

《操作系统课程设计》

实习报告

班级	<u>191181</u>
学号	<u>20181001095</u>
姓名	<u>常文瀚</u>
院系	<u>计算机学院</u>
专业	<u>计算机科学与技术</u>
指导老师	<u>张求明</u>
完成日期	<u>2020 年 12 月 15 日</u>

目录

一、作业调度.....	1
1.1 实验目的.....	1
1.2 实验内容.....	1
1.3 设计思路.....	1
1.3.1 算法.....	1
1.3.2 设计.....	2
1.4 具体代码.....	2
1.5 运行结果.....	10
1.6 总结	10
二、磁盘调度.....	11
2.1 实验目的.....	11
2.2 实验内容.....	11
2.3 设计思路.....	11
2.3.1 算法.....	11
2.3.2 设计.....	12
2.4 具体代码.....	12
2.5 运行结果.....	15
2.6 总结	16
三、熟悉 Linux 文件系统调用	16
3.1 实验目的.....	16
3.2 实验内容.....	16
3.3 设计思路.....	16
3.3.1 实现功能.....	16
3.3.2 设计.....	17
3.3.3 具体函数功能	17
3.4 具体代码.....	18
3.5 运行结果.....	20
3.6 总结	22
四、进程管理.....	22
4.1 实验目的.....	22

4.2 实验内容.....	23
4.3 设计思路.....	23
4.3.1 函数.....	23
4.3.2 设计.....	25
4.4 集体代码.....	25
4.5 运行结果.....	28
4.6 总结	28
五、请求分页系统中的置换算法	28
5.1 实验目的.....	28
5.2 实验内容.....	28
5.3 设计思路.....	29
5.3.1 算法.....	29
5.3.2 设计.....	30
5.4 具体代码.....	30
5.5 运行结果.....	34
5.6 总结	35
六、进程通信	35
6.1 实验目的.....	35
6.2 实验内容.....	35
6.3 设计思路.....	36
6.3.1 预备知识.....	36
6.3.2 设计.....	40
6.4 具体代码.....	40
6.5 运行结果.....	47
6.6 总结	47
七、参考文献.....	47

一、作业调度

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。

1.3 设计思路

1.3.1 算法

- (1) 先来先服务算法

先来先服务调度算法是根据进程进入就绪队列的顺序来占用 cpu，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。当作业调度中采用该算法时，系统将按照作业到达的先后次序来进行调度，优先从后备队列中，选择一个或多个位于队列头部的作业，把他们调入内存，分配所需资源、创建进程，然后放入“就绪队列”，直到该进程运行到完成或发生某事件堵塞后，进程调度程序才将处理机分配给其他进程。

- (2) 短作业优先算法

优先调度并处理短作业，所谓短是指作业的运行时间短。而在作业未投入运行时，并不能知道它实际的运行时间的长短，因此需要用户在提交作业时同时提交作业运行时间的估计值。

- (3) 响应比高者优先算法

高响应比优先调度算法, 就是为每个作业引入动态优先权，并使作业的优先权随着等待时间的增加而以速率 a 提高，保证长作业在等待一定的时间后。必然有机会分配到处理机。作业调度方式为非抢占式。优先权公式定义如下：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$$

1.3.2 设计

- (1) 设计结构体 PCB 存储每个作业 ID、到达时间、运行时间、结束时间、周转时间、带权周转时间
- (2) 使用 vector 容器存储作业块，先到先服务，短作业优先，高响应比优先，以及 4 个临时存储容器
- (3) 先来先服务算法，将作业按照到达时间从小到大排序，各元素依次出队列。
- (4) 短作业优先算法，将作业按照到达时间从小到大排序后按照运行时间从小到大排序，各元素依次出队列
- (5) 高响应比优先算法，按到达时间从小到大排序，保存两个结束时间，小于更小的结束时间的作业进入第二个队列，计算各作业的响应比从小到大排序，取末尾的元素运行，运行结束的元素保存到输出队列

1.4 具体代码

```
bool cometimePaixu(const PCB &p1, const PCB &p2) {
    return p1.ComeTime < p2.ComeTime;
}

void display(vector<PCB> &List) {
    int totalTurnoverTime = 0;
    double totalWeightTime = 0;
    cout<<setw(10)<<"进程名: ";
    for (vector<PCB>::iterator i = List.begin(); i<List.end(); i++)
        cout << setw(6) << (*i).ID;
    cout << endl;

    cout << setw(11) << "到达时间: ";
    for (vector<PCB>::iterator i = List.begin(); i<List.end(); i++)
        cout << setw(6) << (*i).ComeTime;
    cout << endl;

    cout << setw(11) << "运行时间: ";
```

```

for (vector<PCB>::iterator i = List.begin(); i < List.end(); i++)
    cout << setw(6) << (*i).RunTime;
cout << endl;

cout << setw(11) << "结束时间: ";
for (vector<PCB>::iterator i = List.begin(); i < List.end(); i++)
    cout << setw(6) << (*i).FinishTime;
cout << endl;

cout << setw(11) << "周转时间: ";
for (vector<PCB>::iterator i = List.begin(); i < List.end(); i++)
    {
        cout << setw(6) << (*i).TurnoverTime;
        totalTurnoverTime += (*i).TurnoverTime;
    }
cout << endl;

cout << setw(14) << "带权周转时间: ";
for (vector<PCB>::iterator i = List.begin(); i < List.end(); i++)
    {
        cout << setw(6) << fixed << setprecision(2) << (*i).WeightTurnoverTime;
        totalWeightTime += (*i).WeightTurnoverTime;
    }
cout << endl;

cout << "平均周转时间: " << fixed << setprecision(2) << ((double)totalTurnoverTime /
List.size()) << endl;
cout << "平均带权周转时间: " << fixed << setprecision(2) << ((double)totalWeightTime /
List.size()) << endl;

}

void FCFS(vector<PCB> &list) {
    sort(list.begin(), list.end(), cometimePaixu);
    //display(list);
    int finishTimeMin, finishTimeMax;

```

```

        vector<PCB>::iterator i;
        i = list.begin(); //first
//#####完成第一个
#####

        (*i).FinishTime = (*i).ComeTime + (*i).RunTime;
        (*i).TurnoverTime = (*i).RunTime;
        (*i).WeightTurnoverTime = (double)(*i).TurnoverTime/(double)(*i).RunTime;
//#####完成第二个
#####

        i++;
        (*i).FinishTime = (*i).ComeTime + (*i).RunTime;
        (*i).TurnoverTime = (*i).RunTime;
        (*i).WeightTurnoverTime = (double)(*i).TurnoverTime / (double)(*i).RunTime;
//#####存储两个结束时间
#####

        finishTimeMin = min(list[0].FinishTime,list[1].FinishTime);
        finishTimeMax = max(list[0].FinishTime, list[1].FinishTime);
//#####从第三个开始
#####

        i++;
        while (i != list.end()) {
            if ((*i).ComeTime<=finishTimeMin) {//如果下一个进程比最小结束时间早
                (*i).FinishTime = (*i).RunTime + finishTimeMin;
                (*i).TurnoverTime = (*i).FinishTime-(*i).ComeTime;
                (*i).WeightTurnoverTime = (double)(*i).TurnoverTime /
(double)(*i).RunTime;
                finishTimeMin = min(finishTimeMax, (*i).FinishTime);
                finishTimeMax = max(finishTimeMax, (*i).FinishTime);
                i++;
            }
            else {
                (*i).FinishTime = (*i).RunTime + (*i).ComeTime;
                (*i).TurnoverTime = (*i).FinishTime - (*i).ComeTime;
                (*i).WeightTurnoverTime = (double)(*i).TurnoverTime /
(double)(*i).RunTime;

```

```

        finishTimeMin = min(finishTimeMax, (*i).FinishTime);
        finishTimeMax = max(finishTimeMax, (*i).FinishTime);
        i++;
    }
}
display(list);
}
bool runtimePaixu(const PCB &p1, const PCB &p2) {
    return p1.RunTime > p2.RunTime;
}
bool xiangyingPaixu(const PCB &p1, const PCB &p2)
{
    return p1.WeightTurnoverTime < p2.WeightTurnoverTime;
}
void ShortFirst_HighFirst(vector<PCB> &list, vector<PCB> &list1, vector<PCB> &list2, int choice)
{
    int index = 0;
    sort(list.begin(), list.end(), cometimePaixu);
    //    display(list);
    int finishTimeMin, finishTimeMax;
    vector<PCB>::iterator i;
    i = list.begin(); //first
    #####完成第一个
    #####
        (*i).FinishTime = (*i).ComeTime + (*i).RunTime;
        (*i).TurnoverTime = (*i).RunTime;
        (*i).WeightTurnoverTime = (double)(*i).TurnoverTime / (double)(*i).RunTime;
    #####完成第二个
    #####
        i++;
        (*i).FinishTime = (*i).ComeTime + (*i).RunTime;
        (*i).TurnoverTime = (*i).RunTime;
        (*i).WeightTurnoverTime = (double)(*i).TurnoverTime / (double)(*i).RunTime;
    #####存储两个结束时间
    #####

```



```

        finishTimeMin = min(list[0].FinishTime, list[1].FinishTime);
        finishTimeMax = max(list[0].FinishTime, list[1].FinishTime);
//#####从第三个开始
#####
        list1.push_back(list[0]);
        list1.push_back(list[1]);
        i++;
        /*while (i != list.end()) {
            if ((*i).ComeTime <= finishTimeMin) {//如果下一个进程比最小结束时间早
                (*i).FinishTime = (*i).RunTime + finishTimeMin;
                (*i).TurnoverTime = (*i).FinishTime - (*i).ComeTime;
                (*i).WeightTurnoverTime = (double)(*i).TurnoverTime /
(double)(*i).RunTime;

                finishTimeMin = min(finishTimeMax, (*i).FinishTime);
                finishTimeMax = max(finishTimeMax, (*i).FinishTime);
                i++;
            }
            else {
                (*i).FinishTime = (*i).RunTime + (*i).ComeTime;
                (*i).TurnoverTime = (*i).FinishTime - (*i).ComeTime;
                (*i).WeightTurnoverTime = (double)(*i).TurnoverTime /
(double)(*i).RunTime;

                finishTimeMin = min(finishTimeMax, (*i).FinishTime);
                finishTimeMax = max(finishTimeMax, (*i).FinishTime);
                i++;
            }
        }*/
        for (; i < list.end();) //从第三个开始
        {
            if ((*i).RunTime == 0)break;

            if ((*i).ComeTime <= finishTimeMin)
            {
                if ((*i).RunTime == 0)break;
                while ((*i).ComeTime <= finishTimeMin)

```

以插进去

```
{
    if ((*i).RunTime == 0)break;

    list2.push_back(*i); //只要第一个结束了，在此之前到了的都可
    以插进去

    index++;
    i++;
    if (i == list.end()) break;
}
}
else if (!list2.empty())
{

}
else if ((*i).ComeTime <= finishTimeMax) {
    list2.push_back(*i);
    index++;
    i++;

    list2[index - 1].FinishTime = list2[index - 1].ComeTime + list2[index -
1].RunTime;

    list2[index - 1].TurnoverTime = list2[index - 1].FinishTime - list2[index -
1].ComeTime;

    list2[index - 1].WeightTurnoverTime = (double)list2[index -
1].TurnoverTime / (double)list2[index - 1].RunTime;

    finishTimeMin = min(finishTimeMax, list2[0].FinishTime);
    finishTimeMax = max(finishTimeMax, list2[0].FinishTime);

    list1.push_back(*(list2.end() - 1));
    list2.pop_back();
    index--;
    break;
}
else {
```

```

        list2.push_back(*i);
        index++;
        i++;
        list2.push_back(*i);
        index++;
        i++;

        for (int i = 0; i < index; i++)
        {
            list2[i].FinishTime = list2[i].ComeTime + list2[i].RunTime;
            list2[i].TurnoverTime = list2[i].FinishTime - list2[i].ComeTime;
            list2[i].WeightTurnoverTime = (double)list2[i].TurnoverTime /
(double)list2[i].RunTime;
        }
        finishTimeMin = min(list2[0].FinishTime, list2[1].FinishTime);
        finishTimeMax = max(list2[0].FinishTime, list2[1].FinishTime);

        list1.push_back(*(list2.end() - 1));
        list2.pop_back();
        index--;
        list1.push_back(*(list2.end() - 1));
        list2.pop_back();
        index--;
        break;
    }

    for (int i = 0; i < index; i++)
    {
        list2[i].FinishTime = finishTimeMin + list2[i].RunTime;
        list2[i].TurnoverTime = list2[i].FinishTime - list2[i].ComeTime;
        list2[i].WeightTurnoverTime = (double)list2[i].TurnoverTime /
(double)list2[i].RunTime;

    }

    if(choice==1)sort(list2.begin(), list2.end(), runtimePaixu);
    if(choice==2)sort(list2.begin(), list2.end(), xiangyingPaixu);

```

```

        finishTimeMin = min(finishTimeMax, list2[index - 1].FinishTime);
        finishTimeMax = max(finishTimeMax, list2[index - 1].FinishTime);

        list1.push_back(*(list2.end() - 1));
        list2.pop_back();
        index--;

    }

    while (!list2.empty())
    {
        for (int i = 0; i < index; i++)
        {
            list2[i].FinishTime = finishTimeMin + list2[i].RunTime;
            list2[i].TurnoverTime = list2[i].FinishTime - list2[i].ComeTime;
            list2[i].WeightTurnoverTime = (double)list2[i].TurnoverTime /
(double)list2[i].RunTime;
        }
        if (choice == 1)sort(list2.begin(), list2.end(), runtimePaixu);
        if (choice == 2)sort(list2.begin(), list2.end(), xiangyingPaixu);
        finishTimeMin = min(finishTimeMax, list2[index - 1].FinishTime);
        finishTimeMax = max(finishTimeMax, list2[index - 1].FinishTime);

        list1.push_back(*(list2.end() - 1));
        list2.pop_back();
        index--;
    }
    display(list1);
}

```

1.5 实验结果

先到先服务算法										
进程名:	A	B	G	C	H	D	I	J	E	F
到达时间:	0	2	4	5	6	7	8	10	12	15
运行时间:	7	10	8	20	20	30	10	12	40	8
结束时间:	7	12	15	32	35	62	45	57	97	70
周转时间:	7	10	11	27	29	55	37	47	85	55
带权周转时间:	1.00	1.00	1.38	1.35	1.45	1.83	3.70	3.92	2.12	6.88
平均周转时间:	36.30									
平均带权周转时间:	2.46									

短进程优先算法										
进程名:	A	B	G	I	F	J	H	C	D	E
到达时间:	0	2	4	8	15	10	6	5	7	12
运行时间:	7	10	8	10	8	12	20	20	30	40
结束时间:	7	12	15	22	23	34	43	54	73	94
周转时间:	7	10	11	14	8	24	37	49	66	82
带权周转时间:	1.00	1.00	1.38	1.40	1.00	2.00	1.85	2.45	2.20	2.05
平均周转时间:	30.80									
平均带权周转时间:	1.63									

高响应比优先算法										
进程名:	A	B	G	I	C	J	F	H	D	E
到达时间:	0	2	4	8	5	10	15	6	7	12
运行时间:	7	10	8	10	20	12	8	20	30	40
结束时间:	7	12	15	22	35	34	42	55	72	95
周转时间:	7	10	11	14	30	24	27	49	65	83
带权周转时间:	1.00	1.00	1.38	1.40	1.50	2.00	3.38	2.45	2.17	2.08
平均周转时间:	32.00									
平均带权周转时间:	1.83									

sh: pause: command not found										
root@iZh8461uthz90jZ:~/cwh1#										

1.6 总结

通过本次实验，我深入了解计算机内部的工作原理，理解计算机在作业调度上如何进行操作，这些基础知识，是一个计算机专业的学生进行更深层研究的重要一步，但是也在实验过程中发现了以下问题：

- (1) 对操作系统基础知识遗忘较多，很多概念分不清
- (2) 看错题目，没有看到是双作业同时进行，导致设计从一开始就错了，浪费了很多时间
- (3) 在和朋友比对实验结果发现自己的结果和大家的不一样，最短作业优先和高响应比优先算法的结果答案匹配不上，经过反思自己的问题所在。

二、磁盘调度

2.1 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。

2.2 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	209	0	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 9 号，向磁道号增加的方向移动。

2.3 设计思路

2.3.1 算法

- (1) 先来先服务算法

FIFO 算法，也就是先来先服务算法。这种算法的思想比较容易理解。假设当前磁道在某一位置，依次处理服务队列里的每一个磁道，这样做的优点是处理起来比较简单，但缺点是磁头移动的距离和平均移动距离会很大。

- (2) 最短寻道优先算法

这种算法的本质是利用贪心算法来实现，假设当前磁道在某一位置，接下来处理的是距离当前磁道最近的磁道号，处理完成之后再处理离这个磁道号最近的磁道号，直到所有的磁道号都服务完了程序结束。这样做的优点是性能会优于 FIFO 算法，但是会产生距离当前磁道较远的磁道号长期得不到服务，也就是“饥饿”现象，因为要求访问的服务的序列号是动态产生的，即各个应用程序可能不断地提出访问不同的磁道号的请求。

- (3) 电梯调度算法

很形象的电梯调度算法。先按照一个方向(比如从外向内扫描)，扫描的过程中依次访问要求服务的序列。当扫描到最里层的一个服务序列时反向扫描，这里要注意，假设最里层为 0 号磁道，最里面的一个要求服务的序列是 5 号，访问完 5 号之后，就反向了，不需要再往里扫。

结合电梯过程更好理解，在电梯往下接人的时候，明知道最下面一层是没有人的，它是不会再往下走的。

2.3.2 设计

- (1) 设计结构体 PCB 存储每个磁盘 ID、磁道号。
- (2) 引入一个 vector 容器存储所有的磁盘块。
- (3) 先来先服务算法，按表格顺序从小到大排序，依次取出服务。
- (4) 最短寻道优先算法，按磁道号顺序排序，分别比较距离出发点最近的两个服务比较，并逐步向两边扩散，依次比较。
- (5) 对于电梯扫描算法，按磁道号顺序排序，然后根据初始的方向，如果是上升方向，则从开始的磁盘好一直向上扫描直到最大号，然后调转方向，下降扫描直至最小号为止，如果初始方向是下降方向则相反。

2.4 具体代码

```
bool shortFirst(const PCB&p1,const PCB&p2) {
    return p1.num < p2.num;
}

void FCFS(int start) {
    sumDistance = 0;
    aveDistance = 0;
    cout << "先来先服务算法结果如下: " << endl;
    vector<PCB>::iterator i;
    i = list.begin();
    cout << start << "(起始点)";
    for (;i<list.end();i++) {
        cout << "—>" << (*i).ID << "(" << (*i).num << ")";
        sumDistance += abs(start-(*i).num);
        start = (*i).num;
    }
    aveDistance = (double)sumDistance / 14;
    cout << "\n 平均移动距离: " << aveDistance << endl;
}

void Shortest(int start) {
    sumDistance = 0;
    aveDistance = 0;
```

```

sort(list.begin(), list.end(), shortFirst);
cout << "最短寻道优先算法结果如下: " << endl;
vector<PCB>::iterator i;
i = list.begin();
int point=0;//定位在起始点两边最近的点的位置
while(i<list.end()){
    if ((*i).num <= start) {
        i++;
        point++;
    }
    else i++;
}
//i 的值为小于等于起始点的个数
int small=point-1;
int large=point;
int finish = 0; //记录已经写入点的个数
cout << start << "(起始点)";
while (finish < list.size()) {
    if (((large < list.size() && small >= 0) && (abs(start - list[small].num) <=
abs(list[large].num - start))) || large >= list.size()) {
        //if (abs(start - list[small].num) <= abs(list[large].num - start)) {
        cout << "——>" << list[small].ID << "(" << list[small].num << ")";
        sumDistance += abs(start - list[small].num);
        finish++;
        start = list[small].num;
        small--;
    }
    else if (((large < list.size() && small >= 0) && (abs(start - list[small].num) >
abs(list[large].num - start))) || small < 0) {
        cout << "——>" << list[large].ID << "(" << list[large].num << ")";
        sumDistance += abs(start - list[large].num);
        finish++;
        start = list[large].num;
        large++;
    }
}

```



```

    }
    aveDistance = (double)sumDistance / 14;
    cout << "\n 平均移动距离: " << aveDistance << endl;
}

void Scan(int start, char direction) {
    sumDistance = 0;
    aveDistance = 0;
    sort(list.begin(), list.end(), shortFirst);
    cout << "电梯调度算法结果如下: " << endl;
    vector<PCB>::iterator i;
    i = list.begin();
    int point = 0; //定位在起始点两边最近的点的位置
    while (i < list.end()) {
        if ((*i).num <= start) {
            i++;
            point++;
        }
        else i++;
    }
    //i 的值为小于等于起始点的个数
    int small = point - 1;
    int large = point;
    cout << start << "(起始点)";
    if (direction == 'r' || direction == 'R') {
        for (; large < list.size(); large++) {
            cout << "—" << list[large].ID << "(" << list[large].num << ")";
            sumDistance += abs(start - list[large].num);
            start = list[large].num;
        }
        for (; small >= 0; small--) {
            cout << "—" << list[small].ID << "(" << list[small].num << ")";
            sumDistance += abs(start - list[small].num);
            start = list[small].num;
        }
    }
}

```

```

    }
else {
    for (; small >= 0; small--) {
        cout << "—>" << list[small].ID << "(" << list[small].num << ")";
        sumDistance += abs(start - list[small].num);
        start = list[small].num;
    }
    for (; large < list.size(); large++) {
        cout << "—>" << list[large].ID << "(" << list[large].num << ")";
        sumDistance += abs(start - list[large].num);
        start = list[large].num;
    }
}

aveDistance = (double)sumDistance / 14;
cout << "\n 平均移动距离: " << aveDistance << endl;
}

```

2.5 实验结果

```

root@iZh8461uthz90jZ:~# cd cwh2
root@iZh8461uthz90jZ:~/cwh2# ls
cwh2  cwh2.cpp  obj.o
root@iZh8461uthz90jZ:~/cwh2# ./cwh2

*****
请输入你要使用的磁盘调度算法序号
1、先来先服务（FCFS）  2、最短寻道优先  3、电梯调度算法
Choice:1
输入起始的磁道号: 9
先来先服务算法结果如下:
9(起始点)->A(30)->B(50)->C(100)->D(180)->E(20)->F(90)->G(150)->H(70)->I(80)->J(10)->K(160)->L(120)->M(40)->N(110)
平均移动距离: 68.6429

*****
请输入你要使用的磁盘调度算法序号
1、先来先服务（FCFS）  2、最短寻道优先  3、电梯调度算法
Choice:2
输入起始的磁道号: 9
最短寻道优先算法结果如下:
9(起始点)->J(10)->E(20)->A(30)->M(40)->B(50)->H(70)->I(80)->F(90)->C(100)->N(110)->L(120)->G(150)->K(160)->D(180)
平均移动距离: 12.2143

*****
请输入你要使用的磁盘调度算法序号
1、先来先服务（FCFS）  2、最短寻道优先  3、电梯调度算法
Choice:3
输入起始的磁道号: 9
输入扫描方向(l/L)or(r/R):l
电梯调度算法结果如下:
9(起始点)->J(10)->E(20)->A(30)->M(40)->B(50)->H(70)->I(80)->F(90)->C(100)->N(110)->L(120)->G(150)->K(160)->D(180)
平均移动距离: 12.2143

```

2.6 总结

在实验的开始阶段，我对磁道调度算法已经遗忘了大部分，但是通过在网络上搜索资料 and 教学视频成功掌握了算法。在对抗遗忘的过程中，不断唤醒不久前学习过的知识十分令人兴奋！这带给了我很大的自信，已经独自完成课程设计的信心，在今后的学习中我也会不断的勉励自己。

三、熟悉 linux 文件系统调用

3.1 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。

3.2 实验内容

使用文件系统调用编写一个文件工具 filetools，使其具有以下功能：

- (1) 创建新文件
- (2) 写文件
- (3) 读文件
- (4) 修改文件权限
- (5) 查看当前文件权限
- (6) 退出

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，文件按可变记录文件组织，具体记录内容自行设计。

3.3 设计思路

3.3.1 实现功能

- (1) 显示功能选项，并给出选择相应功能的提示
- (2) 输入 1，可以创建新文件
- (3) 输入 2，可以写文件
- (3) 输入 3，可以读文件
- (4) 输入 4，可以修改文件权限
- (5) 输入 5，可以查看当前文件的权限并修改文件权限

(6) 输入 6，直接退出程序

3.3.2 设计

- (1) 退出功能调用 close()函数
- (2) 创建新文件可以调用 open()函数
- (3) 写文件功能可以调用 write()函数
- (4) 修改文件权限功能可以自定义 chmod()函数
- (5) 查看文件权限功能可以调用 exec()函数，运行”ls -l newfile”的功能

3.3.3 具体函数功能

(1) open()

a. 定义

```
int open(const char *pathname, int flags);  
int open(const char *pathname, int flags, mode_t mode);
```

b. 参数意义

pathname：被打开的文件名，可包含路径
flag ：文件打开的方式，参数可以通过“|”组合构成，但前 3 个参数不能互相重合。

参 数	说 明	参 数	说 明
S_IRUSR	所有者拥有读权限	S_IXGRP	群组拥有执行权限
S_IWUSR	所有者拥有写权限	S_IROTH	其他用户拥有读权限
S_IXUSR	所有者拥有执行权限	S_IWOT	其他用户拥有写权限
S_IRGRP	群组拥有读权限	S_IXOTH	其他用户拥有执行权限
S_IWGRP	群组拥有写权限		

(2) write()

a. 定义

```
ssize_t write(int fd,void *buf,size_t count)
```

b. 参数意义

fd: 文件描述符

buf : 指定存储器写入数据的缓冲区

count : 指定读出的字节数。函数返回值: 成功: 已写的字节数 -1 : 出错

3.4 具体代码

```
int main()
{
    int fileID;//file 的 ID
    int num; //字符个数
    int choice;
    char buffer[MAX]; //读取的缓冲队列
    char *path="/bin/ls";
    char *argv[4]={"ls","-l","file1",NULL};
    while(1)
    {
        cout<<"*****"<<endl;
        cout<<"1.创建新文件"<<endl;
        cout<<"2.写文件"<<endl;
        cout<<"3.读文件"<<endl;
        cout<<"4.修改文件权限"<<endl;
        cout<<"5.查看当前文件的权限"<<endl;

        cout<<"0.退出" << endl;

        cout<<"*****"<<endl;
        cin>>choice;
        switch(choice)
        {
            case 0:close(fileID);exit(0);
            case 1:
                fileID=open("file1",O_RDWR|O_TRUNC|O_CREAT,0750);//文件名，打开方式
                if(fileID==-1)
                {
                    cout<<"Failed!"<<endl;
                }
                else
```

```

        cout<<"Succeed! fileID = "<<fileID<<endl;//显示 file 的 ID
        break;
    case 2:
        num=read(0,buffer,MAX);//成功返回读取的字节数
        write(fileID,buffer,num);//从读入的信息送到 file1 里去
        break;
    case 3:
        read(fileID,buffer,MAX);
        write(1,buffer,num);//把 file1 文件的内容在屏幕上输出
        break;
    case 4:
        modify();
        cout<<"成功改变格式! "<<endl;
        break;
    case 5:
        execv(path,argv);//执行 ls -l file1
        break;
    default:
        cout<<"error choice!"<<endl;

    }
}

return 0;
}

int modify()
{
    int c;
    mode_t mode=S_IWUSR;
    cout<<"0.所有用户读写执行"<<endl;
    cout<<"1.所有用户只可读"<<endl;
    cout<<"2.所有用户只可写"<<endl;
    cout<<"3.所有用户只可执行"<<endl;

```

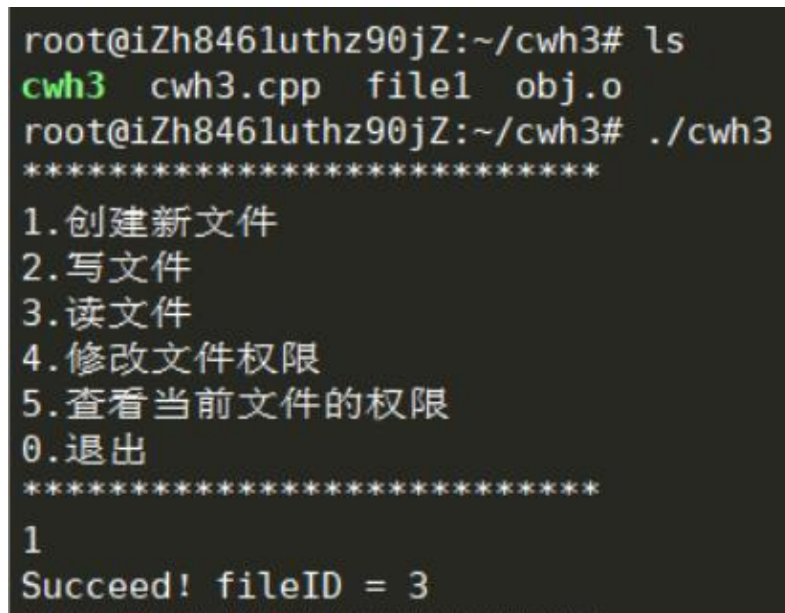
```

cout<<"4.用户组读写执行"<<endl;
cout<<"5.用户组只可读"<<endl;
cout<<"6.用户组只可写"<<endl;
cout<<"7.用户组只可执行"<<endl;
cin>>c;
switch(c)
{
    case 0:chmod("file1",S_IRWXU);break;
    case 1:chmod("file1",S_IRUSR);break;
    case 2:chmod("file1",S_IWUSR);break;
    case 3:chmod("file1",S_IXUSR);break;
    case 4:chmod("file1",S_IRWXG);break;
    case 5:chmod("file1",S_IRGRP);break;
    case 6:chmod("file1",S_IWGRP);break;
    case 7:chmod("file1",S_IXGRP);break;
    default:printf("error choice!\n");
}
return 0;
}

```

3.5 实验结果

3.5.1 运行效果



```

root@iZh846luthz90jZ:~/cwh3# ls
cwh3  cwh3.cpp  file1  obj.o
root@iZh846luthz90jZ:~/cwh3# ./cwh3
*****
1.创建新文件
2.写文件
3.读文件
4.修改文件权限
5.查看当前文件的权限
0.退出
*****
1
Succeed! fileID = 3

```

```
*****
1.创建新文件
2.写文件
3.读文件
4.修改文件权限
5.查看当前文件的权限
0.退出
*****
2
ChangWenhan OS 20181001095
*****
```

```
1.创建新文件
2.写文件
3.读文件
4.修改文件权限
5.查看当前文件的权限
0.退出
*****
3
ChangWenhan OS 20181001095
*****
```

```
*****
1.创建新文件
2.写文件
3.读文件
4.修改文件权限
5.查看当前文件的权限
0.退出
*****
4
0.所有用户读写执行
1.所有用户只可读
2.所有用户只可写
3.所有用户只可执行
4.用户组读写执行
5.用户组只可读
6.用户组只可写
7.用户组只可执行
0
成功改变格式！
*****
```



```
*****
1.创建新文件
2.写文件
3.读文件
4.修改文件权限
5.查看当前文件的权限
0.退出
*****
5
-rwx----- 1 root root 27 Dec 17 19:16 file1
root@iZh846luthz90jZ:~/cwh3#
```

3.5.2 文件权限说明

- rw----- (600) -- 只有属主有读写权限。
- rw-r--r-- (644) -- 只有属主有读写权限；而属组用户和其他用户只有读权限。
- rwx----- (700) -- 只有属主有读、写、执行权限。
- rwxr-xr-x (755) -- 属主有读、写、执行权限；而属组用户和其他用户只有读、执行权限。
- rwx--x--x (711) -- 属主有读、写、执行权限；而属组用户和其他用户只有执行权限。
- rw-rw-rw- (666) -- 所有用户都有文件读、写权限。这种做法不可取。
- rwxrwxrwx (777) -- 所有用户都有读、写、执行权限。更不可取的做法。

3.6 总结

这一部分的知识点对我来说是全新的知识点，因此需要大量的自学。参考网络上的学习资料，我有了一定的了解，收获很多。实验的主要难度是对 `exec()`、`write()`、`read()`、`chmod()` 这些函数的学习与应用，我逐渐熟悉了 Linux 操作系统对文件的操作，理解了 linux 中的文件管理的基本原理，并深感 Linux 系统的强大。

四、进程管理

4.1 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。

4.2 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!

4.3 设计思路

4.3.1 函数

(1) 管道

管道是 Linux 支持的最初 Unix IPC 形式之一，具有以下特点：

- a. 管道是半双工的，数据只能向一个方向流动（用 `lockf` 或关闭文件），需要双向通信时，需要建立起两个管道；
- b. 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
- c. 单独构成一种独立的文件系统：

管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中；

- d. 数据的读出和写入：

一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

管道通过系统调用 `pipe()` 来实现，函数的功能和实现过程如下：

- e. 原型：`int pipe(int fd[2])`
- f. 返回值：如果系统调用成功，返回 0。如果系统调用失败返回-1：

(2) `fork()`

- a. 创建一个新进程。
- b. 系统调用格式：`Pid=fork()`
- c. `fork()` 返回值意义如下：

0: 在子进程中, pid 变量保存的 fork() 返回值为 0, 表示当前进程是子进程。

>0: 在父进程中, pid 变量保存的 fork() 返回值为子进程的 id 值。

-1: 创建失败。

如果 fork() 调用成功, 它向父进程返回子进程的 pid, 并向子进程返回 0, 即 fork() 被调用了一次, 但返回了两次。此时 OS 在内存中建立一个新进程, 所建的新进程是调用 fork() 父进程 (parent process) 的副本, 称为子进程 (child process)。子进程继承了父进程的许多特性, 并具有父进程完全相同的用户级上下文, 父进程与子进程并发执行。内核为 fork() 完成以下操作:

① 为新进程分配一进程表项和进程标识符

进入 fork() 后, 内核检查系统是否有足够的资源来建立一个新进程。若资源不足, 则 fork() 系统调用失败; 否则, 核心为新进程分配一进程表项和唯一的进程标识符。

② 检查同时运行的进程数目

超过预先规定的最大数目时, fork() 系统调用失败。

③ 拷贝进程表项中的数据

将父进程的当前目录和所有已打开的数据拷贝到子进程表项中, 并置进程的状态为“创建”状态。

④ 子进程继承父进程的所有文件

对父进程当前目录和所有已打开的文件表项中的引用计数加 1。

⑤ 为子进程创建进程上、下文

进程创建结束, 设子进程状态为“内存中就绪”并返回子进程的标识符。

⑥ 子进程执行

虽然父进程与子进程程序完全相同, 但每个进程都有自己的程序计数器 PC, 然后根据 Pid 变量保存的 fork() 返回值的不同, 执行了不同的分支语句。

(3) exec()

系统调用 exec() 系列, 也可用于新程序的运行。fork() 只是将父进程的用户级上下文拷贝到新进程中, 而 exec() 系列可以将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上, 以更改新进程的用户级上下文。exec() 系列中的系统调用都完成相同的功能, 它们把一个新程序装入内存, 来改变调用进程的执行代码, 从而形成新进程。如果 exec() 调用成功后, 没有任何数据返回, 这与 fork() 不同。exec() 系列调用在 LINUX 系统库 unistd.h 中, 共有 execl、execlp、execv、execvp 五个, 其基本功能相同, 只是以不同的方式来给出参数。

(4) wait()

等待子进程运行结束。如果子进程没有完成，父进程一直等待。wait()将调用进程挂起，直至其子进程因暂停或终止而发来软中断信号为止。如果在 wait()前已有子进程暂停或终止，则调用进程做适当处理后便返回。核心对 wait()作以下处理：

- a. 首先查找调用进程是否有子进程，若无，则返回出错码；
- b. 若找到一处于“僵死状态”的子进程，则将子进程的执行时间加到父进程的执行时间上，并释放子进程的进程表项；
- c. 若未找到处于“僵死状态”的子进程，则调用进程便在可被中断的优先级上睡眠，等待其子进程发来软中断信号时被唤醒。

4.3.2 设计

- (1) 创建管道；
- (2) 设置软终端信号；
- (3) 创建子进程；
- (4) 子进程 1 每隔 1 秒向子进程 2 发送数据，然后由子进程 2 读取数据并显示在屏幕上；
- (5) 父进程可以捕捉中断信号，kill 两个进程，子进程被 kill 后，父进程也会被 kill；
- (4) 关闭管道。

4.4 具体代码

```
int main()
{

    char buffer[40];
    char info[40];
    int counter=1;
    printf("Process Parent pid %d\n",getpid());
    //创建无名管道
    if(pipe(filedis)<0)
    {
        printf("Create pip failed\n");
        return -1;
    }
    //设置软中断信号
    signal(SIGINT,SignHandler1); //要处理的信号,处理的方式
    //创建子进程 1 ， 2
```

```

child1=fork();
if(child1==0)
{
    printf("child1 pid %d\n",getpid());
    signal(SIGINT,SIG_IGN);/*SIGINT  终止进程  中断进程*/
    signal(SIGUSR1,SignHandler2);/*SIGUSR1  终止进程  用户定义信号 1
SIGUSR2  终止进程  用户定义信号 2*/
    while(1)
    {
        close(filedis[0]);
        sprintf(info,"child1 sends message %d \n",counter);
        write(filedis[1],info,30);
        counter++;
        sleep(1);
    }
}
else if(child1>0)//返回主进程
{
    child2=fork();//创建子进程 2
    if(child2==0)
    {
        printf("child2 pid %d\n",getpid());
        signal(SIGINT,SIG_IGN);
        signal(SIGUSR1,SignHandler2);
        while(1)
        {
            close(filedis[1]);
            read(filedis[0],buffer,40);
            printf("%s",buffer);
        }
    }
    //等待进程 1,2 退出
    waitpid(child1,NULL,0);
    printf("Child process 1 is over\n");
    waitpid(child2,NULL,0);
}

```

```

    printf("Child process 2 is over\n");
    //关闭管道
    close(filedis[0]);
    close(filedis[1]);
    printf("Parent process is killed\n");
}
return 0;
}

void SignHandler1(int iSignNo)
{
    printf("\nParent gets Ctrl+C\n");
    if(iSignNo==SIGINT) //传递 SIGUSR 信号给子进程
    {
        kill(child1,SIGUSR1);
        kill(child2,SIGUSR2);
    }
}

void SignHandler2(int iSignNo)
{
    close(filedis[0]);
    close(filedis[1]);
    if(child1==0&& iSignNo==SIGUSR1)
    {
        printf("Child process 1 is killed by Parent!\n");
        exit(0);
    }
    if(child2==0&& iSignNo==SIGUSR1)
    {
        printf("Child process 2 is killed by Parent!\n");
        exit(0);
    }
}
}

```

4.5 实验结果

```
root@iZh846luthz90jZ:~# cd cwh4
root@iZh846luthz90jZ:~/cwh4# ls
cwh4  cwh4.cpp  obj.o
root@iZh846luthz90jZ:~/cwh4# ./cwh4
Process Parent pid 60546
child2 pid 60548
child1 pid 60547
child1 sends message 1
child1 sends message 2
child1 sends message 3
child1 sends message 4
child1 sends message 5
child1 sends message 6
^C
Parent gets Ctrl+C
Child process 1 is killed by Parent!
Child process 1 is over
Child process 2 is over
Parent process is killed
```

4.6 总结

经过新知识的学习，本次实验是对进程创建函数，等待进程终止函数，以及软中断通信管道的使用。想法很明了，但是使用不够严谨，过程中出了不少错误。管道必须先调用 `open()` 将其打开，想要不导致阻塞，用读写方式 (`O_RDWR`) 打开，以只读方式或者以写方式打开时，进程将会被阻塞，直到对方打开管道。

五、请求分页系统中的置换算法

5.1 实验目的

- (1) 了解虚拟存储技术的特点；
- (2) 掌握请求分页系统的页面置换算法。

5.2 实验内容

- (1) 通过如下方法产生一指令序列，共 320 条指令
 - a. 在 $[1, 32k-2]$ 的指令地址之间随机选取一起点, 访问 M ;
 - b. 顺序访问 $M+1$;
 - c. 在 $[0, M-1]$ 中随机选取 $M1$, 访问 $M1$;
 - d. 顺序访问 $M1+1$;

- e. 在 $[M1+2, 32k-2]$ 中随机选取 $M2$ ，访问 $M2$ ；
- f. 顺序访问 $M2+1$ ；
- g. 重复 A—F，直到执行 320 次指令。

(2) 指令序列变换成页地址流设

- a. 页面大小为 1K；
- b. 分配给用户的内存页块个数为 4 页到 32 页, 步长为 1 页；
- c. 用户虚存容量为 32K。

(3) 计算并输出下述各种算法在不同内存页块下的命中率

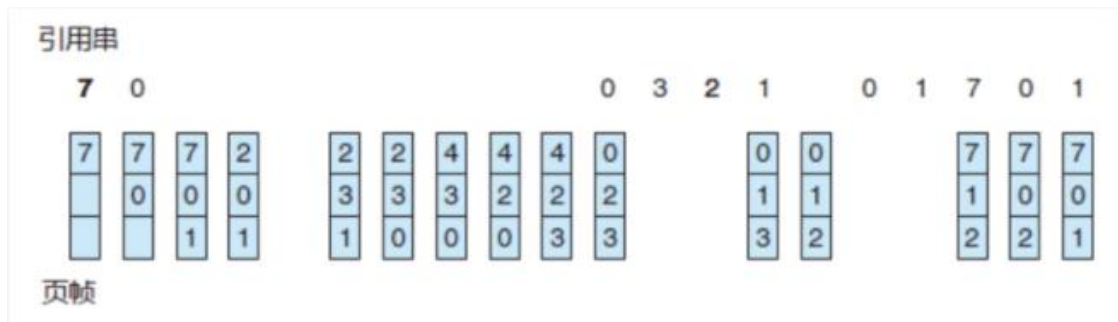
- a. 先进先出 (FIFO) 页面置换算法
- b. 最近最久未使用 (LRU) 页面置换算法
- c. 最佳 (Optimal) 页面置换算法

5.3 设计思路

5.3.1 算法

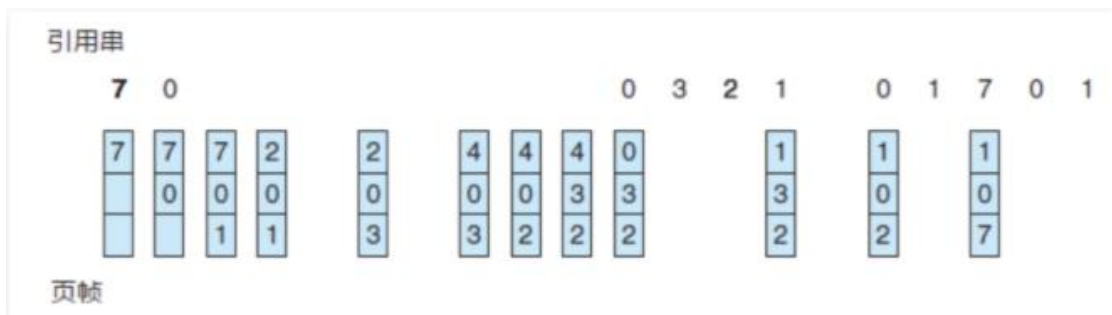
(1) 先进先出 (FIFO) 页面置换算法

该算法总是淘汰最新进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。该算法实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指针，称为替换指针，使它总是指向最老的页面。



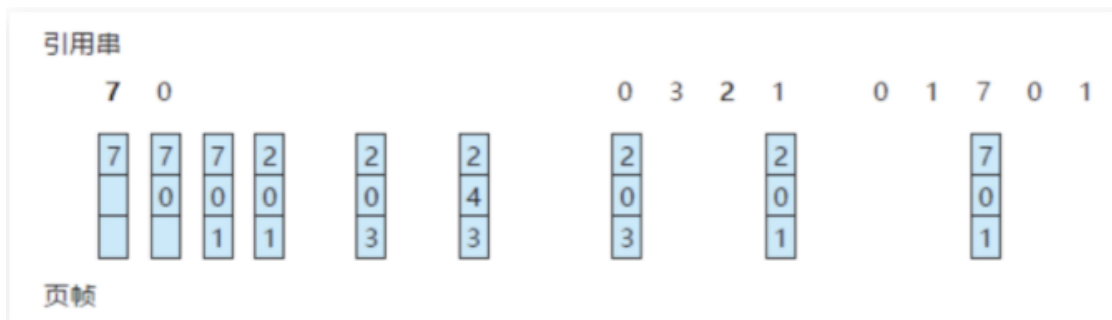
(2) 最近最久未使用 (LRU) 页面置换算法

最近最久未使用 (LRU) 页面置换算法，是根据页面调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，因此，LRU 置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t ，当需淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。



(3) 最佳 (Optimal) 页面置换算法

该算法选择的被淘汰页面，将是以后永远不使用的，或许是在最长（未来）时间内不再被访问的页面。采用该算法，通常可保证获得最低的缺页率。但由于人们目前还无法预知一个进程在内存的若干个页面中，哪一个页面是未来最长时间内不再被访问的，因而该算法是无法实现的，但可以利用该算法去评价其他算法。



5.3.2 设计

- (1) 设计结构体 BLOCK 存储每个指令的页号和访问字段表示多久未被访问；
- (2) 用 random 函数生成随机数的功能生成 320 条随机指令，用数组 temp[320]来存储这 320 条随机数；
- (3) 设计最佳置换算法、最近最久未使用置换算法、先进先出置换算法
- (4) 利用最佳置换算法 (OPT) 模拟页面置换过程；
- (5) 利用最近最久未使用置换算法 (LRU) 模拟页面置换过程；
- (6) 利用先进先出置换算法 (FIFO) 模拟页面置换过程。

5.4 具体代码

```
struct node{
    int id;
    int num;
};

void get_input(){
```

```

int cnt=0;
srand(time(0));
while(cnt<N){
    int m=rand()%max_add+1;
    ins[cnt++]=m;
    if(cnt>=N)break;
    ins[cnt++]=m+1;
    if(cnt>=N)break;
    int m1=rand()%m;
    ins[cnt++]=m1;
    if(cnt>=N)break;
    ins[cnt++]=m1+1;
    if(cnt>=N)break;
    int m2=rand()%(max_add-1-m1)+(m1+2);
    ins[cnt++]=m2;
    if(cnt>=N)break;
    ins[cnt++]=m2+1;
    if(cnt>=N)break;
}
for(int i=0;i<N;i++){
    ins[i]=ins[i]/1024;//得到所在的
}
// for(int i=0;i<N;i++){
//     cout<<ins[i]<<" ";
// }
}

double FIFO(int page){
    hit_num=0;
    list<int>temp;
    for(int i=0;i<page;i++){
        temp.push_back(-1);
    }
    for(int i=0;i<N;i++){
        list<int>::iterator it=find(temp.begin(),temp.end(),ins[i]);

```

```

        if(it==temp.end()){
            temp.pop_front();
            temp.push_back(ins[i]);
        }else{
            hit_num++;
        }
    }
    hit_num=hit_num/N;
    hit_num*=100;
    return hit_num;
}

```

```

double LRU(int page){
    hit_num=0;
    list<node>temp;
    for(int i=0;i<page;i++){
        temp.push_back(node{-1,0});
    }
    list<node>::iterator it;
    for(int i=0;i<N;i++){
        for(it=temp.begin();it!=temp.end();it++){
            it->num++;
        }
        bool flag=false;
        for(it=temp.begin();it!=temp.end();it++){
            if(it->id==ins[i]){
                flag=true;
                it->num=0;
                hit_num++;
                break;
            }
        }
    }
    if(!flag){
        //找到一个最大的
        list<node>::iterator max_it=temp.begin();
    }
}

```

```

        for(it=temp.begin();it!=temp.end();it++){
            if(it->num>max_it->num){
                max_it=it;
            }
        }
        temp.erase(max_it);
        temp.push_back(node{ins[i],0});
    }
}

hit_num=hit_num*1.0/N;
hit_num*=100;
return hit_num;
}

double OPT(int page){
    list<int>temp;
    hit_num=0;
    for(int i=0;i<page;i++){
        temp.push_back(-1);
    }
    for(int i=0;i<N;i++){
        list<int>::iterator it=find(temp.begin(),temp.end(),ins[i]);
        if(it==temp.end()){
            int max_id=-1;

            list<int>::iterator ans;
            for(it=temp.begin();it!=temp.end();it++){
                //初始化
                int cur=0x3f3f3f3f;
                for(int j=i+1;j<N;j++){
                    if(ins[j]==*it){
                        cur=j;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        if(cur>max_id){
            max_id=cur;
            ans=it;
        }
    }
    temp.erase(ans);
    temp.push_back(ins[i]);

}
else{
    hit_num++;
}
}
hit_num=hit_num*1.0/N;
hit_num*=100;
return hit_num;
}

```

5.5 实验结果

```

root@iZh846luthz90jZ:~# ls
cppMNIST cwh1 cwh2 cwh3 cwh4 cwh5 cwh6 install.sh
root@iZh846luthz90jZ:~# cd cwh5
root@iZh846luthz90jZ:~/cwh5# ./cwh5

```

内存页块	FIFO	LRU	OPT
4	55.31	55.31	64.06
5	56.56	56.25	66.56
6	58.13	57.81	69.06
7	58.13	58.75	70.94
8	58.75	60.00	72.50
9	60.00	60.62	74.06
10	60.94	61.25	75.31
11	61.56	62.19	76.56
12	64.06	63.75	77.81
13	67.19	65.31	79.06
14	68.12	66.88	80.00
15	69.38	69.06	80.94
16	69.38	70.00	81.88
17	70.00	71.25	82.81
18	70.31	73.12	83.75
19	70.31	74.38	84.69
20	70.94	75.00	85.31
21	71.88	75.62	85.94
22	79.06	76.56	86.56
23	79.38	77.81	87.19
24	81.25	78.75	87.81
25	82.50	80.31	88.44
26	84.38	81.88	88.75
27	86.56	83.12	89.06
28	86.56	85.62	89.38
29	88.12	87.19	89.69
30	88.75	88.12	90.00
31	89.06	89.38	90.00
32	90.00	90.00	90.00

5.6 总结

页面置换算法的思想可以说比较简单，易懂，但是在实现的时候，我却遇到了很多的问题，比如说在找空闲物理块的时候，起初我是比较物理块是否等于 0，若为 0，则直接把页面放入，后来发现不论什么时候都是把 0 替换出去，才恍然大悟，既然页面标号有 0，就不能用 0 来表示空闲物理块，后来就换成用-1 来表示物理块空闲了。还有许多问题，在同学的帮助下都一解决了，感觉进步了好多好多，也学到了好多东西，都是当初学习编程语言时候所没有体会到的东西，受益良多。

六、进程通信

6.1 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。

6.2 实验内容

编写程序可以由用户选择如下三种进程通信方式：

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。

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。

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。

6.3 设计思路

6.3.1 预备知识

(1) 管道

a. 管道是 Linux 支持的最初 Unix IPC 形式之一，具有以下特点：

① 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；

② 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；

③ 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。

④ 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

b. 管道通过系统调用 `pipe()` 来实现，函数的功能和实现过程如下。

① 原型：`int pipe(int fd[2])`

② 返回值：如果系统调用成功，返回 0。如果系统调用失败返回-1： `errno = EMFILE`（没有空闲的文件描述符）

`ENFILE`（系统文件表已满）`EFAULT`（`fd` 数组无效）

③ 注意：`fd[0]` 用于读取管道，`fd[1]` 用于写入管道。

(2) 消息队列

消息队列是消息的链接表，包括 Posix 消息队列 systemV 消息队列。它克服了另外两种通信方式中信息量有限的缺点，对消息队列具有写权限的进程可以向消息队列中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。消息队列通过系统调用 `msgget()`、`msgsnd()`、`msgrcv()` 来实现，函数的功能和实现过程分别如下：

a. `msgget()`

① 功能：该函数用来创建和访问一个消息队列。

② 原型：`int msgget (key_t key, int msgflg)`

③ 参数：

`key`：消息队列关联的键。

`msgflg`：消息队列的建立标志和存取权限。

④ 返回值：成功执行时，返回消息队列标识值。失败返回-1，`errno` 被设为以下的某个值：

EACCES : 指定的消息队列已存在, 但调用进程没有权限访问它, 而且不拥有 CAP_IPCOWNER 权能。

EEXIST: key 指定的消息队列已存在, 而 msgflg 中同时指定 IPC_CREAT 和 IPC_EXCL 标志。

ENOENT: key 指定的消息队列不存在同时 msgflg 中不指定 IPC_CREAT 标志。

ENOMEM: 需要建立消息队列, 但内存不足。

ENOSPC: 需要建立消息队列, 但已达到系统的限制。

b. msgsnd()

① 功能: 该函数用来把消息添加到消息队列中。

② 原型: `ssize_t msgsnd(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);`

③ 参数:

msqid: 消息队列的识别码。

msgp: 指向消息缓冲区的指针, 此位置用来暂时存储发送和接收的消息, 是一个用户可定义的通用结构。

msgsz: 消息的大小。

msgtyp: 从消息队列内读取的消息形态。如果值为零, 则表示消息队列中的所有消息都会被读取。

msgflg: 用来指明核心程序在队列没有数据的情况下所应采取的行动。如果 msgflg 和常数 IPC_NOWAIT 合用, 则在消息队列呈空时, 不做等待马上返回-1, 并设定错误码为 ENOMSG。当 msgflg 为 0 时, msgsnd() 在队列呈满或呈空的情形时, 采取阻塞等待的处理模式。

④ 返回值: 当成功执行时, msgsnd() 返回拷贝到 mtext 数组的实际字节数。失败返回-1, errno 被设为以下的某个值:

E2BIG: 消息文本长度大于 msgsz, 并且 msgflg 中没有指定 MSG_NOERROR。

EACCES: 调用进程没有读权能, 同时没具有 CAP_IPCOWNER 权能。

EAGAIN: 消息队列为空, 并且 msgflg 中没有指定 IPC_NOWAIT。

EFAULT: msgp 指向的空间不可访问。

EIDRM: 当进程睡眠等待接收消息时, 消息已被删除。

EINTR: 当进程睡眠等待接收消息时, 被信号中断。

EINVAL: 参数无效。

ENOMSG: msgflg 中指定了 IPC_NOWAIT, 同时所请求类型的消息不存在。

c. `msgrcv()`

① 功能：该函数用来从一个消息队列获取消息。

② 原型：

```
int msgrcv(int msgid, void *msgptr, size_t msg_st, long int msgtype, int msgflg);
```

③ 参数：

`msgid`, `msgptr`, `msgst` 的作用也函数 `msgsnd()` 函数的一样。

`msgtype` 可以实现一种简单的接收优先级。如果 `msgtype` 为 0，就获取队列中的第一个消息。如果它的值大于零，将获取具有相同消息类型的第一个信息。如果它小于零，就获取类型等于或小于 `msgtype` 的绝对值的第一个消息。

`msgflg` 用于控制当队列中没有相应类型的消息可以接收时将发生的事情。

④ 返回值：调用成功时，该函数返回放到接收缓存区中的字节数，消息被复制到由

`msg_ptr` 指向的用户分配的缓存区中，然后删除消息队列中的对应消息。失败时返回-1。

(3) 共享内存

共享内存是操作系统常采用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。共享内存通过系统调用 `shmget()`、`shmat()`、`shmdt()`、`shmctl()` 来实现，函数的功能和实现过程分别如下。

a. `shmget()`

① 功能：该函数用来创建共享内存

② 原型：`int shmget(key_t key, int size, int shmflg)`

③ 参数：

第一个参数，与信号量的 `semget` 函数一样，程序需要提供一个参数 `key`（非 0 整数），它有效地为共享内存段命名，`shmget()` 函数成功时返回一个与 `key` 相关的共享内存标识符（非负整数），用于后续的共享内存函数。调用失败返回-1。

不相关的进程可以通过该函数的返回值访问同一共享内存，它代表程序可能要使用的某个资源，程序对所有共享内存的访问都是间接的，程序先通过调用 `shmget()` 函数并提供一个键，再由系统生成一个相应的共享内存标识符（`shmget()` 函数的返回值），只有 `shmget()` 函数才直接使用信号量键，所有其他的信号量函数使用由 `semget` 函数返回的信号量标识符。

第二个参数，`size` 以字节为单位指定需要共享的内存容量。

第三个参数，`shmflg` 是权限标志，它的作用与 `open` 函数的 `mode` 参数一样，如果要想在 `key` 标识的共享内存不存在时，创建它的话，可以与 `IPC_CREAT` 做或操作。共享内存的权限标

志与文件的读写权限一样，举例来说，0644, 它表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。

④ 返回值：

成功返回共享内存的标识符；不成功返回-1，errno 储存错误原因。

EINVAL 参数 size 小于 SHMMIN 或大于 SHMMAX。

EXIST 预建立 key 所致的共享内存，但已经存在。

EIDRM 参数 key 所致的共享内存已经删除。

ENOSPC 超过了系统允许建立的共享内存的最大值(SHMALL)。

ENOENT 参数 key 所指的共享内存不存在，参数 shmflg 也未设 IPC_CREAT 位。

EACCES 没有权限。

ENOMEM 核心内存不足。

b. shmat()

① 功能：第一次创建完共享内存时，它还不能被任何进程访问，shmat()函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间

② 原型：void* shmat(int shmid, const void *shmaddr, int shmflg);

③ 参数：

shmid：共享存储区的标识符；

shmaddr：用户给定的，将共享存储区附接到进程的虚拟地址空间；

shmflg：规定共享存储区的读、写权限，以及系统是否应对用户规定的地址做舍入操作。

其值为 SHMRDONLY 时，表示只能读，其值为 0 时，表示可读、可写，其值为 SHMRND（取整）时，表示操作系统在必要时舍去这个地址。

② 返回值：该系统调用的返回值为共享存储区所附接到的进程虚地址。

c. shmdt()

① 功能：该函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。

② 原型：int shmdt(void *shmaddr)

③ 参数：shmaddr：要断开连接的虚地址，亦即以前由连接的系统调用 shmat（）所返回的虚地址。

④ 返回值：调用成功时，返回 0 值，调用不成功，返回-1 值。

d. shmctl()

① 功能：与信号量的 semctl()函数一样，用来控制共享内存

② 原型: `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`

③ 参数: `shmid`: 由 `shmget` 所返回的标识符;

④ 返回值: 调用成功则返回 0, 如果失败则返回-1。

6.3.2 设计

(1) 初始化控制台界面;

(2) 首先运行管道通信, 使用 `fork()` 函数创建子进程, 在 `fork()` 返回子进程编号时设置父进程在管道读端等待;

(3) 子进程获取进程编号, 发送信息至管道写端, 并写入, 关闭写端;

(4) 父进程读取到消息, 将消息打印在控制台, 关闭管道, 运行结束。

(5) 运行消息队列通信选项, 使用 `fork()` 函数创建子进程, 在 `fork()` 返回子进程编号时设置 `server` 在队头进行等待;

(6) 子进程发送自己的进程编号, 即将信息送入消息队列, 父进程收到消息, 并传回自己的进程编号, 删除消息队列;

(7) `server` 打印出自己正在服务的对象的进程编号, `client` 打印出给予自己服务的 `server` 进程编号;

(8) 创建共享内存以及子进程, 进程 B 将信息写入共享内存, 切断进程与共享内存的连接, 打印写入的信息;

(9) 激活进程 A 对共享内存的控制, 读取存入内存的消息, 打印在控制台上。

6.4 具体代码

//采用管道通信

```
void pipeCon()
{
    int status;
    int fd[2];
    if(pipe(fd)==-1)
    {
        cout<<"创建管道错误"<<endl;
        return ;
    }
    pid_t pid=fork();
    if(pid==-1)
    {
```

```

        cout<<"创建子进程失败"<<endl;
        return ;
    }
    else if(pid==0)
    {
        auto id=getpid();
        string s=to_string(id);
        s+=" is sending a message to parent";
        //char *m="is sending a message to parent";
        char *m=new char[s.size()+2];
        s.copy(m,s.size(),0);
        m[s.size()]=0;
        close(fd[0]);
        write(fd[1],m,strlen(m));
        close(fd[1]);
        return ;
        //exit(0);
    }
    else
    {
        close(fd[1]);
        char p[100];
        int len=read(fd[0],p,sizeof(p));
        p[len]=0;
        cout<<p<<endl;
        close(fd[0]);
        wait(&status);
    }
    return ;
}

```

```

#define MAX_TEXT 512
struct msg_st
{
    long int msg_type;

```

```

    char text[MAX_TEXT];
};

//采用消息队列
void client();
void server();
void messQue()
{

    //client();
    pid_t pid=fork();
    if(pid==0)
        client();
    else
    {
        sleep(1);
        server();
        wait(NULL);
    }
    return ;
}

void client()
{
    int running = 1;
    struct msg_st data;
    //data.msg_type=1;
    char buffer[BUFSIZ];
    int msgid = -1;

    //建立消息队列
    msgid = msgget((key_t)75, 0666 | IPC_CREAT);
    if(msgid == -1)
    {
        cout<<"建立消息队列出现错误"<<endl;
    }
}

```

```

        exit(1);
    }

    string s=to_string(getpid());
    //cout<<s;
    s.copy(data.text,s.size(),0);
    //cout<<data.text;
    data.msg_type = 1;
    //向队列发送数据
    if(msgsnd(msgid, (void*)&data, MAX_TEXT, IPC_NOWAIT) == -1)
    {
        cout<< "消息发送失败"<<endl;;
        exit(1);
    }
    struct msg_st receive;
    long int msgtype=getpid();
    if(msgrcv(msgid, (void*)&receive, BUFSIZ, msgtype, 0) == -1)
    {
        cout<<"消息接收失败"<<endl;
        exit(1);
    }
    else
    {
        cout<<"receive reply from: "<<receive.text<<endl;
    }

    sleep(1);

    exit(0);

}

void server()
{
    int running = 1;

```

```

int msgid = -1;
struct msg_st data;
long int msgtype = 1;

//建立消息队列

msgid = msgget((key_t)75, 0666|IPC_CREAT);
if(msgid == -1)
{
    cout<<"消息队列建立失败"<<endl;
    exit(1);
}

if(msgrcv(msgid, (void*)&data, BUFSIZ, msgtype, 0) == -1)
{
    cout<<"获取消息失败"<<endl;
    exit(1);
}
else
{
    cout<<"serving for client: "<<data.text<<endl;

    int remsgid=stoi((string)data.text);
    //cout<<remsgid;
    struct msg_st reply;
    reply.msg_type=remsgid;
    string s=to_string(getpid());
    //cout<<s;
    s.copy(reply.text,s.size(),0);
    //reply.text
    if(msgsnd(msgid,&reply,MAX_TEXT, IPC_NOWAIT)==-1)
    {
        cout<<"server 发送应答消息错误"<<endl;
        exit(1);
    }
}
}

```

```

// }
// //删除消息队列
if(msgctl(msgid, IPC_RMID, 0) == -1)
{
    cout<<"消息队列删除错误"<<endl;
    exit(1);
}
exit(0);
}

```

//采用共享内存方式

```

void createMem();
void writeMem();
void shareMem()
{
    pid_t pid=fork();
    if(pid==-1)
    {
        cout<<"创建进程错误"<<endl;
        return ;
        //exit(1);
    }
    else if(pid==0)
    {
        createMem();
    }
    else
    {
        writeMem();
        wait(NULL);
    }
    return ;
}

```



```

void createMem()
{
    int shmID=shmget((key_t)233,512,IPC_CREAT|0600);
    if(shmID==-1)
    {
        cout<<"createMem:共享内存创建失败"<<endl;
        exit(1);
    }
    sleep(1);
    char *shmADD=(char*)shmat(shmID,NULL,0);
    char *p;
    cout<<shmADD<<endl;
    shmdt(shmADD);
    shmctl(shmID, IPC_RMID, NULL) ;
    exit(0);
}

```

```

void writeMem()
{
    int shmID=shmget((key_t)233,512,IPC_CREAT|0600);
    if(shmID==-1)
    {
        cout<<"writeMem:共享内存创建失败"<<endl;
        exit(1);
    }
    char *shmADD=(char*)shmat(shmID,NULL,0);
    char *p=(char*)"进程 B 向共享内存中写入数据";
    strcpy( shmADD, p);
    cout<<p<<endl;
    shmdt( shmADD );
    return ;
    //exit(0);
}

```

6.5 实验结果

```
root@iZh8461uthz90jZ:~/p6# ls
obg.o  obj.o  p6  p6.cpp
root@iZh8461uthz90jZ:~/p6# ./p6
请选择通信方式:
1.通过管道
2.通过消息队列
3.通过共享内存
0.退出
请输入: 1
219846 is sending a message to parent
请输入: 2
219849
当前pid为: 219849 SERVER IS WORKING!
0
当前pid为: 0 CLIENT IS WORKING!
CLIENT IS WORKING
SERVER IS WORKING
serving for client: 219849
请输入: receive reply from: 219846
3
进程B向共享内存中写入数据
进程B向共享内存中写入数据
请输入: 0
root@iZh8461uthz90jZ:~/p6#
```

6.6 总结

通过第六题，我了解了操作系统进程通信的原理，以及一些与操作系统进程通信有关的基本函数的操作，在对操作系统进程通信的学习中，我一度对管道，或者消息队列中没有消息该如何处理进程的情况产生了疑惑，终于在进行大量既有文献的查阅后理解了其中的原理，这使得我对进程通信的过程更加熟悉。

七、参考文献

- [1] 陆静, 胡明庆. 几种进程通信方法的研究和比较[J]. 福建电脑, 2007 (2): 61-61.
- [2] 徐江峰, 张战辉, 杨有. 基于 VC++ 的进程通信技术研究[J]. 计算机科学, 2007, 34(9): 262-264.
- [3] <https://www.cnblogs.com/52php/p/5862114.html>
- [4] <https://blog.csdn.net/zjuxsl/article/details/44338059>

- [5] https://blog.csdn.net/Aspiration_H/article/details/108096302
- [6] https://blog.csdn.net/aspiration_h/article/details/108096302
- [7] <https://zhuanlan.zhihu.com/p/58489873>
- [8] <https://www.cnblogs.com/sunsky303/p/9214223.html>
- [9] https://blog.csdn.net/m0_37962600/article/details/81448553
- [10] https://blog.csdn.net/Jaster_wisdom/article/details/52345674
- [11] <https://zhuanlan.zhihu.com/p/142332798>
- [12] <https://www.cnblogs.com/leesf456/p/5413517.html>