid1 子进程后立即调用 wait(NULL);。这个调用使得父进程在这里阻塞，直到 pid1 子进程终止。因为 pid1 子进程很快执行完毕并退出，所以 wait() 调用很快返回，父进程可以继续执行。

接下来，父进程执行第二次 fork() 调用，创建 pid2 子进程。同样，在父进程中，fork() 返回子进程的 PID，父进程继续执行后面的代码。在 pid2 子进程中，fork() 返回 0，然后执行 printf("c\n"); 打印字符 "c"，随后 exit(0); 终止子进程。

父进程在创建 pid2 子进程后立即执行 printf("a\n"); 打印字符 "a"。这时，父进程已经完成了它的所有 fork() 调用和打印操作。

父进程再次调用 wait(NULL); 等待 pid2 子进程终止。由于 pid2 子进程在父进程打印 "a" 之前已经开始执行并很快结束，wait() 调用会很快返回。

父进程执行完毕，程序结束。

2、修改第一题，在每个进程中显示当前进程识别码和父进程识别码。**（2分）**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid1, pid2;

pid1 = fork();

if (pid1 < 0) {

perror("fork failed");

exit(1);

}

if (pid1 == 0) {

printf("Child process 1 (PID: %d, PPID: %d)\n", getpid(), getppid());

printf("b\n");

exit(0);

}

wait(NULL);

pid2 = fork();

if (pid2 < 0) {

perror("fork failed");

exit(1);

}

if (pid2 == 0) {

printf("Child process 2 (PID: %d, PPID: %d)\n", getpid(), getppid());

printf("c\n");

exit(0);

}

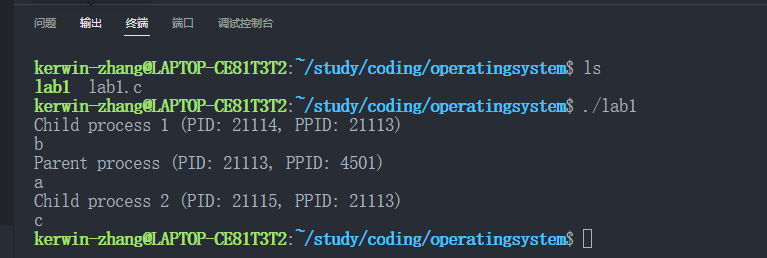
printf("Parent process (PID: %d, PPID: %d)\n", getpid(), getppid());

printf("a\n");

wait(NULL);

return 0;

}



3、 编写程序创建进程树如图1或 图2所示，在每个进程中显示当前进程识别码和父进程识别码。（3分）

父进程

子进程

图1 进程树

图2 进程树

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#define NUM\_CHILDREN 2 // 定义每个进程可以创建的子进程数量

void create\_process\_tree(int depth, int parent\_pid)

{

if (depth <= 0)

{

return; // 达到递归深度，停止创建更多子进程

}

pid\_t child\_pid = fork();

if (child\_pid < 0)

{

perror("fork failed");

exit(1);

}

else if (child\_pid == 0)

{

// 子进程的代码

printf("Child process (PID: %d, PPID: %d) created at depth %d\n", getpid(), getppid(), depth);

// 递归创建子进程

create\_process\_tree(depth - 1, getpid());

exit(0); // 子进程结束

}

// 父进程的代码

printf("Parent process (PID: %d) creating child at depth %d\n", parent\_pid, depth);

wait(NULL); // 等待子进程结束

// 递归创建下一个子进程

create\_process\_tree(depth - 1, child\_pid);

}

int main()

{

pid\_t root\_pid = fork();

if (root\_pid < 0)

{

perror("fork failed");

exit(1);

}

else if (root\_pid == 0)

{

printf("Root process (PID: %d)\n", getpid());

create\_process\_tree(3, getpid()); // 创建深度为 3 的进程树

exit(0); // 根进程结束

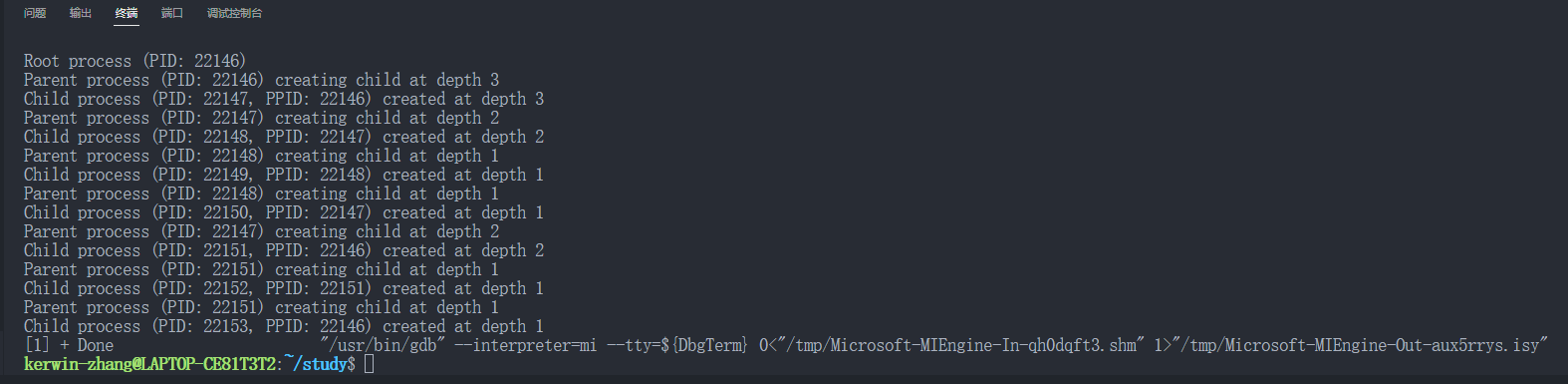
}

// 等待根进程结束

wait(NULL);

return 0;

}



1. 编程创建2个子进程，子进程1运行指定路径下的可执行文件（如：/home/user/birthday），子进程2在暂停10s之后退出，父进程先用阻塞方式等待第一个子进程的结束，然后用非阻塞方式等待另一个子进程的退出，待收集到第二个子进程结束的信息，父进程就返回。（3分）

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

int main()

{

pid\_t child\_pid1, child\_pid2;

// 创建第一个子进程，执行指定路径下的可执行文件

child\_pid1 = fork();

if (child\_pid1 < 0)

{

perror("fork failed");

exit(1);

}

if (child\_pid1 == 0)

{

// 子进程1的代码，执行指定路径下的可执行文件

// 确保 "lab1" 程序具有执行权限

char \*argv[] = {"/home/kerwin-zhang/study/coding/operatingsystem/lab1", NULL};

execv(argv[0], argv); // 使用 execv 来执行 "lab1" 程序

perror("execv failed");

exit(1);

}

// 等待第一个子进程结束

int status;

waitpid(child\_pid1, &status, 0); // 阻塞等待

// 创建第二个子进程，它将暂停10秒后退出

child\_pid2 = fork();

if (child\_pid2 < 0)

{

perror("fork failed");

exit(1);

}

if (child\_pid2 == 0)

{

// 子进程2的代码，暂停10秒后退出

sleep(10);

exit(0);

}

// 父进程非阻塞等待第二个子进程退出

while (waitpid(child\_pid2, &status, WNOHANG) == 0)

{

// 检查子进程是否已经退出

if (WIFEXITED(status))

{

// 子进程已退出，收集其退出状态并退出循环

printf("Child process 2 exited with status %d\n", WEXITSTATUS(status));

break;

}

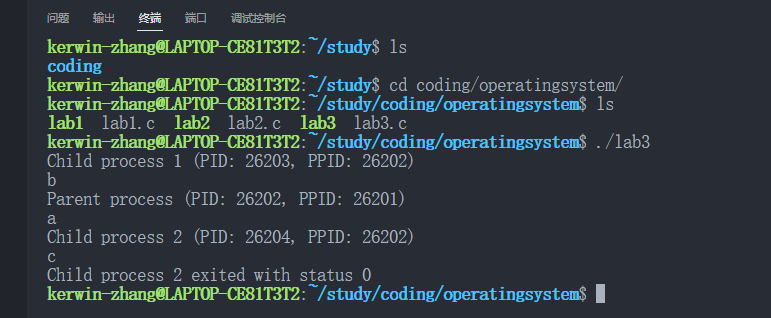
// 否则，继续等待

}

// 父进程结束

return 0;

}



**三、实验总结和体会（1分）**

我深入探索了LINUX系统中的进程创建和管理机制。通过编写 C 程序来创建子进程，并使用 fork() 系统调用来实现进程的复制，我对操作系统的并发执行有了更直观的理解。实验过程中，我学习了如何使用 waitpid() 函数以不同的方式（阻塞和非阻塞）等待子进程的结束，这对于理解进程间通信和同步非常重要。

通过对 execv() 函数的使用，我了解到了如何在子进程中执行另一个程序，并且认识到了确保执行权限正确设置的重要性。此外，我还体会到了在多进程环境下，父进程和子进程之间的资源共享和独立性，以及它们在执行顺序上的不确定性。

实验中，我也遇到了一些挑战，比如正确设置环境变量和处理 fork() 调用失败的情况。这些挑战让我认识到了在实际编程中，错误处理和资源管理的重要性。