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：201900130059 | 姓名： 孙奇 | | 班级： 2019级1班 |
| 实验题目：进程控制 | | | |
| 实验学时：2 | | 实验日期：2021/10/15 | |
| 实验目的：  加深对于进程并发执行概念的理解。实践并发进/线程的创建和控制方法。观  察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换  的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习 Linux 系  统中进/线程创建与控制有关的系统调用的编程和调试技术。 | | | |
| 硬件环境：  CPU: Intel i5-9300H  GPU: UHD630 | | | |
| 软件环境：  Ubuntu 18.04 | | | |
| 实验步骤与内容：   1. 进程实验说明：   进程可以通过系统调用 fork() 创建子进程并和其子进程并发执行。子进程初始的执行映像是父进程的一个副本：子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码、组代码、环境变量、已打开的文件描述符、工作目录和资源限制等。子进程可以通过系统调用族 exec() 装入一个新的执行程序。父进程可以使用wait() 或 waitpid() 系统调用等待子进程的结束并负责收集和清理子进程的退出状态。   1. 独立实验：   参考以上示例程序中建立并发进程的方法，编写一个多进程并发执行程序。父进程每隔 3 秒重复建立两个子进程，首先创建的子进程让其执行 ls 命令，之后创建的子进程让其执行 ps 命令，并控制 ps 命令总在 ls 命令之前执行。   1. 实现思路：   两层循环先后创建两个子进程’ls’和’ps’，子进程创建后pause()挂起，在父进程中依次调用kill先后唤醒子进程’ps’和’ls’，父进程waitpid()等待各子进程执行结束后，回收子进程资源，然后继续执行父进程，完成程序；   1. 执行过程：   2021-10-15 21-35-32 的屏幕截图 | | | |
| 结论分析与体会：   1. 实验反映出的**进程的特征和功能：** 2. 并发性：父进程与子进程、子进程之间可以同时执行，分别完成自己的任务； 3. 动态性：进程是动态产生、动态消亡的，同时进程的运行状态处于经常性的动态变化中； 4. 独立性：进程是调度的基本单位，在执行中是独立的，能够参与并发执行； 5. 交互性：进程在运行过程中可与其他进程发生直接间接的相互作用； 6. **真实操作系统中反映的进程生命期、实体、状态控制：** 7. 进程生命期：进程从创建到消亡的全过程；本实验中，子进程’ps’和’ls’创建后进入就绪态，很快获取CPU资源开始执行，在pause()后挂起，位于阻塞状态，等待唤醒，父进程kill()唤醒子进程后，子进程回到就绪态并迅速开始执行，直到执行结束； 8. 实体：程序段、数据段、PCB构成了一个进程实体，进程是进程实体的运行过程，本实验中’ps’，’ls’有进程实体，其执行过程是对应的进程； 9. 状态控制：本实验进程状态，fork()创建进程，进程位于创建态，创建的进程获取资源进入就绪态，就绪态的进程被CPU调度进入运行态，当pause()或其他请求等待事件发生，进程进入阻塞态，等待被唤醒，当遇到中断则进入就绪态，当进程被kill()唤醒后进入就绪态，然后被CPU调度继续运行，执行结束后，退出到终止态； 10. 进一步理解进程概念和并发概念： 11. 进程：进程实体的运行过程； 12. 并发：同一时间段多个进程实体同时运行； 13. 子进程的创建和执行原理： 14. 创建：fork()系统调用创建当前进程的子进程，根据返回的进程号判断是子进程还是父进程； 15. 执行原理：若进程号为0，则当前上下文为子进程，其拷贝了父进程的资源，执行其对应的程序段；若进程号>0，则当前为父进程，执行父进程对应的程序段； 16. 信号的机理：当一个信号发送给一个进程，那么任何非原子操作都会被中断，若进程定义了信号处理函数，那么它将被执行，否则执行默认的处理函数； 17. 信号实现的进程控制：在多进程并发的情形下，进程可以通过waitpid()等待某进程的执行结束，当对应子进程执行结束后，会通过信号通知父进程回收资源；同时当进程挂起时，其他进程可以通过kill()发送信号唤醒对应的进程，使其继续执行，多个进程间交互的函数都是利用信号来实现进程间的通信的； | | | |

附录：程序源代码

1. lab\_1.h:

#include <sys/types.h>

#include <wait.h>

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

void sigcat() {

printf("%d Process continue\n", getpid());

}

1. lab\_1.cpp:

#include "lab\_1.h"

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

int pidOfLs; // 子进程1--ls

int pidOfPs; // 子进程号2--ps

int status; // 子进程返回状态

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

// 注册函数sigcat用于处理键盘中断

signal(SIGINT, (sighandler\_t)sigcat);

perror("SIGINT");

// 循环四次

int loop = 4;

while (loop--) {

pidOfLs = fork();

if (pidOfLs < 0) {

// 建立子进程失败

printf("Create Process fail!\n");

exit(EXIT\_FAILURE);

}

else if (pidOfLs == 0) {

// 子进程

printf("=======================Child=Process=========================\n");

printf("I am 'ls' Process %d\nMy father is %d\n", getpid(), getppid());

pause();

// 子进程被中断信号唤醒

printf("%d 'ls' child will Running:\n", getpid());

// 输出要执行的命令

for (int i = 0; args[i] != NULL; i++)

printf("%s\n", args[i]);

// 执行命令行

status = execve(args[0], &args[1], NULL);

} else {

sleep(1);

// 父进程

printf("=======================Parent=Process==========================\n");

printf("\nI am Parent Process %d\n", getpid());

// 子进程2--ps

args[0] = "/bin/ps";

args[1] = "-l";

pidOfPs = fork();

if (pidOfPs < 0) {

// 创建子进程失败

printf("Create Process fail!\n");

exit(EXIT\_FAILURE);

} else if (pidOfPs == 0) {

// 子进程

printf("=======================Child=Process=========================\n");

printf("I am 'ps' Child Process %d\nMy father is %d\n", getpid(), getppid());

pause();

// 子进程被中断信号唤醒

printf("%d 'ps' child will Running\n", getpid());

// 输出要执行的命令

for (int i = 0; args[i] != NULL; i++)

printf("%s\n", args[i]);

// 执行命令行

status = execve(args[0], &args[1], NULL);

} else {

sleep(1);

// 父进程

printf("=======================Parent=Process==========================\n");

printf("\nI am Parent Process %d\n", getpid());

if(kill(pidOfPs, SIGINT) >= 0)

printf("%d wakeup %d child: 'ps'.\n", getpid(), pidOfPs);

printf("%d Waiting for 'ps' child done.\n\n", getpid());

waitpid(pidOfPs, &status, 0);

printf("\nChild of 'ps' done, status = %d\n\n", status);

args[0] = "/bin/ls";

args[1] = "-a";

}

if(kill(pidOfLs, SIGINT) >= 0)

printf("%d wakeup %d child: 'ls'.\n", getpid(), pidOfLs);

printf("%d Waiting for 'ls' child done.\n\n", getpid());

waitpid(pidOfLs, &status, 0);

printf("\nChild of 'ls' done, status = %d\n\n", status);

}

sleep(3);

}

}