山东大学 计算机科学与技术 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200130119 | 姓名：于斐 | | 班级：学堂计机22 |
| 实验题目： 进程控制 | | | |
| 实验学时： 4 | | 实验日期： 2024年3月18日 | |
| 实验目的：  加深对进程并发执行概念的理解，实践并发进程/线程的创建和控制方法，观察并体验进程的动态特性，进一步理解进程在其生命周期中创建、变换和撤销状态的过程，掌握进程控制的方法，了解父子进程之间的控制和协作关系，并练习在Linux系统中与进程/线程创建与控制相关的系统调用的编程和调试技术。 | | | |
| 实验过程中遇到和解决的问题：  实验过程中，在两个子进程的运行先后顺序控制部分遇到了困难。由于进程之间彼此独立，主进程难以通过传统的全局变量等方式控制子进程运行流程，需要找到进程间稳定通信的方法。该过程一旦控制不当，则会发生进程死锁，影响整个实验的进行。  最终想到使用 Signal 的方法控制。第一个子进程创建后直接调用 pause()，等待主进程发送的 SIGINT 信号。主进程在创建第二个子进程后等待其执行 ps 结束，之后向子进程发送 SIGINT，从而打断第一个子进程的 pause() 状态（过程中子进程的 SIGINT 信号被空函数处理），使第一个子进程执行 ls，从而达成实验目的。 | | | |
| 实验步骤与内容：  实验环境：Arch Linux x86\_64 with GCC 13.2.1-5  实验步骤：   1. 新建 main.c 文件，写入附录中的代码内容。在无限循环中，首先尝试创建第一个子进程，第一个子进程收到 SIGINT 信号后执行"/bin/ls"命令，第二个子进程执行"/bin/ps"命令。父进程在等待第二个子进程执行完毕后，向第一个子进程发送SIGINT信号，然后等待所有子进程结束后进入下一次循环。这个程序会一直执行，每次执行完毕后会休眠一段时间（由SLEEP\_INTERVAL定义，值为 3）。 2. 新建 main.h 文件，引入必要头文件，定义 SLEEP\_INTERVAL 为 3，并创建 empty\_handler 函数，函数为空。   #define SLEEP\_INTERVAL 3  void empty\_handler(int x) {}   1. 新建 Makefile 文件，在执行 make 时做编译工作，make clean 时清除所有 .o 及无扩展名程序文件。 2. 在程序目录下执行 make clean; make && ./main，观察实验结果。   实验结果如图，程序每隔三秒依次执行 /bin/ps 和 /bin/ls。 | | | |
| 实验总结：  通过本次实验，理解了进程的创建、状态控制和通信机制，加深了对操作系统中进程管理相关概念的认识。通过编写和运行实验程序，更加深入地理解了操作系统中进程的运行机制和实现方式。  通过实验，认识到进程具有动态性（具有生命周期）、并发性（宏观上多个进程同步运行）、独立性（进程资源几乎彼此独立）、异步性（若无控制，两个子进程以未知的顺序及进度运行）等特征。  生命周期：两个子进程运行结束即被销毁，资源被操作系统收回。  实体：每个进程都有一个对应的 PCB，用于存储进程的状态信息、程序计数器、寄存器值等。在 Linux 中，PCB 是由内核管理的数据结构，用于跟踪和管理进程的状态。每个进程都有自己独立的地址空间，包括代码段、数据段、堆和栈等。在 fork() 函数调用后，子进程会复制父进程的内存映像，但各自独立运行。  状态控制：进程在执行过程中会经历不同的状态，如就绪、运行、阻塞等。在实验中，父进程通过 wait() 函数等待子进程的结束，从而控制子进程的状态转换。进程可以通过信号机制来进行状态控制。在实验中，父进程向子进程发送 SIGINT 信号，从而终止子进程的执行。  子进程是通过调用 fork() 函数执行系统调用创建的。在子进程中，通过调用execve() 函数来加载并执行新的程序。  信号是在操作系统中用于通知进程发生了某些事件的一种机制。它可以用来实现进程间的通信、错误处理、以及对特定事件的响应。在实验中，使用了 SIGINT 信号来控制子进程的行为。具体来说，父进程在等待第二个子进程执行完毕后，向第一个子进程发送 SIGINT 信号，从而终止第一个子进程的执行。这种方式是通过信号来控制进程的行为，实现了对子进程的控制。 | | | |

附录：程序关键源代码

#include "main.h"

int main(int argc, char\* argv[]) {

    for (; ; ) {

        pid\_t pids[2];

        pids[0] = fork();

        if (pids[0] < 0) {

            perror("Error occurred while executing fork()");

            exit(EXIT\_FAILURE);

        }

        if (pids[0] == 0) {

            signal(SIGINT, empty\_handler);

            pause();

            char\* args[2] = {"/bin/ls", NULL};

            int status = execve("/bin/ls", args, NULL);

            if (status) {

                perror("Error occurred while executing ls");

                return EXIT\_FAILURE;

            }

            return EXIT\_SUCCESS;

        }

        int status;

        pids[1] = fork();

        if (pids[1] < 0) {

            perror("Error occurred while executing fork()");

            exit(EXIT\_FAILURE);

        }

        if (pids[1] == 0) {

            char\* args[2] = {"/bin/ps", NULL};

            int status = execve("/bin/ps", args, NULL);

            if (status) {

                perror("Error occurred while executing ps");

                return EXIT\_FAILURE;

            }

            return EXIT\_SUCCESS;

        }

        waitpid(pids[1], &status, 0);

        kill(pids[0], SIGINT);

        wait(NULL);

        sleep(SLEEP\_INTERVAL);

    }

}