

**HUBEI UNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**操作系统原理**

**实 验 报 告**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验项目 | 进程控制实验 | | | |
| 学生姓名 | 章崇文 | 学生学号 | 202202296 | |
| 学生班级 | 计算机222 | 完成日期 | 2024.4.3 | |
| 实验成绩 |  | 评阅日期 | |  |
| 评阅教师 |  | | | |

**湖北汽车工业学院实验报告**

班 号 计算机222 学 号 202202296  姓 名 章崇文

选课班中的序号  完成日期 年 月 日 至 节

## 实验一 进程控制实验

一、实验目的

1、掌握进程的概念，理解进程和程序的区别。

2、认识和了解并发执行的实质。

3、学习使用系统调用fork()创建新的子进程方法，理解进程树的概念。

4、学习使用系统调用wait()或waitpid()实现父子进程同步。

5、学习使用getpid()和getppid()获得当前进程和父进程的PID号。

6、掌握使用exec簇函数实现进程映像更换的方法。

7、了解系统编程，学习父进程如何通过创建一个子进程来完成某项特定任务的方法。

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，并分析结果。（1分）

<参考程序>

#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);

}

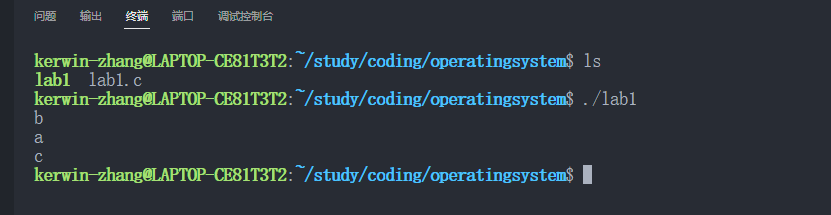
printf("a\n");

wait(NULL);

return 0;

}

执行结果及结果分析：



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。

1. 修改第一题，在父进程中显示当前进程识别码，在每个子进程中显示当前进程识别码和父进程识别码，运行程序查看结果，分析运行结果。（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);

}

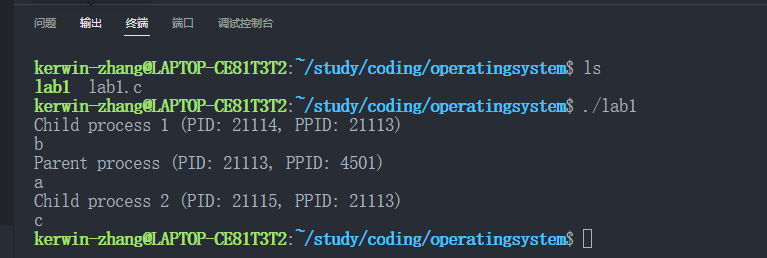
printf("Parent process (PID: %d, PPID: %d)\n", getpid(), getppid());

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。（2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

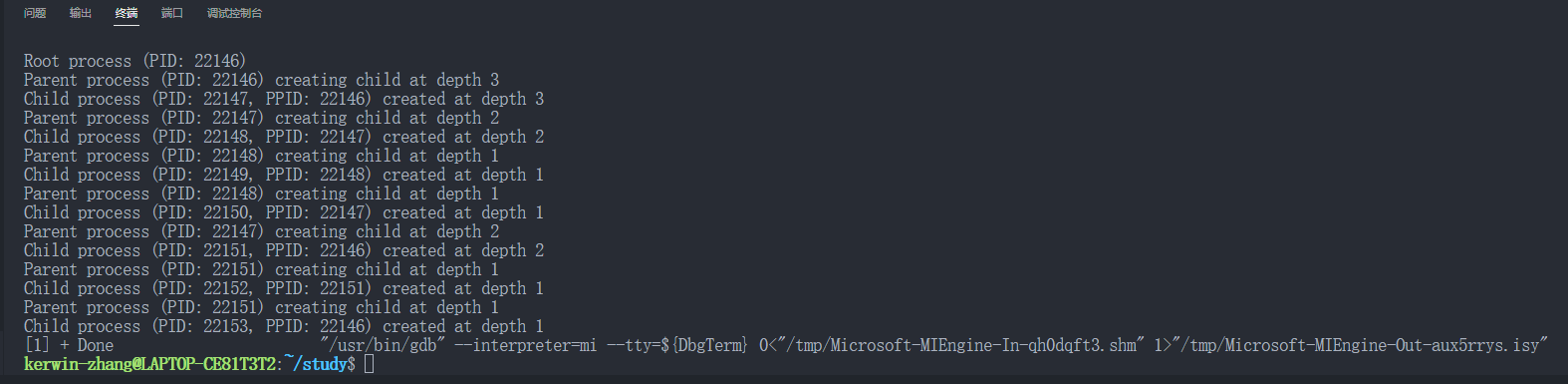
}

// 等待根进程结束

wait(NULL);

return 0;

}



4、分析程序，给出编译及执行过程和结果截屏。（2分）

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t child;

int p;

while ((child = fork()) == -1);

if (child == 0) {

printf("In child: sleep for 10 seconds and then exit.\n");

sleep(10);

exit(0);

} else {

do {

p = waitpid(child, NULL, WNOHANG);

if (p == 0) {

printf("In father: The child process has not exited.\n");

sleep(1);

}

} while (p == 0);

if (p == child) {

printf("Get child exit code then exit!\n");

} else {

printf("Error occurred!\n");

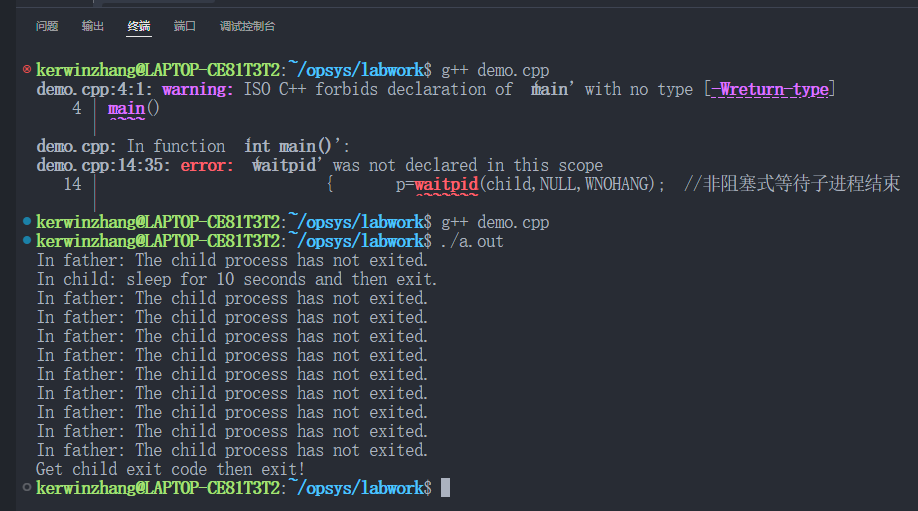
}

}

exit(0);

}

编译及执行过程和运行结果截屏：



分析程序功能：

这段代码的功能是创建一个父子进程，并使用进程间通信机制等待子进程的退出。

代码执行的过程如下：

主函数 main 开始执行，首先声明了变量 child 和 p，它们的类型分别是 pid\_t 和 int。

在一个循环中，调用 fork 函数创建子进程。如果 fork 返回值为 -1，表示创建子进程失败，继续循环直到成功创建子进程。

如果 fork 返回值为 0，说明当前进程是子进程。在子进程中，输出一条消息，然后使用 sleep 函数暂停执行 10 秒钟，最后调用 exit 函数退出子进程。

如果 fork 返回值不为 0，说明当前进程是父进程。进入 else 分支。

在父进程中，使用 do-while 循环来等待子进程的退出。循环中调用 waitpid 函数，传入子进程的进程 ID child，以及 WNOHANG 标志，使 waitpid 成为非阻塞调用，即立即返回结果。

如果 waitpid 返回值为 0，表示子进程尚未退出，输出一条消息并休眠 1 秒。

继续循环，直到 waitpid 返回值不为 0，表示子进程已经退出。

根据 waitpid 的返回值判断子进程的退出状态：

如果返回值等于子进程的进程 ID child，说明成功获取了子进程的退出状态，输出一条消息并退出父进程。

如果返回值不等于子进程的进程 ID child，说明出现了错误，输出一条错误消息。

最后调用 exit 函数退出父进程。

2） #include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

int child, p;

while ((child = fork()) == -1);

if (child == 0) {

execl("/home/kerwinzhang/opsys/labwork/a.out", "", NULL);

exit(0);

} else {

p = waitpid(child, NULL, 0);

if (p == child) {

printf("Get child exit code then exit!\n");

} else {

printf("Error occurred!\n");

}

}

exit(0);

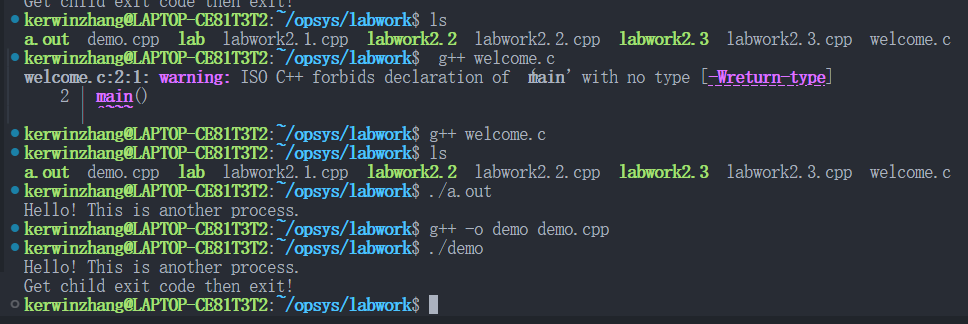
}子进程要加载的程序welcome.c：

#include<stdio.h>

Int main()

{ printf("Hello! This is another process.\n");}

编译及执行过程和运行结果截屏：



分析程序功能：

创建一个子进程并执行指定的外部可执行文件 /home/user/sy1/welcome.out。父进程会阻塞等待子进程的结束，并输出相应的消息。

具体执行过程如下：

主函数 main 开始执行，声明了变量 child 和 p，它们的类型都是 int。

在一个循环中，调用 fork 函数创建子进程。如果 fork 返回值为 -1，表示创建子进程失败，继续循环直到成功创建子进程。

如果 fork 返回值为 0，说明当前进程是子进程。在子进程中，使用 execl 函数执行外部可执行文件 /home/user/sy1/welcome.out，并传入空字符串作为命令行参数。然后调用 exit 函数退出子进程。

如果 fork 返回值不为 0，说明当前进程是父进程。进入 else 分支。

在父进程中，使用 waitpid 函数进行阻塞式等待子进程的结束。调用 waitpid 传入子进程的进程 ID child，并传入 0 作为选项，使父进程阻塞等待子进程结束。

当子进程结束后，waitpid 返回子进程的进程 ID child。判断 p 是否等于 child，如果相等，表示成功获取到子进程的退出状态，输出相应的消息。

如果 p 不等于 child，表示出现了错误，输出错误消息。

最后，调用 exit 函数退出程序。

5、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暂停10s之后退出，父进程先用阻塞方式等待子进程1的结束，然后用非阻塞方式等待子进程2的结束，待收集到二个子进程结束的信息，父进程就返回。给出编译及执行过程和运行结果截屏。（2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;

}

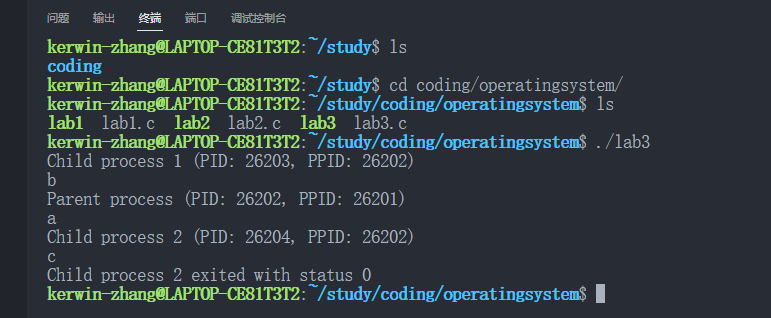
// 否则，继续等待

}

// 父进程结束

return 0;

}



6、编写一个简易的shell解释程序。其运行原理是：当命令行上有命令需要执行时，shell进程获得该命令，然后创建子进程，让子进程执行命令，shell进程等待子进程退出，之后继续等待命令行上的命令周而复始。（附加题）

#include <iostream>

#include <string>

#include <vector>

#include <sstream>

#include <unistd.h>

#include <sys/wait.h>

using namespace std;

int main() {

string input;

while (true) {

// 输出提示符并获取用户输入

cout << "kerwin shell begin to start!" << endl;

cout << "$ ";

getline(cin, input);

// 将用户输入拆分成命令和参数

istringstream iss(input);

vector<string> args;

string arg;

while (iss >> arg) {

args.push\_back(arg);

}

// 如果用户输入的是 "exit"，退出程序

if (args[0] == "exit") {

break;

}

// 创建子进程并执行命令

pid\_t pid = fork();

if (pid == 0) {

// 子进程

char\* argv[args.size() + 1];

for (int i = 0; i < args.size(); i++) {

argv[i] = const\_cast<char\*>(args[i].c\_str());

}

argv[args.size()] = nullptr; // 末尾添加一个空指针

execvp(argv[0], argv);

cerr << "Error: command not found" << endl;

exit(1);

} else if (pid > 0) {

// 父进程

int status;

waitpid(pid, &status, 0);

} else {

// 创建子进程失败

cerr << "Error: fork() failed" << endl;

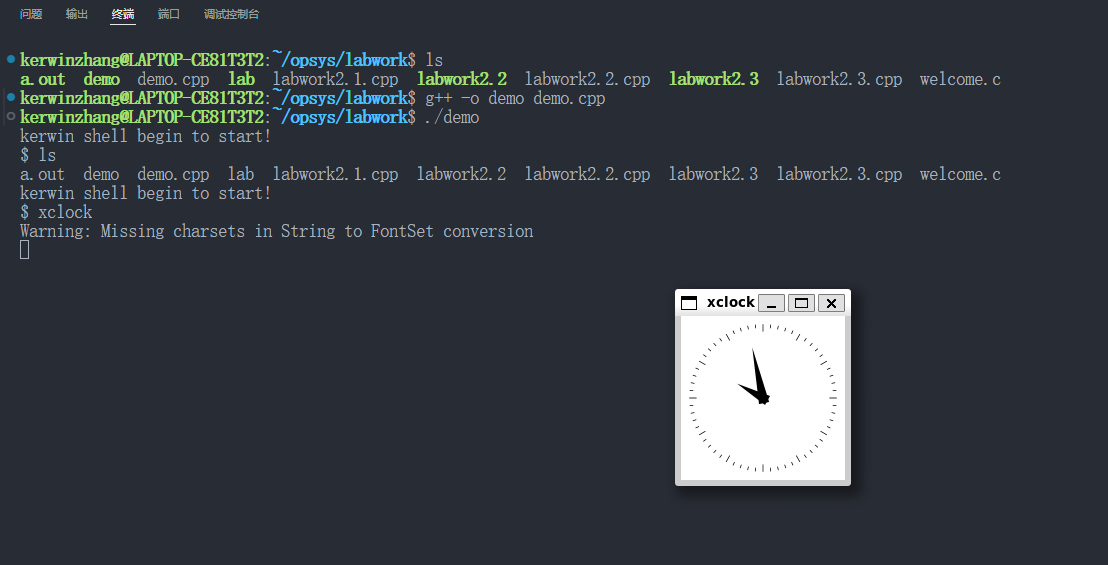
continue;

}

}

return 0;

}



三、实验总结和体会

**总 结**

**深入理解操作系统原理**

**通过实验,我对操作系统的进程管理、内存管理、文件系统等核心概念有了更加深入的理解,为后续的操作系统课程奠定了扎实的基础。**

**熟练掌握系统编程技能**

**在实验过程中,我反复练习使用 Linux 系统调用如 fork()、exec()、wait() 等,熟练掌握了系统编程的相关技能,为未来的工作打下坚实的基础。**

**提升代码调试能力**

**在实验过程中,我经常会遇到各种错误和问题,需要利用 GDB 等工具进行调试。这锻炼了我的代码调试能力,培养了我解决问题的思维方式。**

**增强动手实践能力**

**相比于单纯的理论学习,实践性实验让我能够将所学知识应用到实际编程中,培养了动手能力和解决实际问题的能力。**

**拓展编程语言视野**

**通过使用 GCC 编译器,我不仅学会了 C/C++ 语言的编程方法,还接触到了 Makefile、GDB 等多种编程工具,拓展了编程语言的视野。**

**体 验**

**使用 VMware 虚拟机进行系统编程实验,让我亲身感受到了虚拟化技术的优势。通过环境隔离和快照回滚等功能,我可以安全地进行实验操作,而不用担心对主机系统造成影响。这为我未来的工作和学习打下了良好的基础。**

**在虚拟机中使用不同的操作系统进行编程实验,让我受益匪浅。以前我只是使用其他系统来完成对应的任务,却没有深入了解系统的底层实现。但在这次实验中,通过阅读和使用各个操作系统提供的系统调用API,我对操作系统的概念有了更加清晰的认知。**

**例如,在 Linux 系统中,我学会了使用 fork()、exec()、wait() 等系统调用来管理进程,这让我对进程的概念和进程间通信有了更深入的理解。而在 Windows 系统中,我则接触到了像 CreateProcess()、WaitForSingleObject() 等API,发现了不同系统在设计理念和实现方式上的差异。**

**这种跨平台的实验体验,不仅拓宽了我的编程视野,也培养了我对操作系统底层机制的敏感性。我相信,这些宝贵的实践经验,将为我未来的工作和学习打下坚实的基础。**