《操作系统原理》

实习报告

班级 220221

学号 20221000804

姓名 秦意诚

院系 未来技术学院

专业 电子信息类

指导老师 张求明

完成日期 2024年6月20日

[**实验一 作业调度 4**](#_Toc171097980)

[**1. 题目及要求 4**](#_Toc171097981)

[**1.1实验目的 4**](#_Toc171097982)

[**1.2实验内容 4**](#_Toc171097983)

[**2. 程序功能与设计思路 4**](#_Toc171097984)

[**2.1程序功能 4**](#_Toc171097985)

[**2.2设计思路 5**](#_Toc171097986)

[**3. 数据结构与算法设计 5**](#_Toc171097987)

[**3.1数据结构 5**](#_Toc171097988)

[**3.2算法设计 6**](#_Toc171097989)

[**4. 程序运行结果与分析 12**](#_Toc171097990)

[**4.1程序运行结果 12**](#_Toc171097991)

[**4.2结果分析 13**](#_Toc171097992)

[**5. 所遇问题与解决办法 14**](#_Toc171097993)

[**5.1所遇问题 14**](#_Toc171097994)

[**5.2解决办法 14**](#_Toc171097995)

[**实验二 磁盘调度 14**](#_Toc171097996)

[**1. 题目及要求 14**](#_Toc171097997)

[**1.1实验目的 14**](#_Toc171097998)

[**1.2实验内容 14**](#_Toc171097999)

[**2.程序功能与设计思路 15**](#_Toc171098000)

[**2.1程序功能 15**](#_Toc171098001)

[**2.2设计思路 15**](#_Toc171098002)

[**3.数据结构与算法设计 15**](#_Toc171098003)

[**3.1数据结构 15**](#_Toc171098004)

[**3.2算法设计 16**](#_Toc171098005)

[**4.程序运行结果与分析 19**](#_Toc171098006)

[**4.1程序运行结果 19**](#_Toc171098007)

[**4.2结果分析 20**](#_Toc171098008)

[**5.所遇问题与解决办法 20**](#_Toc171098009)

[**5.1所遇问题 20**](#_Toc171098010)

[**5.2解决办法 20**](#_Toc171098011)

[**实验三 熟悉Linux文件系统调用 21**](#_Toc171098012)

[**1.题目及要求 21**](#_Toc171098013)

[**1.1实验目的 21**](#_Toc171098014)

[**1.2实验内容 21**](#_Toc171098015)

[**2.程序功能与设计思路 21**](#_Toc171098016)

[**2.1程序功能 21**](#_Toc171098017)

[**2.2设计思路 21**](#_Toc171098018)

[**3.数据结构与算法设计 22**](#_Toc171098019)

[**3.1数据结构 22**](#_Toc171098020)

[**3.2算法设计 22**](#_Toc171098021)

[**4.程序运行结果与分析 24**](#_Toc171098022)

[**4.1程序运行结果 24**](#_Toc171098023)

[**4.2结果分析 27**](#_Toc171098024)

[**5.所遇问题与解决办法 27**](#_Toc171098025)

[**5.1所遇问题 27**](#_Toc171098026)

[**5.2解决办法 27**](#_Toc171098027)

[**实验四 进程管理 28**](#_Toc171098028)

[**1.题目及要求 28**](#_Toc171098029)

[**1.1实验目的 28**](#_Toc171098030)

[**1.2实验内容 28**](#_Toc171098031)

[**2.程序功能与设计思路 28**](#_Toc171098032)

[**2.1程序功能 28**](#_Toc171098033)

[**2.2设计思路 29**](#_Toc171098034)

[**3.数据结构与算法设计 29**](#_Toc171098035)

[**3.1数据结构 29**](#_Toc171098036)

[**3.2算法设计 29**](#_Toc171098037)

[**4.程序运行结果与分析 31**](#_Toc171098038)

[**4.1程序运行结果 31**](#_Toc171098039)

[**4.2结果分析 32**](#_Toc171098040)

[**5.所遇问题与解决办法 32**](#_Toc171098041)

[**5.1所遇问题 32**](#_Toc171098042)

[**5.2解决办法 32**](#_Toc171098043)

[**实验五 进程通信 33**](#_Toc171098044)

[**1.题目及要求 33**](#_Toc171098045)

[**1.1实验目的 33**](#_Toc171098046)

[**1.2实验内容 33**](#_Toc171098047)

[**2.程序功能与设计思路 34**](#_Toc171098048)

[**2.1程序功能 34**](#_Toc171098049)

[**2.2设计思路 34**](#_Toc171098050)

[**3.数据结构与算法设计 34**](#_Toc171098051)

[**3.1数据结构 34**](#_Toc171098052)

[**3.2算法设计 35**](#_Toc171098053)

[**4.程序运行结果与分析 39**](#_Toc171098054)

[**4.1程序运行结果 39**](#_Toc171098055)

[**4.2结果分析 40**](#_Toc171098056)

[**5.所遇问题与解决办法 40**](#_Toc171098057)

[**5.1所遇问题 40**](#_Toc171098058)

[**5.2解决办法 40**](#_Toc171098059)

[**心得体会 40**](#_Toc171098060)

# 实验一 作业调度

## 题目及要求

### **1.1实验目的**

1．对作业调度的相关内容作进一步的理解。

2. 明白作业调度的主要任务。

3. 通过编程掌握作业调度的主要算法。

### **1.2实验内容**

1.假设系统中可同时运行两道作业，给出每道作业的到达时间和运行时间，如下表所示：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 作业名 | A | B | C | D | E | F | G | H | I | J |
| 到达时间 | 0 | 2 | 5 | 7 | 12 | 15 | 4 | 6 | 8 | 10 |
| 运行时间 | 7 | 10 | 20 | 30 | 40 | 8 | 8 | 20 | 10 | 12 |

2.分别用先来先服务算法、短作业优先和响应比高者优先三种算法给出作业的调度顺序。

3. 计算每一种算法的平均周转时间及平均带权周转时间并比较不同算法的优劣。

## 程序功能与设计思路

### 2.1程序功能

1.首先，该程序要通过菜单对功能进行选择。需要包括以下几部分的内容：先来先服务算法处理模块，短作业优先算法处理模块，高响应比优先算法模块，三种算法性能对比以及程序退出。

2.在选择每个算法后，应当输出作业的完成序列。同时，为了让输出的信息更加全面，可以输出到达时间，结束时间，运行时间，周转时间和带权周转时间。响应比会随着时间改变，没有输出意义，且响应比也仅仅用于高响应比优先的方式。

3.在对比三种算法时，主要是输出三种方式的平均周转时间，平均带权周转时间。通过这两个参数比较，可以在一定程度上反应算法优劣。

### 2.2设计思路

1.定义关于作业的结构体，存储相关信息。作业信息从文件读入，但读入信息有限，只有到达时间和运行时间，其他的则先可以先置为一定值，之后再更新。

2.为了模拟同时有两道作业可以在系统中运行，定义两个CPU时间，通过作业的到达时间和CPU时间去判断是否可以被选择运行。

3.在FCFS（先来先服务）中，分为CPU可提供使用和不可使用，对于前者满足可以运行的所有作业，寻找先到的，而对于CPU暂不可使用的情况，直接选择之后最先达到的即可。

4.在SJF（短作业优先中）中，和3中一样，在CPU可用作业中选择短作业，反之在CPU忙时直接让下一个到来的上机即可.

5.在HRNN（高响应比优先）中，和之前的思路一样，但是每选择一次作业之后要重新计算一次相应比。同样，在CPU都忙的时候，直接让最先来的运行即可，此时该作业其实就是响应比最大作业。

6.通过循环实现菜单的反复打印，直至退出。

## 数据结构与算法设计

### 3.1数据结构

1. **struct** Job {
2. string name;                //作业名，到达，开始，结束，运行，周转，带权周转时间
3. **double** arrive\_time;         //还有响应比，但是只在第三个算法中会使用到
4. **double** run\_time;
5. **double** start\_time;
6. **double** end\_time;
7. **double** turn\_time;
8. **double** weight\_time;
9. **double** rate;
10. //重载=,方便在之后操作中直接用=进行赋值
11. Job& operator=(**const** Job& job) {
12. **this**->name = job.name;
13. **this**->arrive\_time = job.arrive\_time;
14. **this**->end\_time = job.end\_time;
15. **this**->run\_time = job.run\_time;
16. **this**->start\_time = job.start\_time;
17. **this**->turn\_time = job.turn\_time;
18. **this**->weight\_time = job.weight\_time;
19. **this**->rate = job.rate;
20. **return** \***this**;
21. }
22. };

### 3.2算法设计

1.FCFS

1. //先来先服务
2. vector<Job>FCFS(vector<Job>Jobs) {
3. vector<Job>Finish;       //存放已完成作业
4. vector<Job>::iterator p;
5. CPU1\_time = 0, CPU2\_time = 0;//开始前将两个CPU时间置0
6. **bool** flag;
7. **int** n = Jobs.size();
8. **for** ( **int** i = 0; i < n; i++) {
9. vector<Job>::iterator it = Jobs.begin();
10. Job temp;
11. temp.arrive\_time = INT\_MAX;
12. flag = **false**;
13. **while** (it != Jobs.end()) {                 //寻找当前已到达可以在CPU上运行，而且到达时间最早的作业
14. **if** (it->run\_time != 0 && it->arrive\_time <= temp.arrive\_time && (it->arrive\_time <=CPU1\_time || it->arrive\_time <= CPU2\_time)) {
15. temp = \*it;
16. p = it;
17. flag = **true**;
18. }
19. it++;
20. }
21. **if** (flag == **false**) {
22. it = Jobs.begin();
23. **while** (it != Jobs.end()) {           //找不到就是现在CPU暂时不能用，那只需要找运行到达时间最早的作业即可
24. **if** (it->run\_time != 0 && it->arrive\_time < temp.arrive\_time) {
25. p = it;
26. temp = \*it;
27. }
28. it++;
29. }
30. }
32. Jobs.erase(p);           //接下来模拟上机运行，修改对应参数
33. **if** (CPU1\_time <= CPU2\_time) {               //这是CPU1空闲的情况
34. **if** (i == 0) {
35. CPU1\_time = temp.arrive\_time;   //0表示第一个作业直接上
36. }
37. **if** (flag == **false**) {
38. CPU1\_time = temp.arrive\_time;    //全忙但是CPU1先空出来
39. }
41. temp.start\_time = CPU1\_time;
42. CPU1\_time += temp.run\_time;
43. temp.end\_time = CPU1\_time;            //修改
44. temp.turn\_time = temp.end\_time - temp.arrive\_time;
45. temp.weight\_time = temp.turn\_time / temp.run\_time;
46. Finish.push\_back(temp);
47. }
48. **else** {
49. **if** (i == 1 || flag == **false**) {
50. CPU2\_time = temp.arrive\_time;
51. }
52. temp.start\_time = CPU2\_time;
53. CPU2\_time += temp.run\_time;
54. temp.end\_time = CPU2\_time;
55. temp.turn\_time = temp.end\_time - temp.arrive\_time;
56. temp.weight\_time = temp.turn\_time / temp.run\_time;
57. Finish.push\_back(temp);
58. }
59. }
60. **return** Finish;   //返回完成队列
61. }

2.SJF

1. vector<Job>SJF(vector<Job>Jobs) {
2. vector<Job>Finish;
3. vector<Job>::iterator p;
4. CPU1\_time = 0, CPU2\_time = 0;
5. **bool** flag;
6. **int** n = Jobs.size();
7. **for** (**int** i = 0; i < n; i++) {
8. vector<Job>::iterator it = Jobs.begin();
9. Job temp;
10. temp.arrive\_time = INT\_MAX;
11. temp.run\_time = INT\_MAX;   //这里将到达时间和运行时间设为最大
12. flag = **false**;
13. **while** (it != Jobs.end()) {
14. **if** (CPU1\_time <= CPU2\_time && it->arrive\_time <= CPU1\_time) {  //满足可上CPU1且已经到达
15. **if** (it->run\_time != 0 && it->run\_time <temp.run\_time) {    //在满足的作业中选择运行时间最短的
16. temp = \*it;
17. p = it;
18. flag = **true**;
19. }
20. }
21. **if** (CPU1\_time > CPU2\_time && it->arrive\_time <= CPU2\_time) {    //满足可上CPU2且已经到达
22. **if** (it->run\_time != 0 && it->run\_time < temp.run\_time) {    //在满足的作业中选择运行时间最短的
23. temp = \*it;
24. p = it;
25. flag = **true**;
26. }
27. }
28. it++;
29. }
30. **if** (flag == **false**) {
31. it = Jobs.begin();
32. **while** (it != Jobs.end()) {   //如果暂时没有空闲CPU，则直接选一个先到达的即可，这里不用考虑短作业优先，因为只会有一个先来，而先来必定先上
33. **if** (it->run\_time != 0 && it->arrive\_time < temp.arrive\_time) {
34. temp = \*it;
35. p = it;
36. }
37. it++;
38. }
39. }
40. Jobs.erase(p);
41. **if** (CPU1\_time <= CPU2\_time) {          //满足可上CPU1
42. **if** (i == 0 || flag == **false**) {           //第0道作业以及CPU全忙时的情况
43. CPU1\_time = temp.arrive\_time;
44. }
45. temp.start\_time = CPU1\_time;
46. CPU1\_time += temp.run\_time;
47. temp.end\_time = CPU1\_time;
48. temp.turn\_time = temp.end\_time - temp.arrive\_time;
49. temp.weight\_time = temp.turn\_time / temp.run\_time;
50. Finish.push\_back(temp);
51. }
52. **else** {
53. **if** (i == 1 || flag == **false**) {           //满足可以上CPU2的情况
54. CPU2\_time = temp.arrive\_time;
55. }
56. temp.start\_time = CPU2\_time;
57. CPU2\_time += temp.run\_time;
58. temp.end\_time = CPU2\_time;
59. temp.turn\_time = temp.end\_time - temp.arrive\_time;
60. temp.weight\_time = temp.turn\_time / temp.run\_time;
61. Finish.push\_back(temp);
62. }
63. }
64. **return** Finish;
65. }

3.HRRN

1. //高响应比优先
2. vector<Job>HRRN(vector<Job>Jobs) {
3. vector<Job>Finish;
4. vector<Job>::iterator p;
5. CPU1\_time = 0, CPU2\_time = 0;
6. **bool** flag;
7. **int** n = Jobs.size();
8. **for** (**int** i = 0; i < n; i++) {
9. vector<Job>::iterator it = Jobs.begin();
10. Job temp;
11. temp.rate = INT\_MIN;
12. temp.arrive\_time = INT\_MAX;    //响应比设为最小，到达时间设为最大
13. flag = **false**;
14. **while** (it != Jobs.end()) {
15. **if** (CPU1\_time <= CPU2\_time && it->arrive\_time <= CPU1\_time) {     //满足可以上CPU1的作业，
16. it->rate = (CPU1\_time-it->arrive\_time + it->run\_time) / it->run\_time;    //计算响应比
17. **if** (it->rate != 0 && it->rate > temp.rate) {
18. temp = \*it;
19. p = it;          //不断寻找，找到满足条件的最高响应比
20. flag = **true**;
21. }
22. }
23. **if** (CPU1\_time > CPU2\_time && it->arrive\_time <= CPU2\_time) {          //同理，换一个CPU
24. it->rate = (CPU2\_time - it->arrive\_time + it->run\_time) / it->run\_time;
25. **if** (it->rate != 0 && it->rate > temp.rate) {
26. temp = \*it;
27. p = it;
28. flag = **true**;
29. }
30. }
31. it++;
32. }
33. **if** (flag == **false**) {
34. it = Jobs.begin();
35. **while** (it != Jobs.end()) {              //均忙则找先来的即可，这时不用考虑响应比
36. **if** (it->run\_time != 0 && it->arrive\_time < temp.arrive\_time) {
37. temp = \*it;
38. p = it;
39. }
40. it++;
41. }
42. }
43. Jobs.erase(p);
44. **if** (CPU1\_time <= CPU2\_time) {              //均同理
45. **if** (i == 0 || flag == **false**) {
46. CPU1\_time = temp.arrive\_time;
47. }
48. temp.start\_time = CPU1\_time;
49. CPU1\_time += temp.run\_time;
50. temp.end\_time = CPU1\_time;
51. temp.turn\_time = temp.end\_time - temp.arrive\_time;
52. temp.weight\_time = temp.turn\_time / temp.run\_time;
53. Finish.push\_back(temp);
54. }
55. **else** {
56. **if** (i == 1 || flag == **false**) {
57. CPU2\_time = temp.arrive\_time;
58. }
59. temp.start\_time = CPU2\_time;
60. CPU2\_time += temp.run\_time;
61. temp.end\_time = CPU2\_time;
62. temp.turn\_time = temp.end\_time - temp.arrive\_time;
63. temp.weight\_time = temp.turn\_time / temp.run\_time;
64. Finish.push\_back(temp);
65. }
66. }
67. **return** Finish;
68. }

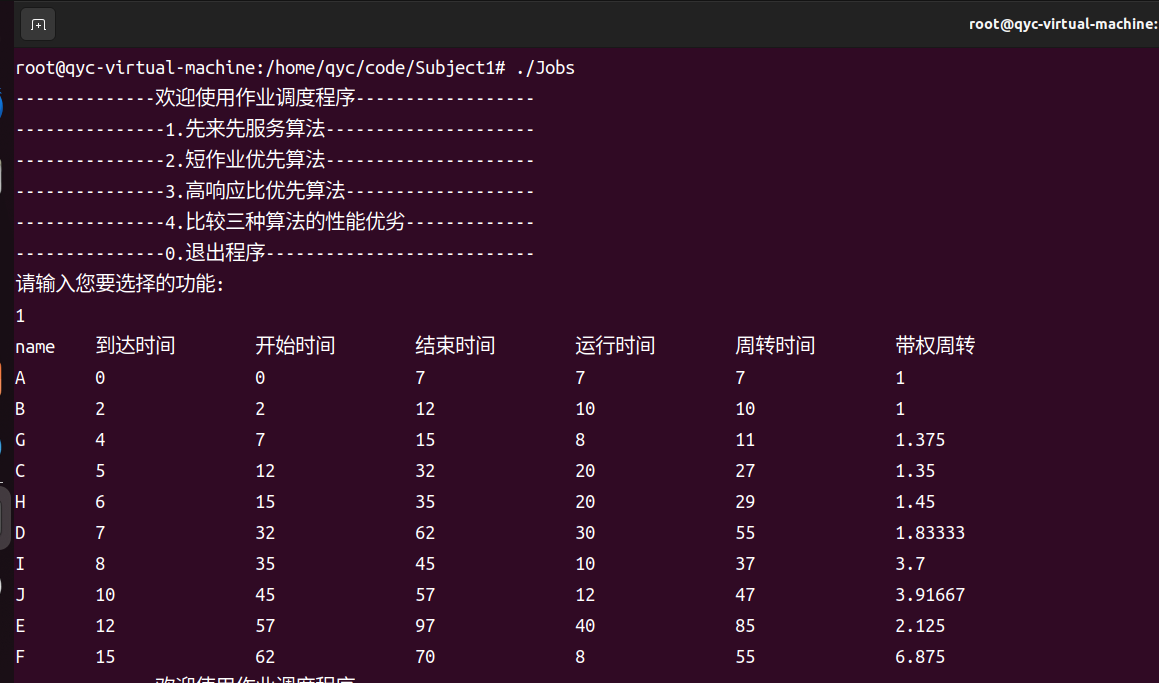
4．输出完成队列

1. **void** PrintInforamtion(vector<Job>v) {
2. cout << "name" << "\t"
3. << "到达时间" << "\t"
4. << "开始时间" << "\t"
5. << "结束时间" << "\t"
6. << "运行时间" << "\t"
7. << "周转时间" << "\t"
8. << "带权周转" << endl;
9. **for** (vector<Job>::iterator it = v.begin(); it != v.end(); it++) {
10. cout << it->name << "\t"
11. << it->arrive\_time << "\t\t"
12. << it->start\_time << "\t\t"
13. << it->end\_time << "\t\t"
14. << it->run\_time << "\t\t"
15. << it->turn\_time << "\t\t"
16. << it->weight\_time << endl;
17. }
18. }

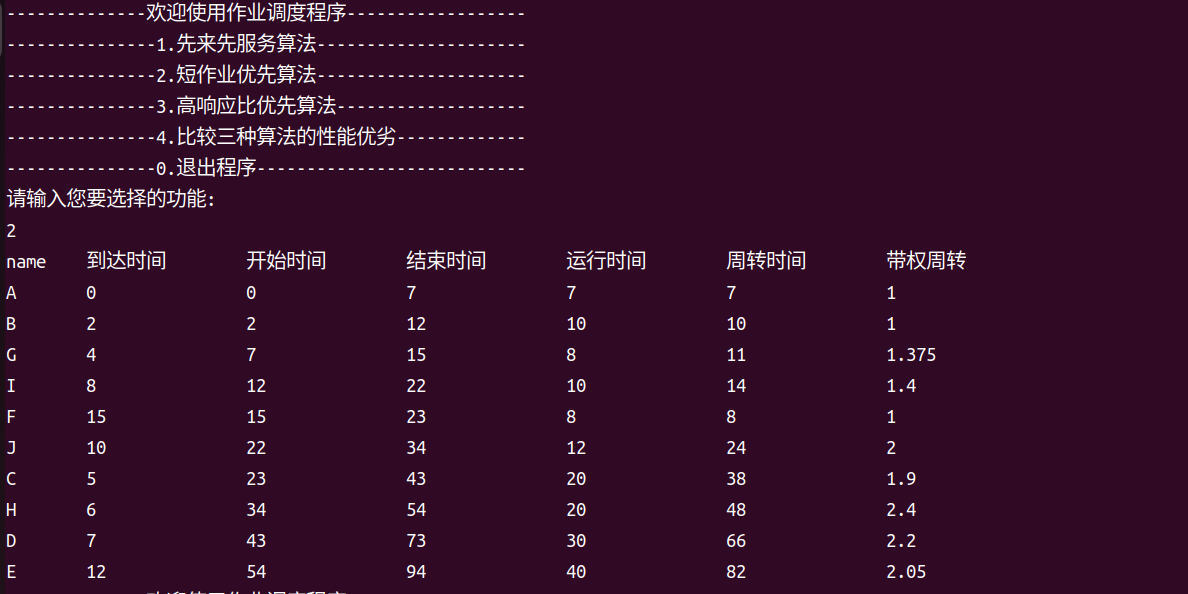
## 程序运行结果与分析

### 4.1程序运行结果

（1）选择1，先来先服务算法



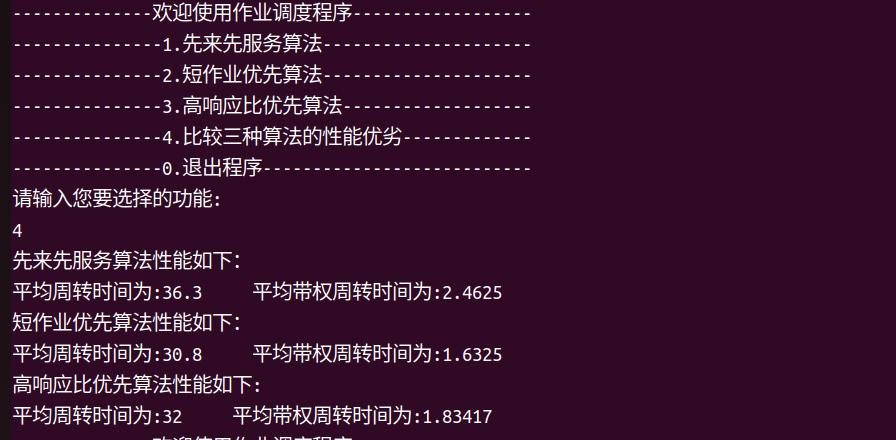
（2）选择2，短作业优先算法



（3）选择3，高响应比优先算法



（4）选择4，对比三种算法



### 4.2结果分析

1.通过手写推导模拟系统中同时有两道作业可以运行，与三种算法对比，发现无误，表示程序的运行结果是正确的。

2.输出格式内容与格式符合预期，不仅输出了作业的完成序列，还输出了对应的到达时间，结束时间，运行时间，周转时间，带权周转时间。在格式上输出内容相互对齐，符合要求。

3.成功输出了三种算法的对比。同时，通过对比我们可以得出在这个作业到达顺序中，短作业优先最好，高相应比优先次之，最差的是先来先服务算法。

## 所遇问题与解决办法

### 5.1所遇问题

一开始定义作业结构体，结构体中定义较少，导致在函数过程中会额外定义很多东西去计算并存储，最后在输出完成序列的时候，算法函数栈区被清空，无法完整输出所有数据，造成了困扰。

在初始编写中用结构体去定义了CPU，导致操作复杂，难以进行。

### 5.2解决办法

将作业的几乎所有相关参数都定义在了结构体中，这样操作会更加方便，也能轻易做联动的计算，会大大简化三个算法的函数内部。

同时，去繁为简，直接用两个int变量记录CPU时间，在反复迭代中更新，避免使用结构体进行复杂操作。

# 实验二 磁盘调度

## 题目及要求

### 1.1实验目的

1.对磁盘调度的相关知识作进一步的了解，明确磁盘调度的原理。

2.加深理解磁盘调度的主要任务。

3.通过编程，掌握磁盘调度的主要算法。

### 1.2实验内容

1. 对于如下给定的一组磁盘访问进行调度：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 请求服务到达 | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| 访问的磁道号 | 30 | 50 | 100 | 180 | 20 | 90 | 150 | 70 | 80 | 10 | 160 | 120 | 40 | 110 |

2. 要求分别采用先来先服务、最短寻道优先(距离相等时按FCFS排序)以及电梯调度算法进行调度。

3. 要求给出每种算法中磁盘访问的顺序，计算出平均移动道数。

4. 假定当前读写头在90号，电梯调度算法向磁道号增加的方向移动。

## 2.程序功能与设计思路

### 2.1程序功能

1.用户通过菜单实现功能选择，包括选择不同的算法和退出

2.选择对应算法后，输出磁盘请求满足顺序，并输出平均移动磁道数。

### 2.2设计思路

1.定义关于请求的结构体存放请求名和磁道数方便后续操作，并从文件读入所有信息。

2.对于先来先服务算法，直接对请求序列顺序执行即可。

3.对于最短寻道算法，要引入一个寻找最短作业的函数，每次在选择时调用该函数辅助选择相对于当前位置最短的请求。

4.对于电梯算法，已指定方向。将大于和小于当前磁道数的请求划分为两部分，并各自排序，对于排序好的两个容器，先执行大于90的部分，再返回执行小于90的部分即可。

5．通过循环实现对功能的选择的重复，直至退出。

## 3.数据结构与算法设计

### 3.1数据结构

1. **struct** Request {    //定义磁盘调度请求的结构体
2. string name;
3. **int** number;
4. //重载=         方便后续操作直接通过=进行赋值
5. Request& operator=(**const** Request& r) {
6. **this**->name = r.name;
7. **this**->number = r.number;
8. **return** \***this**;
9. }
10. };

### 3.2算法设计

（1）先来先服务算法

1. vector<Request> FCFS(vector<Request>Requests,**double**&totalmove) {
2. vector<Request>finish;     //存放完成请求
3. **int** current\_position = 90;   //初始磁道位置
4. **while** (!Requests.empty()) {              //实现这个十分简单，即按顺序扫描序列即可
5. **if** (current\_position >= Requests[0].number) {
6. totalmove += current\_position - Requests[0].number;
7. current\_position = Requests[0].number;
8. }
9. **else** {
10. totalmove+= Requests[0].number - current\_position;
11. current\_position = Requests[0].number;
12. }
13. Request temp = Requests[0];
14. Requests.erase(Requests.begin());
15. finish.push\_back(temp);
16. }
17. **return** finish;
18. }

（2）最短寻道部分，包括主体与辅助函数

1. //辅助实现最短寻道的函数，参数是当前剩余请求和当前位置，返回当前容器下最短寻道的索引
2. **int** FindBest(vector<Request>v, **int** position) {
3. **int** index,i=0;
4. **int** temp = 999;
5. **int** n = v.size();
6. **for** ( ; i < n; i++) {
7. **int** delta = abs(v[i].number - position);
8. **if** (delta < temp) {   //这个<就可以做到即使在相等情况下的FCFS,即使相等也被直接掠过
9. temp = delta;
10. index = i;
11. }
12. }
13. **return** index;
14. }
15. //最短寻道优先
16. vector<Request>Shortest(vector<Request>Requests, **double**& totalmove) {
17. vector<Request>finish;
18. **int** index = 0;
19. **int** current\_position = 90;
20. **while** (!Requests.empty()) {
21. index = FindBest(Requests, current\_position);
22. totalmove += abs(Requests[index].number - current\_position);
23. current\_position = Requests[index].number;
24. Request r = Requests[index];
25. Requests.erase(Requests.begin() + index);//通过迭代器+索引的方式将已处理项目删掉
26. finish.push\_back(r);
27. };
28. **return** finish;
29. }

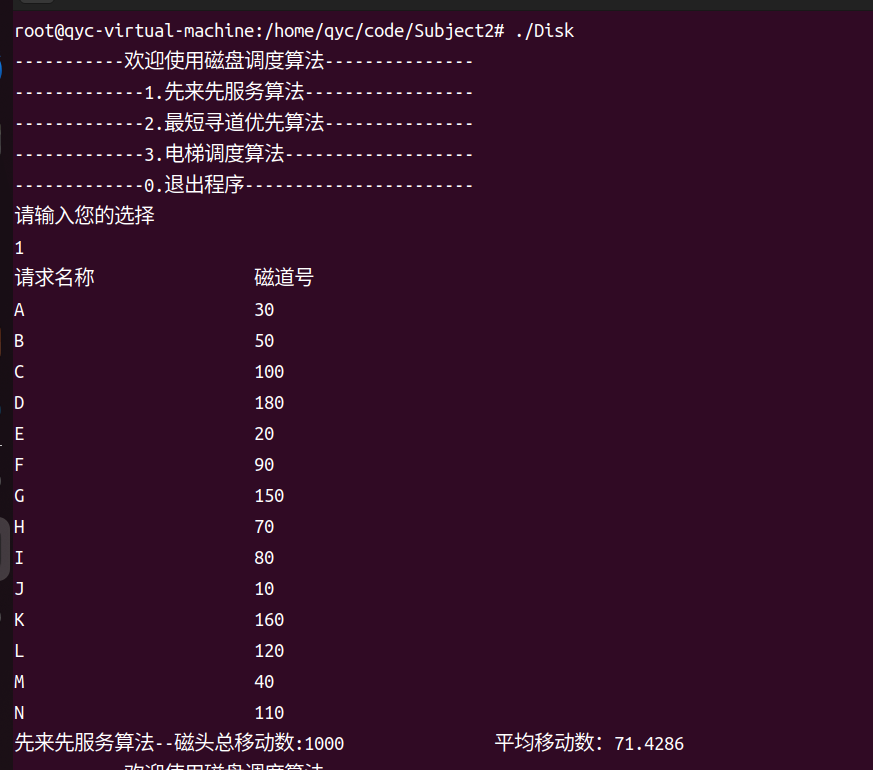
（3）电梯算法部分，包括辅助函数和主体部分

1. //辅助电梯算法实现的函数，将请求根据小于当前磁道号和大于当前磁道号分为两个请求队列
2. pair<vector<Request>,vector<Request>>SortforSCAN(vector<Request>Requests, **int** position) {
3. vector<Request>R1;   //R1存放位置小于当前位置的
4. vector<Request>R2;    //R2存放位置大于当前位置的
5. **for** (vector<Request>::iterator it = Requests.begin(); it != Requests.end(); it++) {
6. **if** (it->number < position) {
7. Request temp = \*it;
8. R1.push\_back(temp);
9. }
10. **else** {
11. Request temp = \*it;
12. R2.push\_back(temp);
13. }
14. }
15. **return** make\_pair(R1, R2);
16. }
17. //辅助电梯算法实现的函数，将传入的vector从小到大排列
18. vector<Request> Sortnumber(vector<Request> v) {
19. vector<Request> Sorted;
20. **int** n = v.size();
21. **while** (n > 0) {
22. **int** min\_index = 0;
23. **int** min\_value = INT\_MAX;
24. **for** (**int** i = 0; i < n; ++i) {
25. **if** (v[i].number < min\_value) {
26. min\_value = v[i].number;
27. min\_index = i;
28. }
29. }
30. Sorted.push\_back(v[min\_index]);
31. v.erase(v.begin() + min\_index);
32. n--;
33. }
34. **return** Sorted;
35. }
36. vector<Request>SCAN(vector<Request>Requests, **double**& totalmove) {
37. vector<Request>finish;
38. **int** current\_position = 90;
39. pair <vector<Request>, vector<Request>>R1\_R2 = SortforSCAN(Requests, current\_position);  //R1是小于的   R2是大于的
40. vector<Request>R1 = R1\_R2.first;
41. vector<Request>R2 = R1\_R2.second;
42. vector<Request>R1\_sort = Sortnumber(R1);
43. vector<Request>R2\_sort = Sortnumber(R2);
45. **for** (vector<Request>::iterator it = R2\_sort.begin(); it != R2\_sort.end(); it++) {     //先处理右侧队列，队列大小已从小到达排列
46. totalmove += it->number - current\_position;
47. current\_position = it->number;
48. Request temp = \*it;
49. finish.push\_back(temp);
50. }
51. reverse(R1\_sort.begin(), R1\_sort.end());     //先将左侧队列反转，改为从大到小排列
52. **for** (vector<Request>::iterator it = R1\_sort.begin(); it != R1\_sort.end(); it++) {
53. totalmove += current\_position - it->number;
54. current\_position = it->number;
55. Request temp = \*it;
56. finish.push\_back(temp);
57. }
58. **return** finish;
59. }

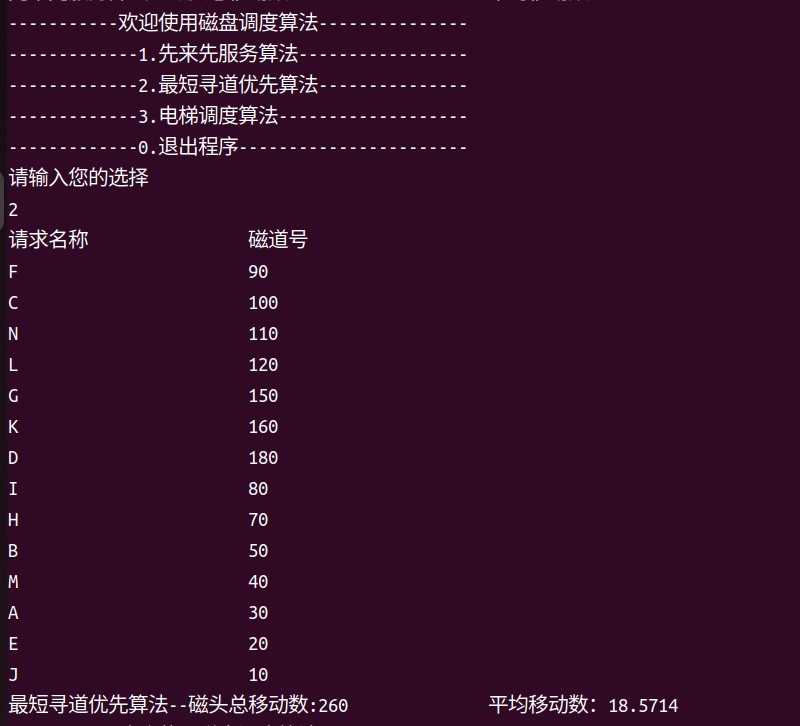
## 4.程序运行结果与分析

### 4.1程序运行结果

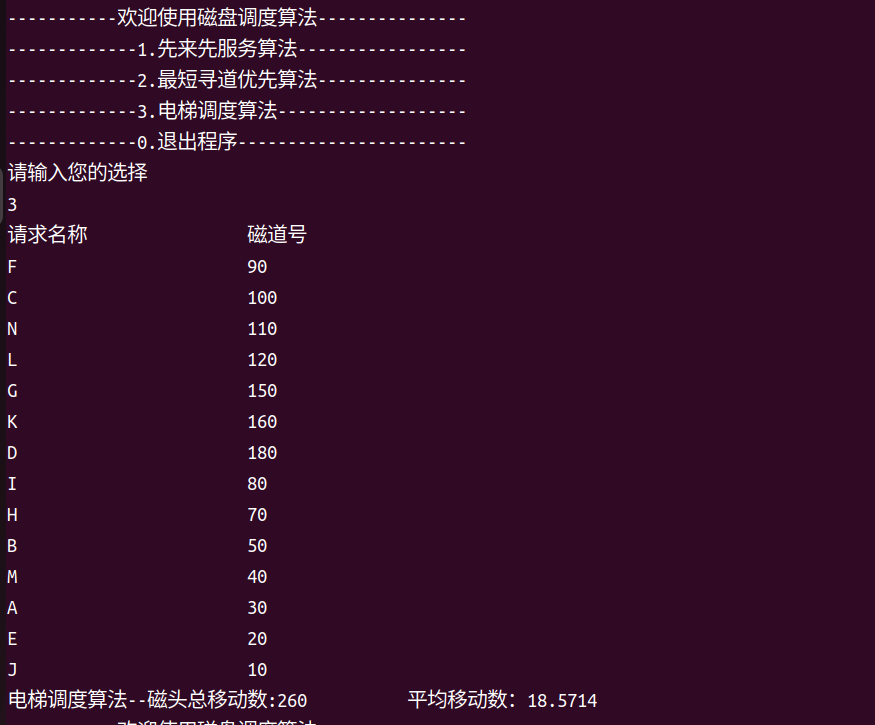
1.选择1，先来先服务算法



2.选择2，短寻道优先算法



3.选择3，电梯算法



### 4.2结果分析

通过手动推理与程序运行结果对比发现一致，证明程序运行结果正确。且正确输出移动序列和各自的位置，计算出了磁头移动总数和平均移动数。短寻道优先和电梯算法运行结果一致，这仅仅是巧合，两个函数的逻辑存在本质差别，且各自没有错误。

## 5.所遇问题与解决办法

### 5.1所遇问题

本题无疑问。

### 5.2解决办法

有体会如下：在定义结构体时，定义一些方法比如赋值的重载，可以方便后续的操作，这是从这一题和之前的第一题得出的体会。

# 实验三 熟悉Linux文件系统调用

## 1.题目及要求

### 1.1实验目的

1. 掌握 linux 提供的文件系统调用的使用方法。

2. 熟悉文件系统的系统调用用户接口。

3. 了解操作系统文件系统的工作原理和工作方式。

### 1.2实验内容

1.使用文件系统调用编写文件工具，包括以下功能：1.创建新文件。2.写文件。3.读文件。4.修改文件的权限。5.查看文件的当前权限。0.退出。

2.提示用户输入功能号，并根据用户输入的功能选择相应的功能。

3.文件按可变记录文件组织，具体记录内容自行设计。

## 2.程序功能与设计思路

### 2.1程序功能

1.通过菜单实现对功能的选择。

2.具有对文件创建，读，写，查看并修改权限的工作。

3.修改权限考虑让用户输入三位八进制数实现修改。

### 2.2设计思路

1.创建文件操作时，创建成功返回成功提示及文件标识符，若标识符Fd小于0，创建失败，进行报错提醒。

2.在对文件执行读和写操作时，均需先打开文件。在写操作时，需要将内容先写在缓冲区中，然后将缓冲区中内容写入文件。同理，在读操作执行过程中，要将文件内容先读入缓冲区，再将其输出。当然，这里所提出的“缓冲区”是自己所定义的。

3.修改文件权限时，通过8进制的三位数进行修改，4-读，2-写，1-可执行，同时还分为用户，组，其他三部分，每一部分对应0-7的数表示对权限的任意组合。

4.在查看文件权限时，利用所提供的资料中的语句实现。这句代码的本质是将命令放在一个字符串数组中，让后通过执行系统调用执行一个外部程序来查看权限。但是要注意的是，在查看完权限后，会将当前程序终止。

## 3.数据结构与算法设计

### 3.1数据结构

定义文件名的字符数组：

1. **char** filename[100];

存放查看权限函数的命令的参数的数组：

1. **char**\* pargv[4] = { "ls","-l",Path,NULL };   //先定义一大小为4的字符数组，作为下函数的参数

读写的存放缓冲区，同名，但在每个函数中均定义了一次“

1. **char** Content[100000];

### 3.2算法设计

1.创建文件

1. **void** CreateNewFile() {
2. **char** filename[100];
3. printf("请输入要创建的文件名:");
4. scanf("%s", filename);
5. Path = filename;
6. Fd = open(Path, O\_RDWR | O\_TRUNC | O\_CREAT, 0755);//第一个表示位置，第二个表示打开方式，第三个是权限的掩码
7. /\*
8. O\_RDONLY 只读方式  O\_WRONLY 只写方式  O\_RDWR读写方式
9. O\_CREAT 如果文件不存在则创建该文件，若存在则忽略。
10. O\_TRUNC 如果文件存在则将文件长度截为 0，属性和所有者不变。
11. C\_EXECL 如果文件存在且 O\_CREAT 被设置则强制 open 调用失败。
12. O\_APPEND 每次写入时都从文件尾部开始。
14. \*/
15. **if** (Fd < 0) {      //Fd小于0是创建失败的
16. perror("创建新文件失败");
17. }
18. **else** {
19. printf("文件%s创建成功", filename);
20. printf("文件标识符为：%d\n", Fd);
21. close(Fd);
22. }
23. }

2.读文件和写文件

1. **void** WriteFile() {
2. **char** filename[100];
3. printf("请输入要写的文件名:");
4. scanf("%s", filename);
5. Path = filename;
6. **char** Content[100000];
7. Fd = open(Path, O\_WRONLY | O\_APPEND);
8. **if** (Fd < 0) {
9. perror("写文件失败：");
10. **return**;
11. }
12. printf("请写入信息:\n");
13. **int** size = read(0, Content, 100000);  //第一个参数0表示从键盘输入，第二个为一个字符数组，第三个指定尺寸
14. write(Fd, Content, size);   //写入，文件名，内容，大小
15. printf("内容已成功写入文件\n");
16. close(Fd);
17. }
19. **void** ReadFIle() {
20. **char** filename[100];
21. printf("请输入要读的文件名:");
22. scanf("%s", filename);
23. Path = filename;
24. Fd = open(Path, O\_RDONLY);
25. **if** (Fd < 0) {
26. perror("读文件失败:");
27. }
28. **char** Content[100000];
29. **int** size = read(Fd, Content, 100000);   //改为从文件读入，其余和上个函数中该语句的参数一致
30. write(1, Content, size);     //表示从终端输出
31. printf("\n读文件执行完毕\n");
32. close(Fd);
33. }

3.修改文件的权限

1. **void** ChangePermission() {
2. **char** filename[100];
3. printf("请输入要修改权限的文件名:");
4. scanf("%s", filename);
5. Path = filename;
7. Fd = open(Path, O\_RDONLY);
8. **if** (Fd < 0) {
9. perror("修改权限失败:");
10. **return**;
11. }
12. PermissionMenu();
13. **int** number;
14. scanf("%d", &number);
15. **int** user = number / 100;
16. **int** group = (number / 10) % 10;
17. **int** other = number % 10;
18. **int** mode = user \* 8 \* 8 + group \* 8 + other;       //修改权限依照8进制实现，依照前面的0777
19. **int** IfSuccess = chmod(Path, mode);
20. **if** (IfSuccess == -1) {
21. printf("权限修改失败\n");
22. }
23. **else** {
24. printf("文件%s权限修改成功\n", Path);
25. }
26. close(Fd);
27. }

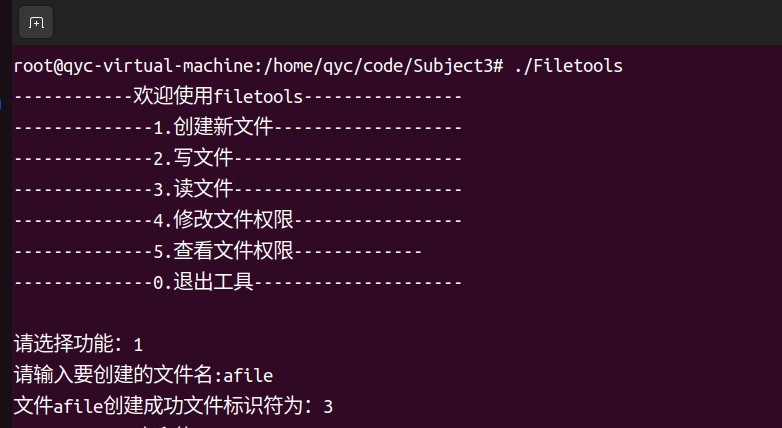
4.查看文件权限

1. **void** CheckPermission() {
2. **char** filename[100];
3. printf("请输入要查看权限的文件名:");
4. scanf("%s", filename);
5. Path = filename;
6. **char**\* pargv[4] = { "ls","-l",Path,NULL };   //先定义一大小为4的字符数组，作为下函数的参数
7. execv("/bin/ls", pargv);   //系统调用，用于执行一个外部程序
8. }

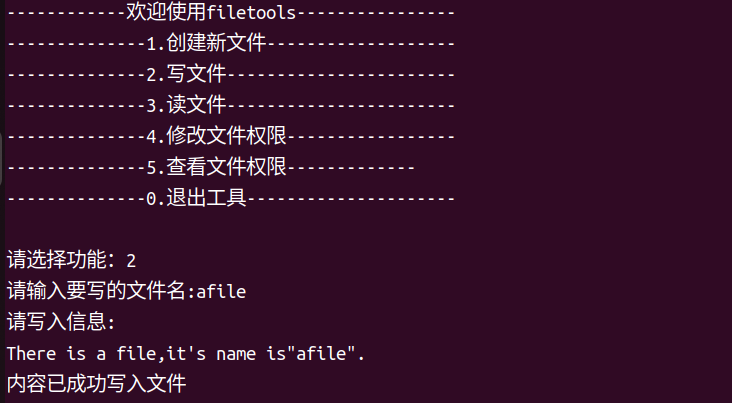
## 4.程序运行结果与分析

### 4.1程序运行结果

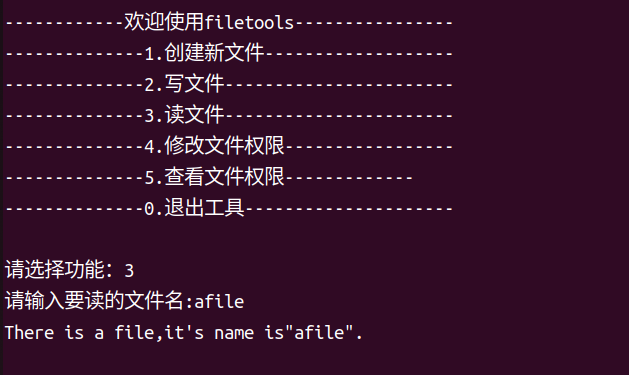
1.选择1，创建文件



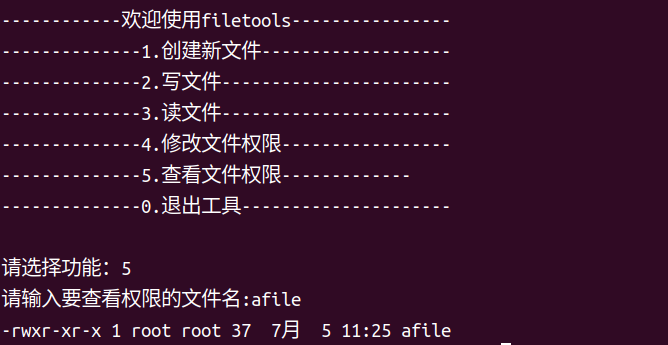
2.选择2，写文件



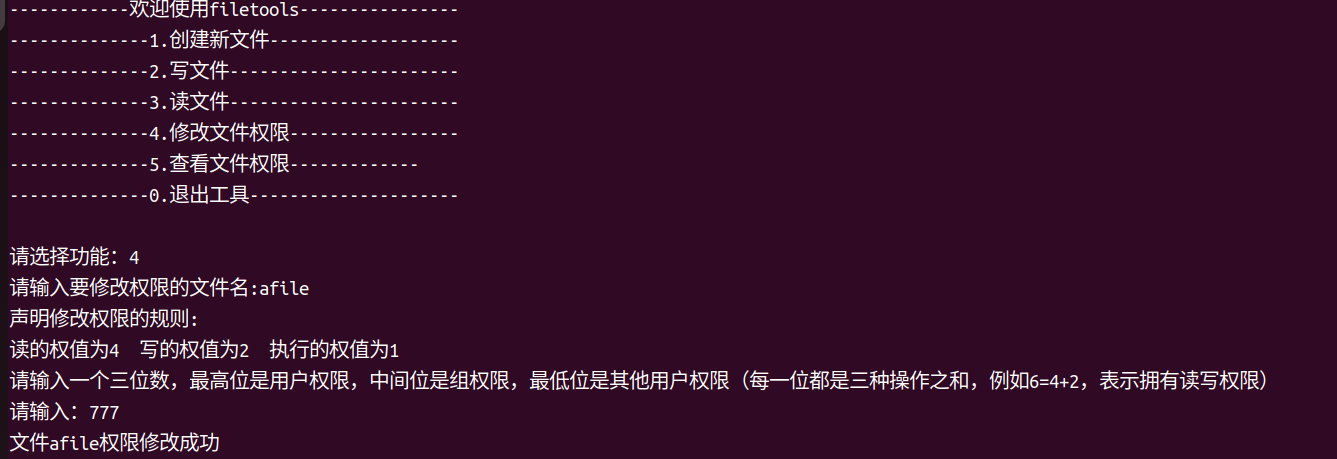
3.选择3，读文件



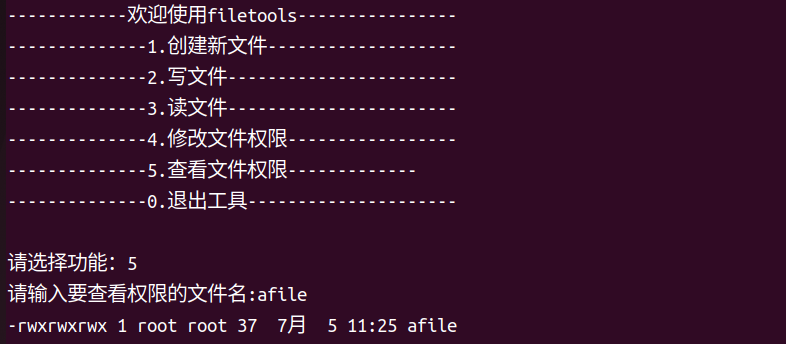
4.修改文权限前，先查看一次文件权限



5.选择4，修改权限



6.选择5，查看修改后的文件权限



### 4.2结果分析

运行功能1，2，3符合预期，文件先被建立，然后写入了内容，再选择读功能，可以读出被创建文件中被写入的内容。

在修改权限之前先查看了权限，rwx r-x r-x表示0755，组和其他没有写权限，这也与代码中创建文件的0755所吻合。

修改文件权限后又查看了文件的权限，rwx rwx rwx表示0777，即三部分拥有全部权限，这与输入的777吻合。

## 5.所遇问题与解决办法

### 5.1所遇问题

在一开始修改文件权限的操作中采用了对每一组采用指令的修改办法，例如

S\_IRWXU 用户（所有者）读、写和执行  
S\_IRUSR 用户（所有者）读  
S\_IWUSR 用户（所有者）写  
S\_IXUSR 用户（所有者）执行

该方法有很大弊端，因为这仅仅是一组中的内容，三组会产生27种组合，这么实现显然会显得麻烦。

执行查看权限后退出程序，认为是代码问题，导致进行了很多无用的检查。

### 5.2解决办法

查询资料了解到，execv("/bin/ls", pargv);进行了系统调用，执行了外部程序，所以现程序被直接终止。

在修改权限上，选择采用三位的八进制数来进行修改，这样可以非常简单的实现多种权限的排列和组合，利用语句也较为容易实现，int IfSuccess = chmod(Path, mode);这一句即可实现。

# 实验四 进程管理

## 1.题目及要求

### 1.1实验目的

1.理解进程的概念，明确进程和程序的区别。

2.理解并发执行的实质。

3.掌握进程的同步，撤销等进程控制方法。

### 1.2实验内容

父进程使用系统调用pipe()建立一个管道，然后使用系统调用fork()创建两个子进程：子进程1和子进程2。

子进程1每隔1秒通过管道向子进程2发送数据：I send message x times.（x初值为1，以后发送一次后做加一操作），子进程2从管道读出信息，并显示在屏幕上。

父进程用系统调用signal()来捕捉来自键盘的中断信号SIGINT（即按Ctrl+C键,）；当捕捉到中断信号后，父进程用系统调用kill()向两个子进程发出信号，子进程捕捉到信号后分别输出如下信息后终止：

Child Process 1 is killed by Parent!

Child Process 2 is killed by Parent!

父进程等待两个子进程终止后，释放管道并输出如下的信息后终止

Parent Process is Killed!

## 2.程序功能与设计思路

### 2.1程序功能

当程序开始执行时，每隔一秒子进程1通过管道向子进程2发送信息，子进程2从管道读出消息并显示。

当按下Ctrl+c时，两个子进程被终止，且父进程也被终止。

### 2.2设计思路

通过进行系统调用，创建一个进程，并在该进程下建立两个子进程。子进程1和子进程2要忽略终止信号，仅仅接收父进程传来的信号并执行对应操作函数。进程的创建失败和成功要添加提醒。

同时，为了避免出现僵尸进程，要将子进程1和子进程2收回。

## 3.数据结构与算法设计

### 3.1数据结构

定义进程ID和管道的描述符

1. pid\_t P1;
2. pid\_t P2;
3. **int** PipeFd[2];

### 3.2算法设计

1.引用的头文件

1. #include <stdio.h>      // 包含标准输入输出库
2. #include <stdlib.h>     // 包含标准库，提供一些通用函数如exit
3. #include <unistd.h>     // 包含Unix标准函数库，提供read, write, sleep等
4. #include <string.h>     // 包含字符串操作函数库
5. #include <signal.h>     // 包含信号处理库
6. #include <sys/types.h>  // 包含系统类型定义
7. #include <sys/wait.h>   // 包含等待子进程状态改变的函数库

2.两个处理函数

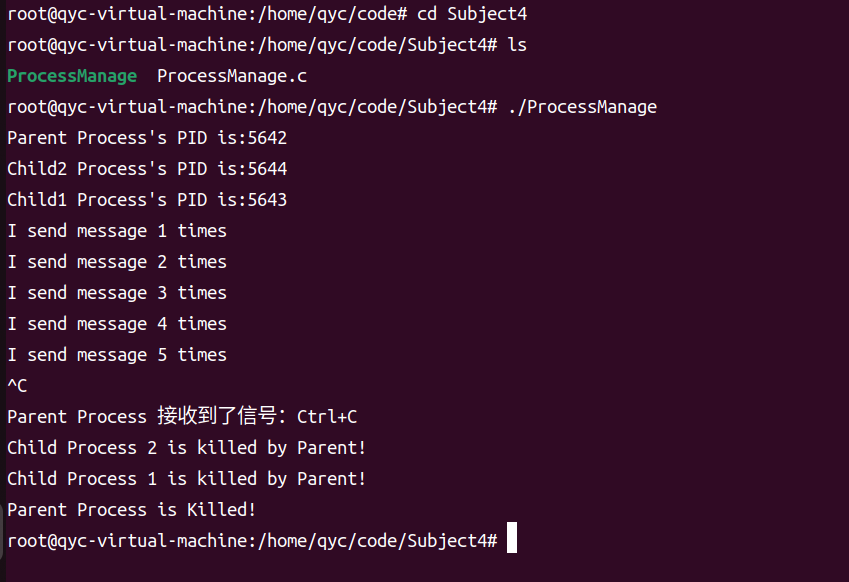
1. **void** Handle1(**int** s) {
2. printf("\nParent Process 接收到了信号：Ctrl+C\n");
3. **if** (s == SIGINT) {
4. kill(P1, SIGUSR1);     //向P1传送SIGUSR1,
5. kill(P2, SIGUSR2);
6. }
7. }
8. **void** Handle2(**int** s) {
9. close(PipeFd[0]);
10. close(PipeFd[1]);
11. **if** (P1==0 && s == SIGUSR1) {
12. printf("Child Process 1 is killed by Parent!\n");
13. exit(0);
14. }
15. **if** ( P2==0 && s == SIGUSR2) {
16. printf("Child Process 2 is killed by Parent!\n");
17. exit(0);
18. }
19. }

3.主函数部分

1. **int** main() {
2. **int** number = 1;
3. pid\_t getpid();     //创建父进
4. **char** Buffer1[MAX];
5. **char** Buffer2[MAX];
6. printf("Parent Process's PID is:%d\n", getpid());
7. //pipe创建管道，成功返回0，失败返回-1
8. **if** (pipe(PipeFd) < 0) {
9. printf("未能成功创建管道\n");
10. **return** -1;
11. }
12. signal(SIGINT, Handle1);    //signal()函数，第一个参数表示处理的信号，第二个参数是函数指针，指向处理函数
14. P1 = fork();//创建子进程
15. **if** (P1 == -1) {
16. perror("子进程1创建失败；");
17. exit(EXIT\_FAILURE);
18. }
19. **if** (P1 == 0) {
20. printf("Child1 Process's PID is:%d\n", getpid());
21. signal(SIGINT, SIG\_IGN);    //忽略ctrl+C
22. signal(SIGUSR1, Handle2);
23. **while** (1) {
24. close(PipeFd[0]);      //0表示读管道，1表示写管道
25. sprintf(Buffer1, "I send message %d times", number); //第一个参数是目标字符串，第二个参数是内容
26. write(PipeFd[1], Buffer1, MAX);   //P1是负责写的，所以关闭读管道
27. number++; //
28. sleep(1);
29. }
30. }
31. **else** **if** (P1 > 0) {
32. P2 = fork();
33. **if** (P2 == -1) {
34. perror("子进程2创建失败；");
35. exit(EXIT\_FAILURE);
36. }
37. **if** (P2 == 0) {
38. printf("Child2 Process's PID is:%d\n", getpid());
39. signal(SIGINT, SIG\_IGN);
40. signal(SIGUSR2, Handle2);
41. **while** (1) {
42. close(PipeFd[1]);    //P2是要读的，所以关闭写管道，读管道打开
43. read(PipeFd[0], Buffer2, MAX);
44. printf("%s\n", Buffer2);
45. }
46. }
47. **else** {
48. waitpid(P1, NULL, 0);    //避免僵尸进程,要将其收回
49. waitpid(P2, NULL, 0);
50. close(PipeFd[0]);
51. close(PipeFd[1]);
52. printf("Parent Process is Killed!\n");
53. }
54. }
55. **return** 0;
56. }

## 4.程序运行结果与分析

### 4.1程序运行结果



### 4.2结果分析

程序运行结果符合预期，每一秒进程1向进程2发送信息，并被进程2输出。当按下终止符时，提示接收到了信号，收回两个子进程，最后杀死主进程。

## 5.所遇问题与解决办法

### 5.1所遇问题

一开始没有考虑收回两个子进程，导致printf("Child Process 1 is killed by Parent!\n");和printf("Child Process 2 is killed by Parent!\n");后没有下文，这其实是产生了僵尸进程。

一开始没有考虑全面，没有给主进程结束编写提示。

优化不全面，当子进程1写时，其实读端是不需要的，但一开始没有关闭。

### 5.2解决办法

向管道中写的进程关闭读端，读管道的关闭写端，提高效率。

添加代码片段：

1. **else** {
2. waitpid(P1, NULL, 0);    //避免僵尸进程,要将其收回
3. waitpid(P2, NULL, 0);
4. close(PipeFd[0]);
5. close(PipeFd[1]);
6. printf("Parent Process is Killed!\n");
7. }

这样可以收回不用进程，避免出现僵尸进程，同时添加了主进程被杀死的提示。

# 实验五 进程通信

## 1.题目及要求

### 1.1实验目的

1.理解管道机制、消息缓冲队列、共享存储区机制进行进程间的通信。

2.理解通信机制。

### 1.2实验内容

编写一主程序可以由用户选择如下三种进程通信方式

1.使用管道来实现父子进程之间的进程通信：

子进程向父进程发送自己的进程标识符，以及字符串“is sending a message to parent”。父进程则通过管道读出子进程发来的消息，将消息显示在屏幕上，然后终止。

2.使用消息缓冲队列来实现 client 进程和 server 进程之间的通信：

server 进程先建立一个关键字为 SVKEY（如 75）的消息队列，然后等待接收类型为 REQ（例如 1）的消息；在收到请求消息后，它便显示字符串“serving for client”和接收到的 client 进程的进程标识数，表示正在为 client 进程服务；然后再向 client 进程发送应答消息，该消息的类型是 client 进程的进程标识数，而正文则是 server 进程自己的标识ID。client 进程则向消息队列发送类型为 REQ 的消息（消息的正文为自己的进程标识 ID） 以取得 sever 进程的服务，并等待 server 进程发来的应答；然后显示字符串“receive reply from”和接收到的 server 进程的标识 ID。

3.使用共享存储区来实现两个进程之间的进程通信：

进程 A 创建一个长度为 512 字节的共享内存，并显示写入该共享内存的数据；进程 B 将共享内存附加到自己的地址空间，并向共享内存中写入数据。

## 2.程序功能与设计思路

### 2.1程序功能

1.通过菜单实现对功能的选择，包括对通信方式的选择和退出功能。

2.通过管道实现进程通信，父进程要读出并显示子进程发出的消息。

3.消息缓冲队列上要指出服务端和客户端的进程ID，输出提供服务和就收服务的两端各自的进程ID。

4.在共享存储区中由进程B向其中写入内容，并由进程A读出并显示。

### 2.2设计思路

1.在管道通信中，模仿第四题的思路，当进程只需要写时关闭读端，需要读时关闭写端。最后要收回发送信息的子进程，避免出现僵尸进程。

2.在消息缓冲队列通信中，需要各自定义服务端和客户端的函数。在设计中，我将发送的消息类型和回复的消息类型设置为不同，这样更加严谨。通过系统调用实现向消息队列中信息的发送与接收。当错误时，要抛出错误。最后在整体函数中依然要收回两个进程。

3.在共享存储区中，定义进程A和进程B，B向共享存储区中写入信息，A读其中的信息。对应进程结束后要脱离共享存储区，并给出提示。在整体函数中要考虑回收两个进程。

4.通过循环反复实现对菜单的使用，直到最后退出。

## 3.数据结构与算法设计

### 3.1数据结构

消息队列中对消息结构体的定义

1. **struct** msgbuf {
2. **long** mtype;
3. **char** mtext[MaxSize];
4. }msg;

管道通信中对管道描述的定义

1. **int** Pipefd[2];

### 3.2算法设计

1.头文件与宏

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4. #include<unistd.h>
5. #include<sys/types.h>
6. #include<sys/ipc.h>
7. #include<sys/msg.h>
8. #include<sys/shm.h>
9. #include<wait.h>
11. #define MaxSize 256
12. #define SVKEY 75   //关键字,这里消息缓冲队列和共享存储区使用了同一个关键字，但两者没有任何关系
13. #define RequestType 4321    //消息缓冲队列中的请求信息类型
14. #define ResponseType 1234    //消息缓冲队列中的回复请求类型

2.管道通信

1. //管道通信
2. **void** Pipe(){
3. **int** Pipefd[2];
4. pid\_t Parent= getpid();
5. printf("父进程的PID为：%d\n",Parent);
6. **if** (pipe(Pipefd) < 0) {
7. printf("Pipe创建失败");
8. **return**;
9. }
10. pid\_t Child;
11. Child = fork();
12. **if** (Child == 0) {
13. **char** Information[50];
14. close(Pipefd[0]);   //写，因此关闭读端
15. sprintf(Information, "Child PID:%d is sending a meaasge to parent", getpid());
16. write(Pipefd[1], Information, 50);
17. sleep(1);
18. exit(0);
19. }
20. **else** {
21. close(Pipefd[1]);   //读，关闭写端
22. **char** Buffer[50];
23. read(Pipefd[0], Buffer, 50);
24. printf("%s\n", Buffer);
25. close(Pipefd[0]);
26. close(Pipefd[1]);
27. waitpid(Child, NULL, 0);
28. printf("Child Process is over!\n");
29. }
30. }

3.消息队列中，服务端客户端函数及整体函数

1. **void** Client() {
2. **int** QID = msgget(SVKEY, 0777 | IPC\_CREAT); // 获取或创建消息队列，第一个参数是关键字，第二个参数是权限掩码以及打开方式
3. **if** (QID == -1) {
4. perror("客户端：获取消息或创建消息队列失败");
5. exit(EXIT\_FAILURE);
6. }
7. pid\_t Cpid = getpid();
8. **struct** msgbuf msg;
9. **int**\* Cint = (**int**\*)msg.mtext;   //将进程号（原为字符串类型），转化为整形类型
10. \*Cint = Cpid;
11. msg.mtype = RequestType;    //定义请求类型
12. **if** (msgsnd(QID, &msg, **sizeof**(**int**), 0) == -1) {    //第一个参数是标识符，第二个是接收消息结构体的指针，第三个是长度，第四个是消息类型，0则表示都可以接收
13. perror("客户端：消息发送失败");
14. exit(EXIT\_FAILURE);
15. }
16. //第一个参数是标识符，第二个接收消息的结构的指针,第三个长度，第四个消息类型
17. msgrcv(QID, &msg, **sizeof**(**int**), ResponseType, 0); // 接收服务端回应
18. // 检查接收到的消息类型是否为RESPONSE\_TYPE
19. **if** (msg.mtype == ResponseType) {
20. printf("客户端PID:%d 接收到了服务端的回应 服务端PID:%d\n", Cpid, \*(**int**\*)msg.mtext);
21. }
22. **else** {
23. fprintf(stderr, "客户端:未接收到预期回应类型\n");
24. exit(EXIT\_FAILURE);
25. }
26. exit(0);
27. }
29. **void** Server() {
30. **int** QID = msgget(SVKEY, 0777 | IPC\_CREAT);
31. **if** (QID == -1) {
32. perror("服务端：获取消息或创建消息队列失败");
33. exit(EXIT\_FAILURE);
34. }
35. **struct** msgbuf msg;
36. **if** (msgrcv(QID, &msg, MaxSize, RequestType, 0) == -1) { // 接收类型为1的消息
37. perror("服务端：消息接收失败");
38. exit(EXIT\_FAILURE);
39. }
40. pid\_t Spid = \*(**int**\*)msg.mtext;
41. printf("服务端PID:%d 为客户端提供了服务 客户端PID:%d\n", getpid(), Spid);
43. // 发送回应给客户端
44. msg.mtype = ResponseType; // 使用相同的消息类型
45. \*(**int**\*)msg.mtext = getpid();
46. **if** (msgsnd(QID, &msg, **sizeof**(**int**), 0) == -1) {
47. perror("服务端：消息发送失败");
48. exit(EXIT\_FAILURE);
49. }
51. exit(0);
52. }
54. **void** Bufferqueues() {
55. pid\_t pid1, pid2;
56. **do** {
57. pid1 = fork();
58. } **while** (pid1 == -1); // 处理 fork 失败的情况
60. **if** (pid1 == 0) {
61. Server(); // 子进程1执行服务端逻辑
62. exit(0); // 子进程执行完毕后退出
63. }
65. **do** {
66. pid2 = fork();
67. } **while** (pid2 == -1); // 处理 fork 失败的情况
69. **if** (pid2 == 0) {
70. Client(); // 子进程2执行客户端逻辑
71. exit(0); // 子进程执行完毕后退出
72. }
73. wait(NULL); // 等待子进程1结束
74. wait(NULL); // 等待子进程2结束
75. }

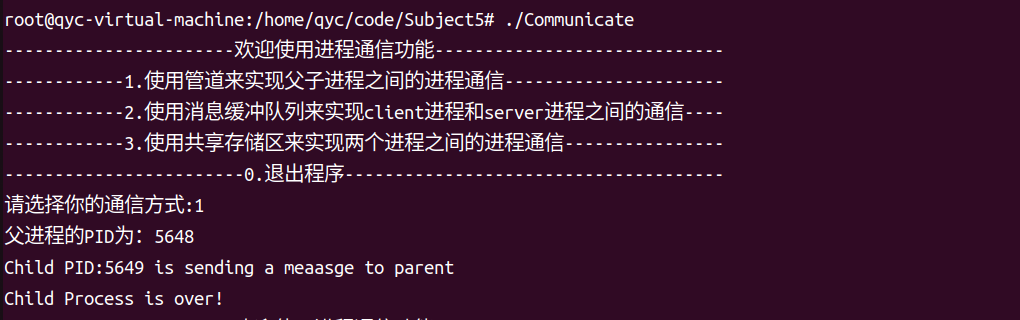
4.共享存储区，两个进程的函数及整体函数

1. **void** ProcessA() {
2. **int** SID = shmget(SVKEY, 512, 0777 | IPC\_CREAT);   //第一个参数是关键字，第二个参数是大小，第三个参数是权限掩码以及打开方式
3. **char**\* addr = shmat(SID, 0, 0);  //将共享内存段附加到调用进程的地址空间  第一个参数标识符，第二个是附加的地址，第三个是附着的操作的标志
4. printf("共享存储区内容:%s\n", addr);
5. **if** (shmdt(addr) == 0) {
6. printf("进程A脱离了共享存储区，与共享内存区断开了连接\n");
7. }
8. exit(0);
9. }
11. **void** ProcessB() {
12. **int** SID = shmget(SVKEY, 512, 0777 | IPC\_CREAT);
13. **char**\* addr = shmat(SID, 0, 0);
14. **char**\* messages[] = { "进程B写入了信息"};
15. memset(addr, '\0', 512);//addr内容初始化
16. strncpy(addr, messages[0], 512);   //将一个字符串复制到另一个字符串中
17. **if** (shmdt(addr) == 0) {
18. printf("进程B脱离了共享存储区，与共享内存区断开了连接\n");
19. }
20. exit(0);
22. }
24. **void** Sharedmemory() {
25. pid\_t pid1, pid2;
26. **do** {
27. pid1 = fork();
28. } **while** (pid1 == -1); // 处理 fork 失败的情况
30. **if** (pid1 == 0) {
31. ProcessB();
32. exit(0);
33. }
35. **do** {
36. pid2 = fork();
37. } **while** (pid2 == -1);
39. **if** (pid2 == 0) {
40. ProcessA();
41. exit(0);
42. }
43. wait(NULL);
44. wait(NULL);
45. }

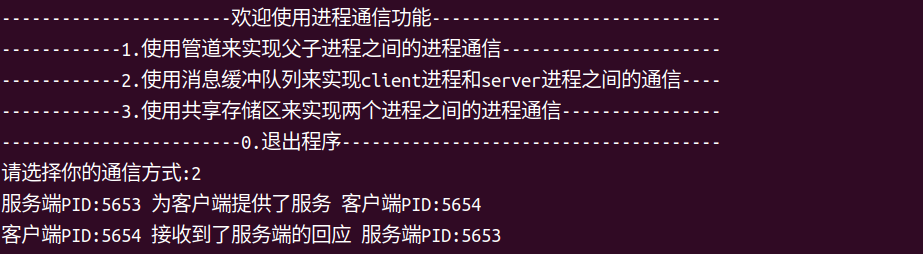
## 4.程序运行结果与分析

### 4.1程序运行结果

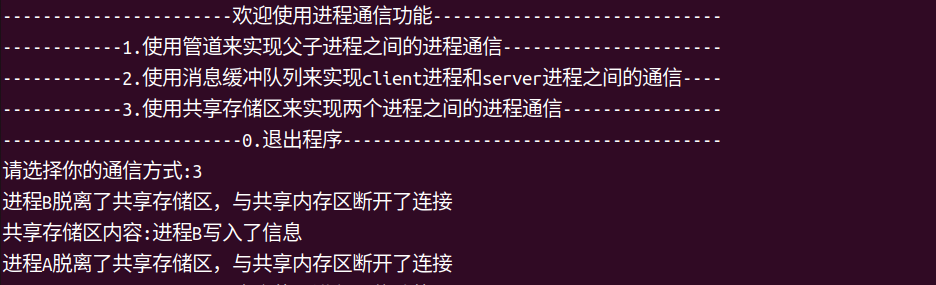
1.选择1，管道通信



2.选择2，消息队列



3.选择3，共享存储区



### 4.2结果分析

运行结果正确，符合预期。

在管道通信中，父子进程的ID都被输出，消息成功接收，最后成功收回了进程。

在消息队列中，客户端服务端ID被成功输出，且接收和提供服务过程被成功打出。

在共享存储区中，B写入的信息被成功读出，且两个进程成功脱离了共享存储区。

## 5.所遇问题与解决办法

### 5.1所遇问题

一开始在消息队列的部分，回复类型和接收类型是一样的，因此在阅读代码造成了很大的困扰，同时这样也可能会产生无法预知的错误。

### 5.2解决办法

定义了两个不同的消息类型：

#define RequestType 4321    //消息缓冲队列中的请求信息类型

#define ResponseType 1234    //消息缓冲队列中的回复请求类型

同时受到前面几题的指示，每一部分都选择收回子进程避免僵尸进程。对于读写端的开放与关闭，做出合理的安排，避免降低效率。

# 心得体会

在这次操作系统课程设计中，我经历了从对Linux几乎一无所知到对其有了基本了解的转变。一开始，面对Linux系统的陌生命令和操作界面，我感到有些手足无措。但随着设计的推进，我逐渐学会了如何在Linux环境下进行基本的编程和调试工作，这个过程让我对操作系统有了更为深刻的认识。

通过实验，我不仅学习了作业调度和磁盘调度的理论知识，还亲自编写程序实现了这些算法。这让我对算法的工作原理和实际应用有了更直观的理解。我也开始理解了操作系统是如何通过这些调度算法来提高系统资源的利用率和响应速度。

在文件系统调用的实验中，我学会了如何在Linux系统中进行文件的创建、读写和权限管理。这些基本的文件操作技能对于我作为一名编程学习者来说是非常宝贵的。

进程管理的实验让我对进程的概念、生命周期和进程间的通信有了初步的认识。我学会了使用管道进行父子进程间的通信，以及如何利用消息队列和共享内存这些IPC机制来实现不同进程之间的数据交换。

特别值得一提的是，我学会了使用GDB调试器。在之前的学习中，我对于程序调试的概念非常模糊，但通过这次实践，我能够设置断点、逐步执行程序、观察变量值，这些技能极大地提高了我解决编程问题的能力。

总的来说，这次课程设计是一次充满挑战的学习经历。它不仅让我对操作系统的工作原理有了初步的理解，也锻炼了我的编程和调试技能。我相信，这些基础知识和实践经验将为我未来的学习打下坚实的基础，并在我的编程生涯中发挥重要作用。