**鲁东大学信息与电气工程学院**

**实验报告**

（ 2022 — 2023 学年第 1 学期）

2220180106

**课程名称**  操作系统

**实验题目** 进程控制与进程调度

**专 业** 软件工程

**班 级**  2101

**姓 名**  梅英豪

**学 号**  20212203341

2023 年 4 月 8 日

|  |  |  |  |
| --- | --- | --- | --- |
| 实验题目 | 实验2 进程控制与进程调度 | | |
| 实验类型 | 验证、设计 | 实验日期 | 2023 年 3 月 25 日 |
| 题目来源 | ✔1.必修 2.选修 3.自拟(设计) 4.专题 | | |
|  | | | | |
| 七、指导教师评语及成绩  教师签名：  **年 月 日**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 优秀 | 良好 | 中等 | 及格 | 不及格 | | | | | |

**一、实验目的及要求**

1．加深对进程概念的**理解**，进一步认识并发执行的实质；

2．**掌握**Linux操作系统中进程的创建和终止操作；

3．**掌握**在Linux操作系统中创建子进程并加载新映像的操作；

4．加深对进程概念的**理解**，明确进程和程序的区别；

5．深入**理解**系统如何组织进程；

6．**理解**常用进程调度算法的具体实现；

7．加深对死锁概念的**理解**，进一步认识死锁的预防、避免和检测手段。**二、实验仪器设备与软件环境**

1．一台装有Windows的微型计算机（Win7或Win10）；

2．在微型计算机上（通过虚拟机）安装了Linux环境；

3．Linux环境下的编辑器（vi、gedit或者vscode），编译器（gcc），调试器（gdb）。**三、实验基础（一）进程控制部分**

**1、LINUX 进程**

与传统的进程一致，Linux进程也主要有 3部分组成：程序段、数据段和进程控制块。

程序段存放进程执行的指令代码，具有可读、可执行、不可修改属性，但允许系统中多个进程共享这一代码段，因此程序与进程具有一对多的属性。

数据段是进程执行时直接操作的所有数据（包括变量在内），具有可读、可写、不可执行属性。

Linux 中每个进程 PCB的具体实现用一个名为 task\_struct的数据结构来表示，在 Linux 内核中有个默认大小为 512B的全局数组 task，该数组的元素为指向 task\_struct结构的指针。在创建新进程时，Linux 将会在系统空间中分配一个 task\_struct 结构，并将其首地址加入到 task 数组。当前正在运行的进程的 task\_struct 结构由一个 current指针来指示。

**2、所涉及的命令**

（1）ps：查看用户空间的当前进程。

（2）top ：实时的对系统处理器的状态监视。

（3）pstree -h ：列出进程树并高亮标出当前进程。

（4）vmstat：对系统的虚拟内存、进程、CPU活动进行监视。

（5）strace：监视用户空间程序发出的全部系统调用。

（6）ltrace：解释并记录执行程序调用的动态链接库以及进程接收到的信号。

可用 ltrace -f -i -S ./executable-file-name 查看指定程序的执行过程。

（7） sleep x：睡眠指定时间，x 为指定睡眠的秒数。

（8）kill [-9] PID ：结束或终止 process ID 进程。

（9）kill %n ：终止在 background中的第 n 个 job。

（10）jobs：查看正在 background 中执行的 process。

**3、所涉及的系统调用**

**1) 进程的创建使用 fork系统调用，其函数原型为：**

#include <unistd.h>

#include <sys/types.h>

pid\_t fork(void);

若fork调用成功，就会创建一个新的进程，新创建的进程称为子进程，调用fork的进程称为父进程。子进程拷贝了父进程的代码段、数据段、堆栈段等，因此子进程是父进程的复制品。此后，父子进程就并发执行，都从fork调用之后的语句开始执行。

fork调用返回的是pid\_t类型的pid值，其中pid\_t是个有符号的整型量，若调用成功，内核会将控制返回给父子进程，在父进程中返回的是新建子进程的进程识别号pid，在子进程中返回 0值，程序员可以据此来区分父子进程。若调用出错，则返回-1，表示创建子进程失败。

**2) 获取进程的 PID**

系统中当前存在的每个进程都有一个非负整数的唯一进程 ID，用来唯一地标识一个进程，称为PID，系统内核函数可通过PID来引用PCB。当然在一个进程终止后，其PID还可用作其它进程的识别号。此外，每个进程还包括用户ID（UID）和组ID（GID），用来确定进程对系统中文件和设备的相关存取权限。

系统将进程的识别号 0 和 1 保留给了两个重要的进程：其中进程 ID为 0 的进程是调度进程，又称系统进程或交换进程（swapper），该进程是内核的一部分，按照一定的原则为进程分配处理机。进程 ID 为 1 的进程即为 init 进程，该进程是一个用户进程，是/sbin/init 程序的执行，该进程是所有其它用户进程的祖先，且系统中的所有孤儿进程都由其收养。

得到当前进程的 PID。函数原型： int getpid()

得到当前进程的父进程的 PID。函数原型： int getppid()

**3) system()：在程序中运行一个命令**

程序中调用 system函数时，用字符串参数传递一个 shell 命令，并执行

函数原型： #include <stdio.h>

int system(char \*string)

调用举例：system(“ps -af”);

**4) exec( )和 fork( )联合使用**

系统调用exec和fork( )联合使用能为程序开发提供有力支持。用fork( )建立子进程，然后在子进程中使用exec( )，这样就实现了父进程与一个与它完全不同子进程的并发执行。

一般，wait、exec联合使用的模型为：

int status;

............

if (fork( )= =0)

{

...........;

execl(...);

...........;

}

wait(&status);

**5) exit（）**

终止进程的执行。

系统调用格式：

void exit(status)

int status;

其中，status是返回给父进程的一个整数，以备查考。

**4、进程的控制**

进程因创建而存在，因执行完成或异常原因而终止.在进程的生命周期中，进程在内存中有三种基本状态:就绪，执行，阻塞.进程状态的转换是通过进程控制原语来实现的。

Linux操作系统提供了相应功能的系统调用及命令，来实现用户层的进程控制。**（二）进程调度部分**

无论是在批处理系统还是分时系统中，用户进程数一般都多于处理机数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用处理机。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程，以使之执行。

进程调度选择一个可用的进程（可能从多个可用进程集合中选择）到 CPU 上执行。它是操作系统设计的中心问题之一。

进程调度算法有先进先出、短进程优先、简单轮转法、多级队列法和多级反馈队列等。

* 先进先出：算法总是把处理机分配给最先进入就绪队列的进程，一个进程一旦分得处理机，便一直执行下去，直到该进程完成或阻塞时，才释放处理机。容易引起作业用户不满，常作为一种辅助调度算法。
* 进程优先：该算法从就绪队列中选出下一个“CPU 执行期最短”的进程，为之分配处理机。该算法虽可获得较好的调度性能，但难以准确地知道下一个 CPU 执行期，而只能根据每一个进程的执行历史来预测。
* 简单轮转法：系统将所有就绪进程按FIFO 规则排队，按一定的时间间隔把处理机分配给队列中的进程。这样，就绪队列中所有进程均可获得一个时间片的处理机而运行。
* 多级队列法：将系统中所有进程分成若干类，每类为一级。
* 多级反馈队列：在系统中设置多个就绪队列，并赋予各队列以不同的优先权。**（三）死锁**

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

产生死锁的原因：

1、因为系统资源不足。

2、进程运行推进的顺序不合适。

3、资源分配不当。

产生死锁的条件有四个：

1、互斥条件：所谓互斥就是进程在某一时间内独占资源。

2、请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3、不剥夺条件:进程已获得资源，在末使用完之前，不能强行剥夺。

4、循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。

1、采用资源静态分配策略，破坏"部分分配"条件；

2、允许进程剥夺使用其他进程占有的资源，从而破坏"不可剥夺"条件；

3、采用资源有序分配法，破坏"环路"条件。

死锁的避免不严格地限制死锁的必要条件的存在，而是系统在系统运行过程中小心地避免死锁的最终发生。最著名的死锁避免算法是银行家算法。四、实验内容

本次实验任务有三个部分：进程的创建与终止、进程的调度、死锁（银行家算法）。（一）**进程的创建与终止**

1．编写一个C程序，并使用系统调用fork()创建一个子进程。要求如下：①在子进程中分别输出当前进程为子进程的提示、当前进程的PID和父进程的PID、根据用户输人确定当前进程的返回值、退出提示等信息。②在父进程中分别输出当前进程为父进程的提示、当前进程的PID和子进程的PID、等待子进程退出后获得的返回值、退出提示等信息；

2．编写另一个C程序，使用系统调用fork()以创建一个子进程，并使用这个子进程调用exec函数族以执行系统命令ls。（二）**进程调度**

编写C程序，模拟实现单处理器系统中的进程调度算法，实现对多个进程的模拟调度，要求分别模拟实现先到先服务算法 FIFO、时间片轮转调度算法 RR、最短作业优先算法 SJF、优先级调度算法 PRIOR。（三）**死锁**

1．创建两个进程分别只有 A 和 B 其中的一个锁，并互相等待对方的锁。查看执行结果，并分析原因。

2．编写银行家算法程序：

（1）输入各个进程的最大需求资源、已分配资源、系统可用资源并有序的显示出来；

（2）输入申请的进程号和对应的要求，并对其进行资源请求算法的判断；

（3）在上面判断合法的前提下进行试分配，利用安全性算法求出安全序列；

（4）如果存在不安全，也即是存在死锁，找到占用最多资源的进程，并解除占用。五、实验过程（80分）

本实验要求学生按照以下操作步骤（含代码、数据、图表等）完成实验，并提交实验结果截图。（一）进程的创建与终止（20分）

1．使用系统调用fork()创建一个子进程

本实验属于验证型实验，主要目的是验证fork()的返回值。首先在主程序中通过fork()创建子进程，并根据fork()的返回值确定所处的进程是子进程还是父进程，然后分别在子进程和当前进程(父进程)中调用getpid()、getppid()、wait()等函数以完成实验内容。

下面给出实验内容(1)的示例代码，其中涉及两个函数sleep()和WEXITSTATUS()。sleep() 函数的作用是让进程挂起一段时间，以秒为单位。WEXITSTATUS()函数用来判断子进程是不是正常退出的：如果是，就返回一个非零值(取子进程传送给exec函数族的参数的低8位)；如果子进程不是正常退出的，则返回0。

#include <unistd.h>

#include <sys/types.h>

#include <errno.h>

#include <sys/wait.h>

#include <stdlib.h>

int main()

{

pid\_t childpid;/\*子进程的PID\*/

int retval;/\*由用户提供的子进程返回值\*/

int status;/\*子进程向父进程提供的退出状态\*/

/\*创建一个新进程\*/

childpid=fork();

if(childpid>=0)//创建成功

{

if (childpid==0)//fork()的返回值为0，这表示当前处在子进程中

{

printf("CHILD：I am the child process!\n");

printf("CHILD：Here's my PID：%d\n", getpid());

//输出当前进程的PID

Printf("CHILD：My parent's PID is：%d\n", getppid());

//输出当前进程的父进程的PID

printf("CHILD：The value of fork return is：d\n",childpid);

//输出fork() 的返回值

printf("CHILD：Sleep for 1 second...\n");

sleep(1); //让当前进程睡眠1秒

printf("CHILD：Enter an exit value (0~255)：");

//输人子进程执行完毕后的返回值

scanf("%d", &retval);

printf("CHILD：Goodbye!\n");

exit(retval); //子进程退出，退出值为用户给定的返回值

}

else //fork() 返回一个新的PID，这表示当前处在父进程中

{

printf("PARENT：I am the parent process!\n");

printf("PARENT：Here's my PID：%d\n", getpid());

//输出当前进程的PID

printf("PARENT：The value of my child's PID is：%d\n",childpid);

//输出当前进程的子进程的PID

printf("PARENT：I will now wait for my child to exit.\n");

wait(&status); //等待子进程运行结束，并保存其状态

printf("PARENT：Child's exit code is：%d\n",WEXITSTATUS (status));

//输出子进程的返回值

printf("PARENT：Goodbye!\n");

exit(0); //父进程退出

}

}

else //fork()返回-1，这表示进程创建失败

{

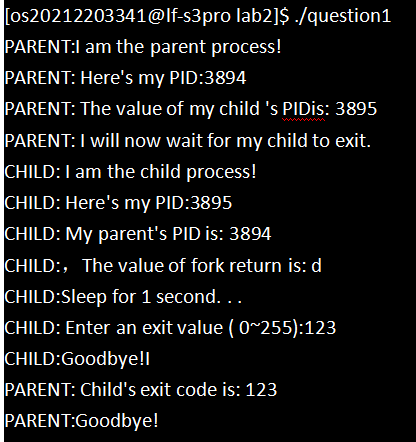
perror("fork error!"); //显示错误信息

exit(0);

}

}

实验结果：（5分）



2．使用系统调用fork()以创建一个子进程，并使用这个子进程调用exec函数族以执行系统命令ls

学生在实验内容1的基础上，在fork()调用之后于子进程中使用exec函数族执行命令Is。具体请参考exec函数族的帮助页面。

实验代码如下：（10分）

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

int status;

pid = fork();

if (pid < 0) { // error occurred

printf("Fork failed\n");

return 1;

} else if (pid == 0) { // child process

printf("Child process\n");

execl("/bin/ls", "ls", "-l", NULL);

printf("exec failed\n"); // this line should not be reached

exit(1);

} else { // parent process

printf("Parent process\n");

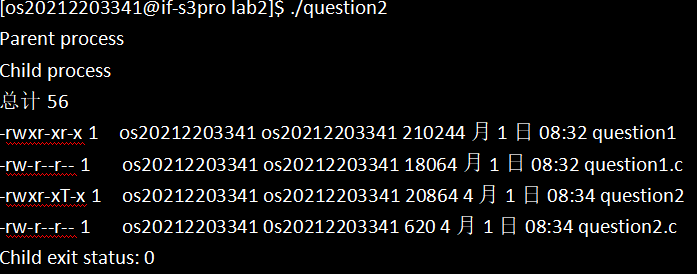
wait(&status);

printf("Child exit status: %d\n", WEXITSTATUS(status));

}

return 0;

实验结果：（5分）



（二）进程调度（30分）

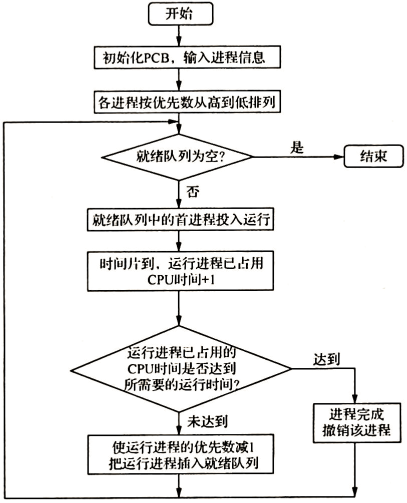
本实验首先需要为每个进程设计一个进程控制块。进程控制块可以根据具体的调度算法来确定自身所须包含的信息，如进程名、优先数、到达时间、需要运行的时间、已用CPU 时间、进程状态等。进程控制块用C语言中的结构体来表示。其次需要设计就绪队列，一般用链表表示，具体到C语言中则用指针来实现。然后需要实现具体的调度算法，这里涉及多种操作，如排序操作、链表操作等。最后需要设计程序的输出方式，在输出结果时，既可以输出调度进程的顺序以及每个进程的起始时间、终止时间等，又可以输出CPU每次调度的过程。

1．优先级调度算法的实现

下面给出采用基于动态优先数的进程调度算法，优先数大者优先，且优先数每运行一个CPU时间单位就降低一级(即将优先数减1)。相比固定优先数调度算法，该算法稍显复杂。具体运行情况描述如下。

进程的优先数及需要的运行时间事先已人为指定。进程的运行时间以1个CPU时间单位为单位进行计算。每个进程可以处于W(就绪)、R(运行)、F(完成)三种状态之一。就绪进程在获得CPU后只能运行一个CPU时间单位。运行后，将进行控制块中已占用CPU时间加1。如果运行一个CPU时间单位后，进程的已占用CPU时间达到了所需要的运行时间，则撤销该进程。如果运行一个CPU时间单位后，进程的已占用CPU时间尚未达到所需要的运行时间，即进程还需要继续运行，则此时将进程的优先数减1(即降低一级)，然后把它插入就绪队列并等待CPU。

每进行一次调度，程序就输出一次运行进程和就绪队列中所有进程的信息，以便进行检查。重复以上过程，直到所有进程运行完毕为止。

示例调度程序的流程图如下图所示。

代码如下：

#include <stdio.h>

#include <stdlib.h>

#define getpch(type) (type\*)malloc(sizeof(type))

struct pcb { /\*定义进程控制块\*/

char name[10]; //进程名

char state; //进程状态："w"表示就绪，"R"表示运行

int nice; //进程优先数

int ntime; //需要运行的时间

int rtime; //已经运行的时间

struct pcb\* link;

} \*ready=NULL, \*p;

typedef struct pcb PCB;

char sort() /\*进程优先数排序函数，优先数大者优先，并生成就绪队列\*/

{

PCB \*first, \*second;

int insert=0;

if((ready==NULL)||((p->nice)>(ready->nice)))/\*优先数最大者插人队首\*/

{

p->link=ready;

ready=p;

}

else /\*对进程比较优先数，并调整它们的位置\*/

{

first=ready;

second=first->link;

while(second!=NULL)

{

if((p->nice) >(second->nice)) /\*若插入的进程比当前进程的优先数大\*/

{ /\*插到当前进程的前面\*/

p->link=second;

first->link=p;

second=NULL;

insert=1;

}

else /\*若插入的进程优先数最小，则插到队尾\*/

{

first=first->link;

second=second->link;

}

}

if(insert==0) first->link=p;

}

}

char input() /\*输人各个进程参数，建立进程控制块并排序生成就绪队列\*/

{

int i, num;

printf("\n请输人被调度的进程数目：");

scanf("%d", &num);

for(i=0; i<num; i++)

{

printf("\n进程号No.%d：", i);

p=getpch(PCB);

printf("\n输人进程名：");

scanf("%s", p->name);

printf("输入进程优先数：");

scanf("%d", &p->nice);

printf("输入进程运行时间：");

scanf("%d", &p->ntime);

printf("\n");

p->rtime=0;

p->state='W';

P->link=NULL;

sort(); /\*调用sort() 函数\*/

}

}

int space()/\*链表中节点个数的统计函数\*/

{

int l=0;

PCB\* pr=ready;

while(pr!=NULL)

{

l++;

pr=pr->link;

}

return(l);

}

char disp(PCB \* pr) /\*进程显示函数，用于显示当前进程\*/

{

printf("\n qname \t state\t nice\t ntime\t runtime\n");

printf("%s\t", pr->name);

printf("%c\t", pr->state);

printf("%d\t", pr->nice);

printf("%d\t", pr->ntime);

printf("%d\t", pr->rtime);

printf("\n");

}

char check() /\*进程查看函数\*/

{

PCB\* pr;

printf("\n\*\*\*\*当前正在运行的进程是：%s", p->name); /\*显示当前运行的进程\*/

disp(p);

pr=ready;

if (pr!=NULL)

printf("\n\*\*\*\*当前就绪队列状态为："); /\*显示就绪队列的状态\*/

else

printf("\n \*\*\*\*当前就绪队列状态为：空\n");/\*显示就绪队列的状态为空\*/

while(pr!=NULL)

{

disp(pr);

pr=pr->link;

}

}

char destroy() /\*建立进程撤销函数(进程运行结束，撤销进程)\*/

{

printf("进程[%s] 已完成.\n", p->name);

free(p);

}

char running() /\*建立进程就绪函数(进程运行时间到，设置进程处于就绪状态)\*/

{

(p->rtime) ++;

if(p->rtime==p->ntime)

destroy(); /\*调用destroy() 函数\*/

else

{

(p->nice)--;

p->state='W';

sort(); /\*调用sort() 函数\*/

}

}

int main()/\*主函数\*/

{

int len, h=0;

char ch;

input();

len=space();

while((len!=0)&&(ready!=NULL))

{

ch=getchar();

h++;

printf("\n The execute number：%d\n", h);

p=ready;

ready=p->link;

p->link=NULL;

p->state='R';

check();

running();

printf("\n按任意键继续......");

ch=getchar();

}

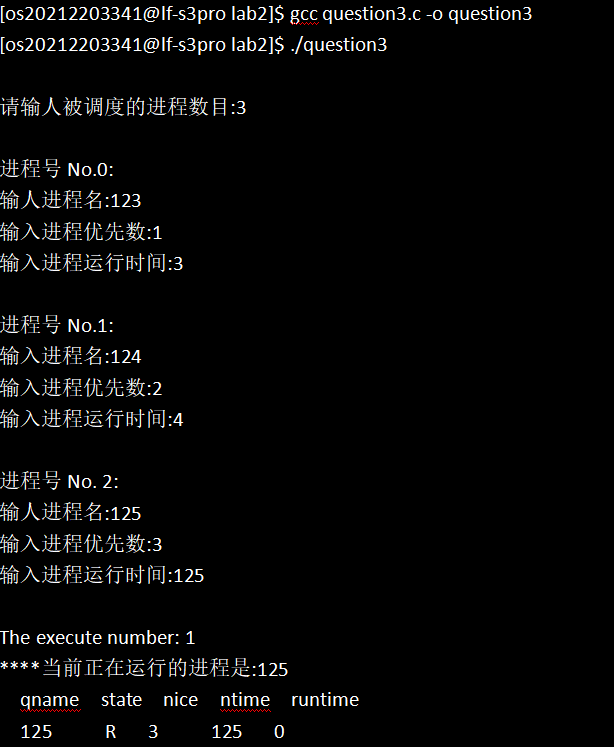
printf("\n\n所有进程已经运行完成!\n");

ch=getchar();

return 0;

}

实验结果：（4分）





2．先来先服务算法的实现（8分）

代码如下：

#include <stdio.h>

#define MAX\_PROCESS 10

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int waiting\_time;

int turnaround\_time;};

int main() {

struct Process processes[MAX\_PROCESS];

int n, i;

printf("Enter the number of processes: ");

scanf("%d", &n);

printf("Enter the arrival time and burst time for each process:\n");

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

printf("Process %d: ", i + 1);

scanf("%d %d", &processes[i].arrival\_time, &processes[i].burst\_time);

processes[i].pid = i + 1;

}

// sort the processes by arrival time

for (i = 0; i < n - 1; i++) {

int j, min\_index = i;

for (j = i + 1; j < n; j++) {

if (processes[j].arrival\_time < processes[min\_index].arrival\_time) {

min\_index = j;

}

}

if (min\_index != i) {

struct Process temp = processes[i];

processes[i] = processes[min\_index];

processes[min\_index] = temp;

}

}

// calculate completion time, waiting time, and turnaround time

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

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

if (i == 0) {

processes[i].completion\_time = processes[i].arrival\_time + processes[i].burst\_time;

} else {

processes[i].completion\_time = processes[i - 1].completion\_time + processes[i].burst\_time;

}

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

total\_waiting\_time += processes[i].waiting\_time;

total\_turnaround\_time += processes[i].turnaround\_time;

}

// print the results

printf("\nPID\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",

processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time,

processes[i].completion\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage waiting time: %.2f\n", (float) total\_waiting\_time / n);

printf("Average turnaround time: %.2f\n", (float) total\_turnaround\_time / n);

return 0;

}

实验结果：



3．时间片轮转调度算法实现（9分）

代码如下：

#include <stdio.h>

#define MAX\_PROCESS 10#define TIME\_QUANTUM 2

struct process {

int pid; // 进程ID

int arrival\_time; // 到达时间

int burst\_time; // 执行时间

int remaining\_time; // 剩余执行时间

int finish\_time; // 完成时间

int turnaround\_time;// 周转时间

int waiting\_time; // 等待时间};

int main() {

int n, i, j, time = 0, completed = 0;

struct process p[MAX\_PROCESS];

float total\_waiting\_time = 0, total\_turnaround\_time = 0;

printf("请输入进程数量：");

scanf("%d", &n);

// 获取每个进程的到达时间和执行时间

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

printf("请输入进程%d的到达时间和执行时间：", i+1);

scanf("%d %d", &p[i].arrival\_time, &p[i].burst\_time);

p[i].pid = i+1;

p[i].remaining\_time = p[i].burst\_time;

}

// 时间片轮转调度

printf("\n时间片轮转调度结果：\n");

printf("进程ID 到达时间 执行时间 完成时间 周转时间 等待时间\n");

while (completed < n) {

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

if (p[i].remaining\_time > 0) {

if (p[i].remaining\_time <= TIME\_QUANTUM) {

time += p[i].remaining\_time;

p[i].remaining\_time = 0;

} else {

time += TIME\_QUANTUM;

p[i].remaining\_time -= TIME\_QUANTUM;

}

if (p[i].remaining\_time == 0) {

completed++;

p[i].finish\_time = time;

p[i].turnaround\_time = p[i].finish\_time - p[i].arrival\_time;

p[i].waiting\_time = p[i].turnaround\_time - p[i].burst\_time;

if (p[i].waiting\_time < 0) p[i].waiting\_time = 0;

total\_waiting\_time += p[i].waiting\_time;

total\_turnaround\_time += p[i].turnaround\_time;

printf(" %d %d %d %d %d %d\n",

p[i].pid, p[i].arrival\_time, p[i].burst\_time, p[i].finish\_time,

p[i].turnaround\_time, p[i].waiting\_time);

}

}

}

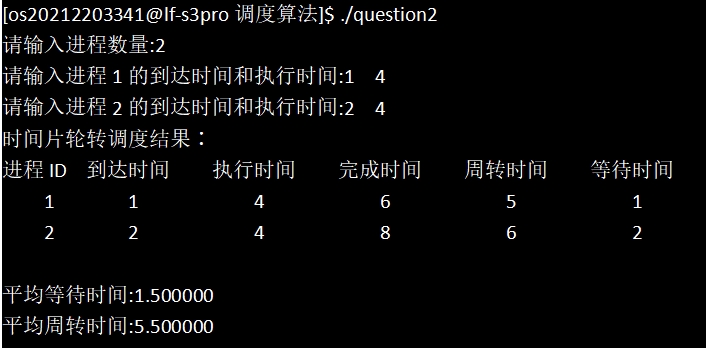
}

printf("\n平均等待时间：%f\n", total\_waiting\_time / n);

printf("平均周转时间：%f\n", total\_turnaround\_time / n);

return 0;}

实验结果：



4．最短时间优先算法的实现（9分）

代码如下：#include <stdio.h>

struct process {

int pid; // 进程编号

int arrival\_time; // 到达时间

int burst\_time; // 执行时间

int waiting\_time; // 等待时间

int turnaround\_time; // 周转时间

int completed; // 是否已完成};

int main() {

int n, i, j, min\_burst\_time, min\_burst\_index, time\_elapsed = 0;

float total\_waiting\_time = 0, total\_turnaround\_time = 0;

struct process processes[50];

printf("请输入进程数量：");

scanf("%d", &n);

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

printf("请输入进程 %d 的到达时间和执行时间：", i + 1);

scanf("%d %d", &processes[i].arrival\_time, &processes[i].burst\_time);

processes[i].pid = i + 1;

processes[i].completed = 0;

}

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

// 找出还未完成的进程中，执行时间最短的进程

min\_burst\_time = 9999;

min\_burst\_index = -1;

for (j = 0; j < n; j++) {

if (processes[j].arrival\_time <= time\_elapsed && !processes[j].completed && processes[j].burst\_time < min\_burst\_time) {

min\_burst\_time = processes[j].burst\_time;

min\_burst\_index = j;

}

}

// 如果找到了一个可以执行的进程，更新其信息

if (min\_burst\_index != -1) {

processes[min\_burst\_index].completed = 1;

processes[min\_burst\_index].waiting\_time = time\_elapsed - processes[min\_burst\_index].arrival\_time;

processes[min\_burst\_index].turnaround\_time = processes[min\_burst\_index].waiting\_time + processes[min\_burst\_index].burst\_time;

total\_waiting\_time += processes[min\_burst\_index].waiting\_time;

total\_turnaround\_time += processes[min\_burst\_index].turnaround\_time;

time\_elapsed += processes[min\_burst\_index].burst\_time;

}

// 如果未找到可以执行的进程，时间加1，继续循环

else {

time\_elapsed++;

}

}

printf("进程编号\t到达时间\t执行时间\t等待时间\t周转时间\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

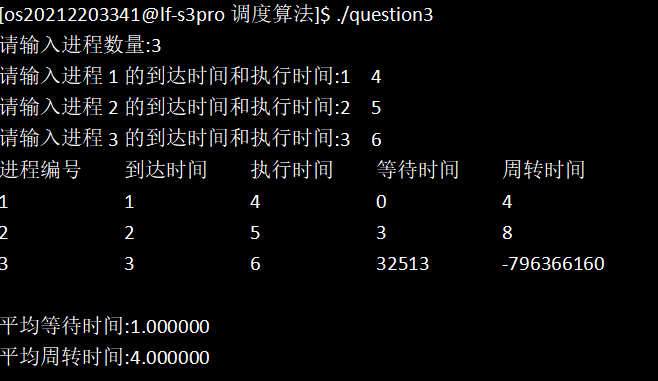
printf("\n平均等待时间：%f\n", total\_waiting\_time / n);

printf("平均周转时间：%f\n", total\_turnaround\_time / n);

return 0;

}

实验结果：



（三）死锁（30分）

1．创建两个进程分别只有 A 和 B 其中的一个锁，并互相等待对方的锁（8分）

代码如下：#include <stdio.h>#include <unistd.h>#include <fcntl.h>

int main() {

int a\_lock, b\_lock;

pid\_t pid;

// 创建锁文件

a\_lock = open("a.lock", O\_CREAT | O\_RDWR, 0666);

b\_lock = open("b.lock", O\_CREAT | O\_RDWR, 0666);

// 在子进程中获取 A 锁，并等待 B 锁

pid = fork();

if (pid == 0) {

printf("子进程获取 A 锁...\n");

flock(a\_lock, LOCK\_EX);

printf("子进程等待 B 锁...\n");

flock(b\_lock, LOCK\_EX);

printf("子进程获取 B 锁，完成！\n");

}

// 在父进程中获取 B 锁，并等待 A 锁

else {

printf("父进程获取 B 锁...\n");

flock(b\_lock, LOCK\_EX);

printf("父进程等待 A 锁...\n");

flock(a\_lock, LOCK\_EX);

printf("父进程获取 A 锁，完成！\n");

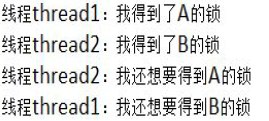
}

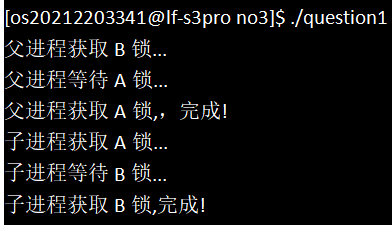
return 0;

}

实验结果：

（样例：）显示类似以下的结果：





2．银行家算法程序（22分）

该程序是在银行家算法的基础上添加了死锁的解除模块得来的，死锁的解除采用的方法是：当系统发生死锁时，找到已分配资源最大的死锁进程，剥夺其已分配资源，再次检测是否发生死锁。

（1）算法思想

先对用户提出的请求进行合法性检查，即检查请求是否大于需要的，是否大于可利用的。若请求合法，则进行预分配，对分配后的状态调用安全性算法进行检查。若安全，则分配；若不安全，则查找出占用资源最多的进程，剥夺其已分配资源，再次检测是否发生死锁。

（2）资源请求算法步骤

①如果Request＜or=Need,则转向步骤②；否则，认为出错，因为它所需要的资源数已超过它所宣布的最大值。

②如果Request＜or=Available,则转向步骤③；否则，表示系统中尚无足够的资源，进程必须等待。

③系统试探把要求的资源分配给进程 Pi，表并修改下面数据结构中的数值：

Available=Available-Request;

Allocation=Allocation+Request;

Need=Need-Request;

④系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。

（3）安全性算法步骤

①设置两个向量

* Work。它表示系统可提供进程继续运行所需要的各类资源数目，执行安全算法开始时，Work=Available;
* Finish。它表示系统是否有足够的资源分配给进程，使之运行完成，开始时先做Finish[i]=false，当有足够资源分配给进程时，令 Finish[i]=true。

②查找这样的 i 使其满足：

Finish[i]=false

Need<or=Work

如找到，执行步骤③；否则，执行步骤④。

③当进程获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work=Work+Allocation;

Finish[i]=true;

转向步骤②。

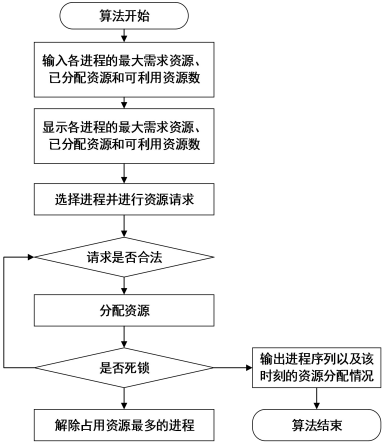
④如果所有进程的 Finish[i]=true,则表示系统处于安全状态；否则，系统处于不安全状态。

数据结构：

* 可用资源向量 available: 这是一个含有 m个元素的数组，其中的每一个元素代表一类可利用资源数目。
* 最大需求矩阵 max 它是一个的矩阵，定义了系统中 n 个进程中得每一个进程对类资源的最大需求。
* 可分配矩阵 allocation: 这也一个的矩阵，定义了系统中每一类资源当前已分配给每一进程的资源数。
* 需求矩阵 need: 这表示每一个进程尚需的各类资源数。
* need[i][j]=max[i][j]-allocation[i][j]。

变量说明：

* 可用资源向量 available[3];
* 最大需求矩阵 max[4][3];
* 可分配矩阵 allocation[4][3];
* 需求矩阵 need[4][3];
* 进程状态标识 finish[4]。

（4）算法流程图

代码如下：

#include <bits/stdc++.h>

using namespace std;

#define N 100

#define M 100

int Available[M], // 每种资源现有的实例数量

Max[N][M], // 每个进程i的对资源j最大需求

Allocation[N][M], // 每个进程现在所分配的每个资源的实例数量

Need[N][M], // 每个进程还需要的剩余的资源

Request[N][M], // 进程资源请求

sign[N], // 记录安全序列

process\_num, // 进程数量

recourse\_num; // 资源数量

void init() // 初始化进程资源状况

{

cout << "请输入进程数和资源数：";

cin >> process\_num >> recourse\_num;

for (int i = 0; i < process\_num; i++)

{

for (int j = 0; j < recourse\_num; j++)

{

cin >> Allocation[i][j];

}

}

for (int i = 0; i < process\_num; i++)

{

for (int j = 0; j < recourse\_num; j++)

{

cin >> Max[i][j];

Need[i][j] = Max[i][j] - Allocation[i][j];

}

}

for (int j = 0; j < recourse\_num; j++)

{

cin >> Available[j];

}

}

void print\_information() // 打印进程资源表

{

cout << "-----------------------------------进程资源表------------------------------------" << endl;

cout << "| 进程名称 | 已占用资源 | 尚需资源 | 最大所需资源 | 资源剩余实例 |" << endl;

cout << "|" << setw(16) << "|";

for (int j = 0; j < 4 \* recourse\_num; j++)

{

cout << setw(4) << char(j % recourse\_num + 'A');

if ((j + 1) % recourse\_num == 0)

cout << " |";

}

cout << endl;

int k = 0;

for (int i = 0; i < process\_num; i++)

{

cout << "|" << setw(10) << "P[" << i << "]"; // 进程名称

cout << " |";

for (int j = 0; j < recourse\_num; j++)

{

cout << setw(4) << Allocation[i][j]; // 已占用资源

}

cout << " |";

for (int j = 0; j < recourse\_num; j++)

{

cout << setw(4) << Need[i][j]; // 尚需资源

}

cout << " |";

for (int j = 0; j < recourse\_num; j++)

{

cout << setw(4) << Max[i][j]; // 最大所需资源

}

if (k == 0)

{

cout << " |";

for (int j = 0; j < recourse\_num; j++)

{

cout << setw(4) << Available[j]; // 资源剩余实例

}

cout << " |" << endl;

k = 1;

}

else

cout << " |" << setw(16) << "|" << endl;

}

cout << "---------------------------------------------------------------------------------" << endl

<< endl;

}

int security\_test() // 安全性检查

{

int work[recourse\_num];

bool finish[process\_num];

int count, k = 0;

for (int i = 0; i < recourse\_num; i++) // 初始化work

{

work[i] = Available[i];

}

for (int i = 0; i < process\_num; i++) // 设置false

{

finish[i] = false;

}

for (int i = 0; i < process\_num; i++) // 满足条件释放资源，并从头开始扫描进程集合

{

count = 0;

for (int j = 0; j < recourse\_num; j++)

if (finish[i] == false && Need[i][j] <= work[j])

count++;

if (count == recourse\_num) // 当进程各类资源都满足NEED<=WORK时

{

for (int j = 0; j < recourse\_num; j++)

work[j] = work[j] + Allocation[i][j]; // 释放进程

finish[i] = true;

sign[k] = i; // 记录下满足条件的进程

k++;

i = -1; // 从头开始扫描

}

}

for (int i = 0; i < process\_num; i++)

if (finish[i] == false)

return 0;

return 1;

}

int Banker\_Algorithm(int i)

{

for (int m = 0; m < recourse\_num; m++)

{

if (Request[i][m] > Need[i][m])

{

cout << "所需资源数超出其宣布的最大值!" << endl;

return 0;

}

else if (Request[i][m] > Available[m])

{

cout << "无足够资源，p[" << i << "]需等待!" << endl;

return 0;

}

}

// 尝试为进程分配资源

for (int j = 0; j < recourse\_num; j++)

{

Available[j] = Available[j] - Request[i][j];

Allocation[i][j] = Allocation[i][j] + Request[i][j];

Need[i][j] = Need[i][j] - Request[i][j];

}

cout << "尝试进行资源分配后的进程资源表: " << endl;

print\_information();

// 执行安全性算法

cout << "正在进行安全性检查" << endl;

if (security\_test())

{

cout << "经安全性检查，系统安全，本次分配成功" << endl;

cout << "安全进程序列为：";

for (int i = 0; i < process\_num - 1; i++)

{

cout << "P[" << sign[i] << "]--->"; // 输出安全的进程序列

}

cout << "P[" << sign[process\_num - 1] << "]";

}

else

{

cout << "安全性检测失败!系统不安全!!!正在恢复资源分配------" << endl;

for (int j = 0; j < recourse\_num; j++) // 恢复进程资源分配

{

Available[j] = Available[j] + Request[i][j];

Allocation[i][j] = Allocation[i][j] - Request[i][j];

Need[i][j] = Need[i][j] + Request[i][j];

}

}

return 0;

}

int main()

{

init();

cout << "初始化进程信息为：" << endl;

print\_information();

int i, tt = 0;

while (tt != 999)

{

cout << endl

<< "请输入请求资源Request[进程标号i][资源类型j]:" << endl;

cout << "进程i=：";

cin >> i;

cout << "各类资源数量(A B C)=: ";

for (int m = 0; m < recourse\_num; m++)

cin >> Request[i][m];

cout << endl;

// 执行操作系统第五次实验银行家算法

Banker\_Algorithm(i);

// 输出每次判断产生的执行序列

cout << endl

<< "当前资源分配表：" << endl;

print\_information();

cout << endl

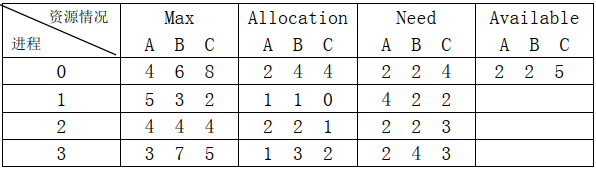
<< "请输入N(当N=999退出)：" << endl;

cin >> tt;

}

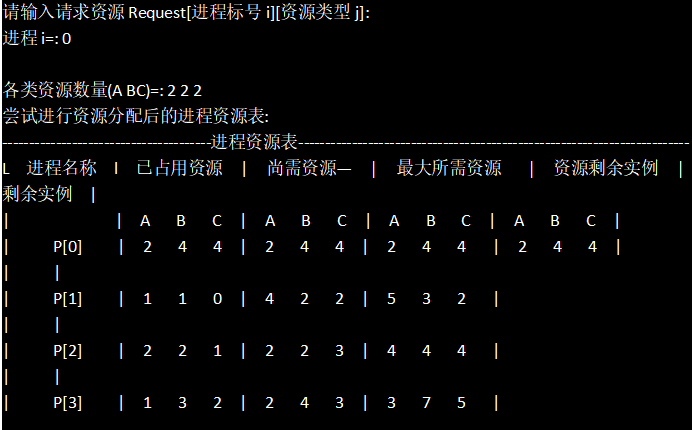
return 0;

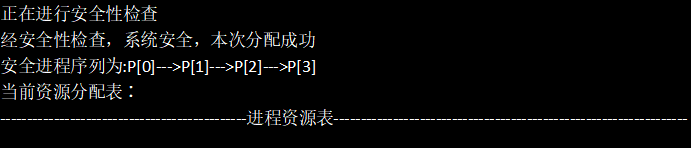
}

* T0 时刻的资源分配表(各种资源的数量分别为：10、5、7)。
* 请求进程：0，需要资源（2,2,2）

实验结果：

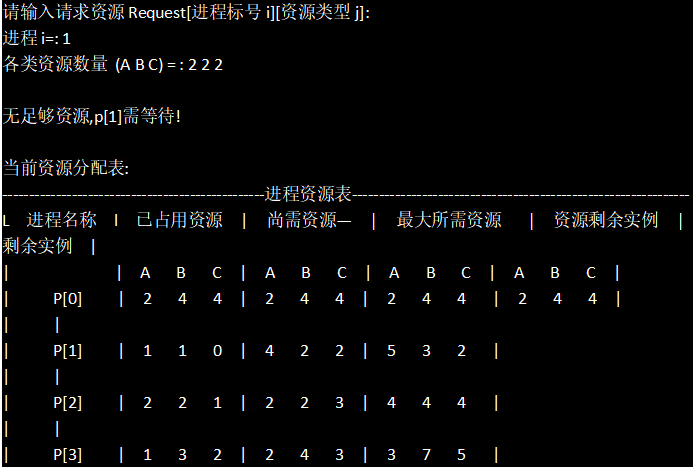






* 请求进程：1，需要资源（2,2,2）

实验结果：



* 请求进程：2，需要资源（2,2,7）

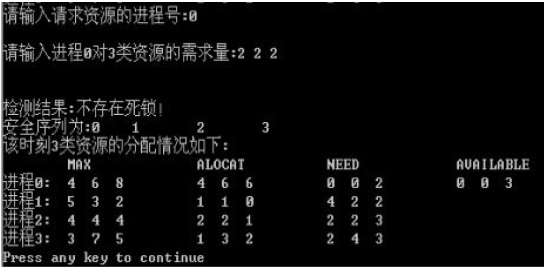
实验结果：



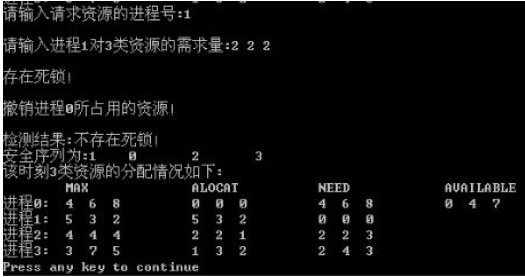
* 模拟一个银行家算法,初始化时让系统拥有一定的资源,用键盘输入的方式申请资源；



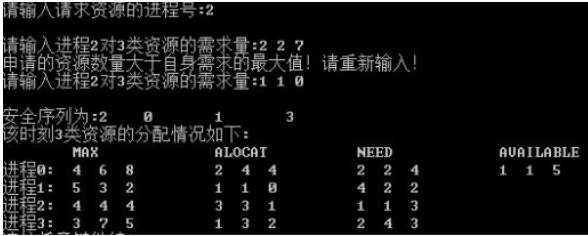
* 如果预分配后，系统处于安全状态，则修改系统的资源分配情况,如果预分配后，系统处于不安全状态，则提示不能满足请求；
* 无死锁：



* 有死锁：



* 请求量大：

六．实验结果分析（20分）

1. 实验思考（1）总结调用fork() 函数后的三种返回情况。

在调用fork(函数后，会出现以下三种返回情况:  
1.父进程中，fork()返回子进程的进程ID，子进程中返回值为0。这意味着在父进程中  
可以通过判断fork()的返回值是否等于0，来判断当前代码是否在子进程中执行。而在子进程中，可以通过判断fork()的返回值是否等于0，来确定自己是子进程。2.fork()返回-1，这意味着在创建子进程时出现错误。通常是因为当前进程已经创建了太多的子进程，或者系统资源不足等原因导致。3.在子进程中，如果调用了exec()函数族中的任何一个函数，那么该函数不会返回，而是用新的程序替换掉当前的进程映像。如果在父进程中调用了 wait()或waitpid()等函数，可以等待子进程的退出并获取子进程的退出状态。  
(2）总结fork()和wait()配合使用的情况，并尝试在父进程中取消wait()函数，观察进程的运行情况。  
在使用fork()函数创建子进程后，通常需要使用wait()或waitpid(函数等待子进程的退出。这是因为在子进程退出之前，父进程会一直处于阻塞状态，无法继续执行。而调用wait(或waitpid(函数可以防止这种情况发生，让父进程等待子进程的退出并获取其退出状态。当父进程中调用wait()或 waitpid()函数时，它会等待任何一个子进程退出，并获取其退  
出状态。如果有多个子进程同时退出，那么、 wait()或waitpid()函数将会以某种方式挑选一个子进程返回其退出状态。如果父进程中没有调用wait()或 waitpid()函数，则子进程退出后会变成“僵尸进程”，占用系统资源，可能会导致系统出现问题。如果在父进程中取消wait()函数的调用，那么子进程将会在退出后变成“僵尸进程”，占用系统资源。在父进程中可以通过调用signal()函数来注册SIGCHILD信号的处理函数，并在处理函数中调用 wait()或waitnid()函数来等待子进程的退出并获取其退出状态。  
(3）验证、总结 exec函数族的具体使用方法。exec函数族是Linux/Unix操作系统中用于执行其他程序的函数，其常用的几个函数包括execl、e.xe&x、exec.le、gxecxe.,等。这些函数的作用是将一个可执行程序加载到当前进程的地址空间中，并运行该程序。当调用exec函数族中的任意一个函数时，当前进程的内存映像被新的程序覆盖，因此原来进程的代码段、数据段等被新程序的代码和数据替代，进程的PID和进程上下文（如打开的文件描述符、信号处理器等）则保持不变。exec函数族的具体使用方法如下: execl.和execle,函数这两个函数可以用于执行一个程序，并传递给该程序一系列的参数，其函数原型为: int execl.(const. char \*path，c.oanst char\*aTg，.. .) ; int exec.le(const char \*path，const char \*ang，...，char \* const. envp[]);其中，path参数指定要执行的程序的路径，arg.,和后续的可变参数指定要传递给该程序的命令行参数。execle,函数在参数列表的末尾增加了一个指向环境变量数组的指针，该数组包含了要传递给新程序的环境变量。execy.和execve,函数这两个函数与exec.l.和execle.类似，也用于执行一个程序，不同的是，它们接收一个指向参数数组和环境变量数组的指针，其函数原型为: intexacy (const. char \*path，char \*const. a.gy[]); int execue (const. char \*path，char \*const.axgy,[l，char \*c.onst snwn[l);  
其中，path参数指定要执行的程序的路径，agy,参数是指向参数列表的指针，e.nMp,参数是指向环境变量数组的指针。需要注意的是，在使用exec函数族时，被调用程序的路径必须是一个可执行文件的绝对路径或者相对路径，并且该文件必须具有可执行权限。如果被调用程序的路径无效，exec函数族将返回-1，并且errno,变量被设置为ENOENT(文件不存在错误）。总之，exec函数族是一个强大的函数族，它可以用于执行其他程序，并将当前进程的内存映像替换为被调用程序的内存映像。在实际开发中，exec函数族常常被用来实现进程间的通信和任务切换，以及实现各种系统工具和应用程序。

2．分析、总结

就进程管理来说，操作系统通过进程控制块（PCB）来管理每个进程的信息和状态，通过各种调度算法来对进程进行调度和切换。fork和exec函数族则是操作系统中常用的系统调用，fork用于创建一个子进程，而exec用于加载并执行另一个程序。通过fork和exec的结合使用，我们可以实现创建新进程并在该进程中执行其他程序的功能。

具体而言，使用fork函数可以复制当前进程的副本，并在子进程中返回0，在父进程中返回子进程的pid。通过判断fork的返回值，我们可以确定当前进程是父进程还是子进程，从而分别执行不同的代码。而wait函数则是在父进程中等待子进程结束，并获取子进程的返回状态。通过这种方式，我们可以让父进程等待子进程执行完毕后再继续执行，从而实现进程间的同步和协作。

总之，进程管理和系统调用是操作系统中非常重要的内容，它们与操作系统的性能和稳定性密切相关。在编写操作系统相关的代码和程序时，我们需要对这些知识点有一定的掌握，并且灵活运用各种调度算法和系统调用来实现不同的功能。

3．体会

从以上问题中可以看出，操作系统是一个非常庞大且复杂的领域，需要我们花费大量的时间和精力去学习和掌握。在学习过程中，我们需要注重理论和实践的结合，通过编写代码和实验来加深对知识点的理解和掌握。此外，需要多关注操作系统的发展历程和最新的技术趋势，不断学习新的知识和技能，才能跟上行业的发展和变化。

另外，在学习操作系统的过程中，需要注重思考和总结，对于每一个知识点都要有自己的理解和归纳。只有通过不断的思考和总结，才能真正将知识点掌握和运用自如。