南昌大学实验报告

学生姓名：丁俊　　学　　号：8003119100　　专业班级：信安 193 班

实验类型：☐ 验证 ☐ 综合 ☐ 设计 ☐ 创新　实验日期：4.29　实验成绩：

一、 实验项目名称

进程调度

二、 实验目的

1、理解进程调度的过程。

2、掌握各种进程调度算法的实现方法

3、通过实验比较各种进程调度算法的优劣。 进程调度算法是系统管理进程调度，提高系统性能的重要手段。通过本次实验理解进程 调度的机制，在模拟实现先来先服务 FCFS、轮转 RR（q=1）、最短进程优先 SPN、最短剩 余时间 SRT、最高响应比优先 HRRN 算法的基础上，比较各种进程调度算法的效率和优劣， 从而了解系统进程调度的实现过程。

三、 实验基本原理

1、 **FCFS** 先来先服务也可以称为是 FIFO 先进先出。此策略是当前正在运行的进程停止 执行时，选择在就绪队列中存在时间最长的进程运行。这种策略执行长进程比执行短 进程更好。

2、 **轮转** 这种策略是以一个周期性间隔产生时钟中断，当中断发生时，当前正在运行的 进程被置于就绪队列中，然后基于 FCFS 策略选择下一个就绪作业运行，目的是为了 减少在 FCFS 策略下短作业的不利情况。

3、 **SPN 最短进程优先** 这种策略是下一次选择所需处理时间最短的进程。是非抢占策略，目的也是为减少 FCFS 策略对长进程的偏向。

4、 **SRT 最短剩余时间** 这种策略下调度器总是选择预期剩余时间最短的进程。是抢占策略。

5、 **HRRN 最高响应比优先** 当当前进程完成或被阻塞时，选择响应比 R 最大的就绪进程， R=(w+s)/s 其中 w: 等待处理器的时间，s:期待的服务时间。这样长进程被饿死的可能性下降。

四、 主要仪器设备及耗材

PC+linux 系统

GCC

五、 实验步骤

程序代码：

```
#include <stdio.h>
#include <iostream>
#include <cmath>
#include <queue>
#include <iomanip>
#include <cstring>
using namespace std;

queue<int> process; // 进程就绪队列从 0-4

void FCFS(float Atime[], float Stime[]) {
    float Ctime[5]; // 完成时间
    float ZZtime[5]; // 周转时间
    float DQtime[5]; // 响应比
    int i;
    Ctime[0] = Stime[0];
    printf("完成结束时间\t");
    for (i = 1; i < 5; i++)
        Ctime[i] = Stime[i] + Ctime[i - 1]; // 完成时间==服务时间+上一个进程完成时间
    for (i = 0; i < 5; i++)
        printf("%5.2f  ", Ctime[i]);
    printf("\n");
    printf("等待+服务时间\t");
    for (i = 0; i < 5; i++) {
        ZZtime[i] = Ctime[i] - Atime[i]; // 周转时间=完成时间-到达时间
        printf("%5.2f  ", ZZtime[i]);
    }
    printf("\n");
    printf("响应比\t\t");
    for (i = 0; i < 5; i++) {
        DQtime[i] = ZZtime[i] / Stime[i]; // 响应比
        printf("%5.2f  ", DQtime[i]);
    }
    printf("\n");
}

void SJF(float Atime[], float Stime[]) {
    float Ctime[5];
```

```
    float ZZtime[5];
    float DQtime[5];
    int i, j;
    // 开始时刻只有一个进程执行
    Ctime[0] = Stime[0];
    ZZtime[0] = Ctime[0] - Atime[0];
    DQtime[0] = ZZtime[0] / Stime[0];
    for (i = 1; i < 5; i++) {
        for (int j = i; j < 5; j++) { // 排序
            for (int d = i + 1; d < 5; d++) {
                if (Ctime[i - 1] >= Atime[i] && Ctime[i - 1] >= Atime[d]
    && Stime[j] > Stime[d]) {
                    int m, n;
                    m = Stime[j];
                    Stime[j] = Stime[d];
                    Stime[d] = m;
                    n = Atime[j];
                    Atime[j] = Atime[d];
                    Atime[d] = n;
                }
            }
        }
        if (Atime[i] < Ctime[i - 1]) { // 当前到达时间在上一个作业结束之
    前
            Ctime[i] = Ctime[i - 1] + Stime[i];
            ZZtime[i] = Ctime[i] - Atime[i];
            DQtime[i] = ZZtime[i] / Stime[i];
        } else { // 当前到达时间在上一个作业结束时间之后
            Ctime[i] = Atime[i] + Stime[i];
            ZZtime[i] = Ctime[i] - Atime[i];
            DQtime[i] = ZZtime[i] / Stime[i];
        }
    }
    printf("完成时间\t");
    for (int i = 0; i < 5; i++)
        printf("%5.2f  ", Ctime[i]);
    printf("\n");
    printf("周转时间\t");
    for (int i = 0; i < 5; i++)
        printf("%5.2f  ", ZZtime[i]);
    printf("\n");
    printf("响应比\t");
    for (int i = 0; i < 5; i++)
        printf("%5.2f  ", DQtime[i]);
```

```
81.
82.}
83.
84.// 时间片轮转算法
85.void RR(float Atime[], float Stime[]) {
86.    float atime[5], stime[5]; // 因为下面操作时数据会改变拷贝一下
87.    float Ctime[5];
88.    float ZZtime[5];
89.    float DQtime[5];
90.    for (int k = 0; k < 5; k++) {
91.        atime[k] = Atime[k];
92.        stime[k] = Stime[k];
93.    }
94.    int q;
95.    scanf("%d", &q);
96.    int point = 0; // 从 0 开始记录每段时间片的下标
97.    process.push(0); // 把第一个进程放进队列
98.    int processMoment[100]; // 存储每个时间片 p 对应的进程编号
99.    int Currenttime = 0;
100.     int i = 1; // 指向还未处理的进程下标
101.     int temptime;
102.     int finalProcessNumber = 0; //执行 RR 算法后，进程的个数
103.     int processTime[50]; // 每个时间片执行的时间
104.     //声明此变量控制 CurrentTime 的累加时间，当前进程的服务时间小于时间片 q 的
    时候，起到重要作用
105.     // Currenttime 初始化
106.     if (Stime[0] >= q) {
107.         Currenttime = q;
108.     } else {
109.         Currenttime = Stime[0];
110.     }
111.
112.     while (!process.empty()) { // 当进程队列不空
113.         for (int j = i; j < 5; j++) { //使得满足进程的到达时间小于当前时
    间的进程都进入队列
114.             if (Currenttime >= Atime[i]) {
115.                 process.push(i); // 进程编号
116.                 i++;
117.             }
118.         }
119.         if (Stime[process.front()] < q) {
120.             temptime = Stime[process.front()];
121.         } else {
122.             temptime = q;
```

```
123.         }
124.         Stime[process.front()] -= q;  //进程每执行一次，就将其服务时
   间 -q
125.
126.         processMoment[point] = process.front();
127.         point++;
128.         processTime[finalProcessNumber] = temptime;
129.         finalProcessNumber++;
130.
131.         if (Stime[process.front()] <= 0) {
132.             process.pop();
133.         } else {
134.             process.push(process.front());
135.             process.pop();
136.         }
137.         Currenttime += temptime; // 加上单个时间片的长度
138.     }
139.     printf("各进程的执行时刻信息:\n");
140.     printf(" 0 时刻---->%d 时刻", processTime[0]);
141.     processTime[finalProcessNumber] = 0;
142.     int time = processTime[0];
143.     int count = 0;
144.     for (int i = 0; i < finalProcessNumber; i++) {
145.         count = 0;
146.         cout << setw(3) << processMoment[i] << setw(3) << endl; // 输出
   进程编号
147.
148.         while (count != processMoment[i] && count < 5) {
149.             count++;
150.         }
151.         Ctime[count] = time; // 完成时间
152.         cout << "标记:" << Ctime[count] << endl;
153.         if (i < finalProcessNumber - 1) {
154.             cout << setw(3) << time << "时刻
   " << "-->" << setw(2) << time + processTime[i + 1] << "时刻
   " << setw(3);
155.             time += processTime[i + 1];
156.         }
157.     }
158.     printf("\n");
159.     // 打印计算
160.     for (int i = 0; i < 5; i++) {
161.         ZZtime[i] = Ctime[i] - atime[i]; // 周转时间
162.     }
```

```
    for (int i = 0; i < 5; i++) {
        DQtime[i] = ZZtime[i] / stime[i];
    }
    printf("完成时间:\t");
    for (int j = 0; j < 5; j++) {
        printf("%5.2f  ", Ctime[j]);
    }
    printf("\n");
    printf("周转时间:\t");
    for (int j = 0; j < 5; j++) {
        printf("%5.2f  ", ZZtime[j]);
    }
    printf("\n");
    printf("响应比:\t");
    for (int j = 0; j < 5; j++) {
        printf("%5.2f ", DQtime[j]);
    }
}

// 最高响应比优先算法
void HRN(float Atime[], float Stime[]) {
    float Ctime[5];
    float ZZtime[5];
    float DQtime[5];
    int t = 1;
    float m, n, k, f;
    Ctime[0] = Stime[0];
    ZZtime[0] = Ctime[0] - Atime[0];
    DQtime[0] = ZZtime[0] / Stime[0];
    float  maxvalue;
    int next; // 下一个要执行的进程
    for (int i = 1; i < 5; i++) {
        maxvalue = 1 + (Ctime[i - 1] - Atime[i]) / Stime[i]; // 响应比
        next = i; // 现在在第几个进程
        for (int j = i; j < 5; j++) {
            if ((Ctime[i - 1] - Atime[j]) / Stime[j] + 1 > maxvalue && Atime[i] <= Ctime[i - 1]) { // 在第一个进程完成结束前到达
                next = j; // 找出最大的响应比
            }
        }
        m = Atime[i], Atime[i] = Atime[next], Atime[next] = m;
        n = Stime[i], Stime[i] = Stime[next],   Stime[next] = n; // 把最大响应比的进程移动到前面
```

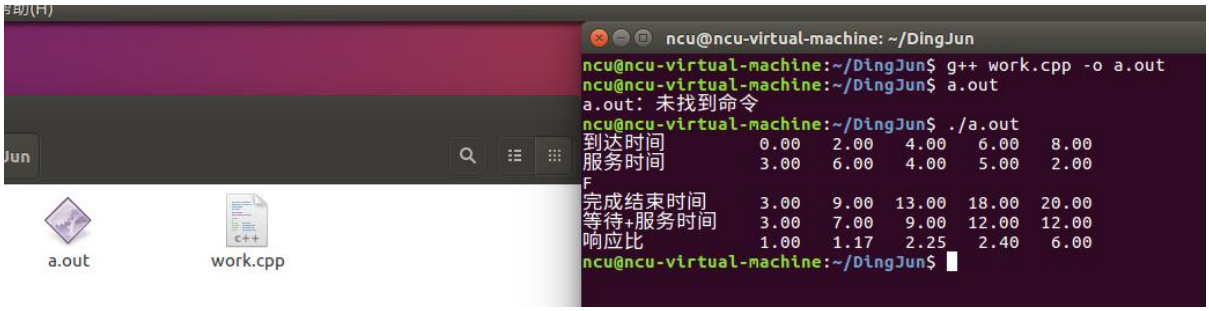
```
        Ctime[next] = Ctime[i - 1] + Stime[next]; // 计算完成时间
    }
    for (int i = 0; i < 5; i++)
        printf("%5.2f ", Atime[i]);
    printf("\n");
    for (int i = 0; i < 5; i++)
        printf("%5.2f ", Stime[i]);
    printf("\n");
    for (int i = 1; i < 5; i++) {
        Ctime[i] = Stime[i] + Ctime[i - 1];
    }
    printf("完成时间\t");
    for (int i = 0; i < 5; i++)
        printf("%5.2f ", Ctime[i]);
    printf("\n");
    printf("周转时间\t");
    for (int i = 1; i < 5; i++)
        ZZtime[i] = Ctime[i] - Atime[i];
    for (int i = 0; i < 5; i++)
        printf("%5.2f ", ZZtime[i]);
    printf("\n");
    printf("响应比\t");
    for (int i = 1; i < 5; i++)
        DQtime[i] = ZZtime[i] / Stime[i];
    for (int i = 0; i < 5; i++)
        printf("%5.2f ", DQtime[i]);
    printf("\n");
}

int main() {
    char c;
    int i;
    float Atime[5] = {0, 2, 4, 6, 8};
    float Stime[5] = {3, 6, 4, 5, 2};
    printf("到达时间    ");
    for (i = 0; i < 5; i++)
        printf("%5.2f  ", Atime[i]);
    printf("\n");
    printf("服务时间    ");
    for (i = 0; i < 5; i++)
        printf("%5.2f  ", Stime[i]);
    printf("\n");
    scanf("%c", &c);
    switch (c) {
```

```
248.         case 'F':
249.             FCFS(Atime, Stime);
250.             break;
251.         case 'S':
252.             SJF(Atime, Stime);
253.             break;
254.         case 'R':
255.             RR(Atime, Stime);
256.             break;
257.         case 'H':
258.             HRN(Atime, Stime);
259.             break;
260.         default:
261.             printf("error");
262.     }
263.
264. }
```

运行结果：

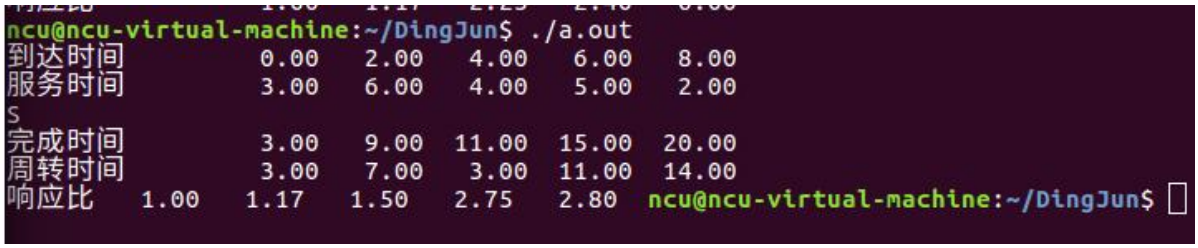
FCFS 先来先服务算法



分析：先来先服务算法是按照进程到来时的时刻顺序调度的，设五个进程序号分别为ABCDE，则调度顺序为 A->B->C->D->E,那么有下表格：
周转时间=完成时间-到达时间；
响应比=(等待时间+服务时间)/服务时间=周转时间/服务时间

进程	到达时间	服务时间	完成时间	周转时间	响应比
A	0.00	3.00	3.00	3.00	1
B	2.00	6.00	9.00	7.00	1.17
C	4.00	4.00	13.00	9.00	2.25
D	6.00	5.00	18.00	12.00	2.40
E	8.00	2.00	20.00	12.00	6.00

SJF 最短进程优先算法



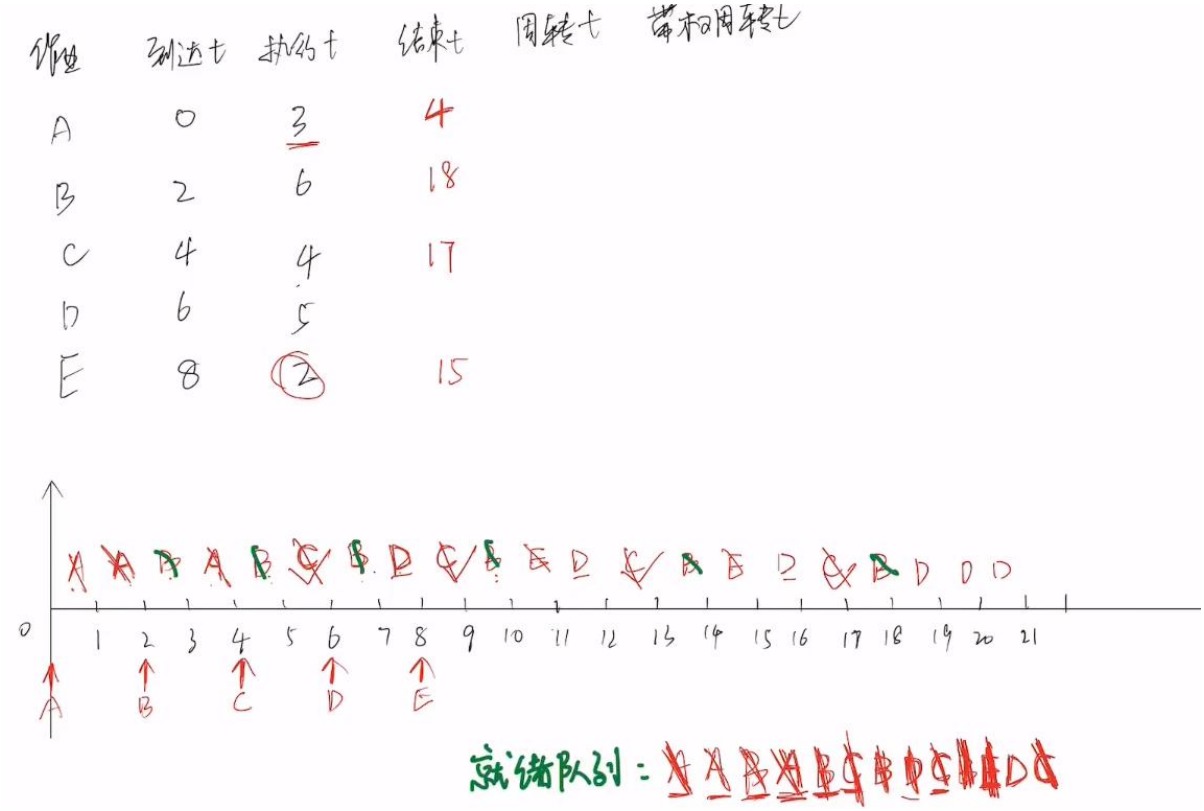
分析：SJF 最短进程优先算法是优先调度短作业的一种算法，将每个进程与其估计运行时间进行关联选取估计计算时间最短的作业投入运行。开始 0 时刻只有一个 A 进程执行。3 时刻 A 进程执行完成，在此之前 B 进程进入进程队列，此时也只有它一个，等到 9 时刻 B 也执行完毕。在此之前 C、D、E 进程已经进入进程队列，这时比较三个进程中服务时间最短的进程即 E 最短，然后是 C、D。在程序中从第 2 个进程开始遍历，当前时刻为 A 进程执行完成的时刻，每次判断在当前时刻是否有新进程加入且找出服务时间最短的进程并将它提前交换。

上面程序的完成时间不是指的具体进程，而是每段进程执行完的时间，所以要看间隔则执行顺序为 A->B->E->C->D.

如下表：

进程	到达时间	服务时间	完成时间	周转时间	响应比
A	0.00	3.00	3.00	3.00	1.00
B	2.00	6.00	9.00	7.00	1.17
C	4.00	4.00	15.00	11.00	2.75
D	6.00	5.00	20.00	14.00	2.80
E	8.00	2.00	11.00	3.00	1.50

RR 时间片轮转图：

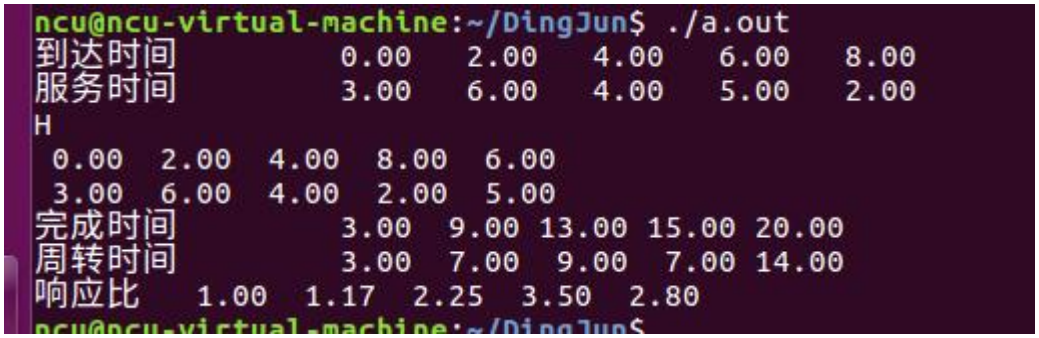


时间片轮转算法我才用了更加详细的做法，用一个队列存储就绪的进程数量，记录每一个时间片上的具体进程，0~4 分别代表 5 个进程的编号，有可能某个进程的服务时间<时间片长度，此时该段时间就会变为这个进程的服务时间长度，用 processTime 数组存储每段时间的长度。当队列不为空时持续进行轮转执行，每次取出队首的进程并删除，执行完成后判断该时刻上是否有新进程进入，有则加入；在次之后，再判断执行的当前进程是否全部完成，若没有则添加到就绪队列尾部，依次进行。

```
ncu@ncu-virtual-machine:~/DingJun$ ./a.out
到达时间          0.00   2.00   4.00   6.00   8.00
服务时间          3.00   6.00   4.00   5.00   2.00
R
1
各进程的执行时刻信息:
 0时刻---->1时刻  0
标记:1
  1时刻--> 2时刻  0
标记:2
  2时刻--> 3时刻  1
标记:3
  3时刻--> 4时刻  0
标记:4
  4时刻--> 5时刻  1
标记:5
  5时刻--> 6时刻  2
标记:6
  6时刻--> 7时刻  1
标记:7
  7时刻--> 8时刻  3
标记:8
  8时刻--> 9时刻  2
标记:9
  9时刻-->10时刻  1
标记:10
 10时刻-->11时刻  4
标记:11
 11时刻-->12时刻  3
标记:12
 12时刻-->13时刻  2
标记:13
 13时刻-->14时刻  1
标记:14
 14时刻-->15时刻  4
标记:15
 15时刻-->16时刻  3
标记:16
 16时刻-->17时刻  2
标记:17
 17时刻-->18时刻  1
标记:18
 18时刻-->19时刻  3
标记:19
 19时刻-->20时刻  3
标记:20

完成时间:         4.00  18.00  17.00  20.00  15.00
周转时间:         4.00  16.00  13.00  14.00   7.00
响应比:  1.33  2.67  3.25  2.80  3.50 ncu@ncu-virt
```

HRN 最高响应比优先算法：



由于 HRN 不是抢占算法，由完成时间可以看出顺序为 A->B->C->E->D,开始时只执行 A 一个进程，执行完后比较其余进程的响应比。

R(B)=1+(3-2)/6=1.17;只有 B 一个进程，执行 B；

R(C)=1+(9-4)/4=2.25;R(D)=1+(9-6)/5=1.6;R(E)=1+(9-8)/2=1.5;执行 C；

R(D)=1+(13-6)/5=2.4;R(E)=1+(13-8)/2=3.5;执行 E

进程	到达时间	服务时间	完成时间	周转时间	响应比
A	0.00	3.00	3.00	3.00	1.00
B	2.00	6.00	9.00	7.00	1.17
C	4.00	4.00	13.00	9.00	2.25
D	6.00	5.00	20.00	14.00	2.80
E	8.00	2.00	15.00	7.00	3.50

六、 思考讨论题或体会或对改进实验的建议

我们可以发现

先来先服务算法特点：算法简单，可以实现基本上的公平。

短作业优先算法：

(1)平均周转时间、平均带权周转时间都有明显改善。SJF/SPF 调度算法能有效的降低作业的平均等待时间，提高系统吞吐量。

(2）未考虑作业的紧迫程度，因而不能保证紧迫性作业（进程）的及时处理、对长作业的不利、作业（进程）的长短含主观因素，不一定能真正做到短作业优先。

高响应比优先调度算法

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

RR 时间片轮转法

1、进程阻塞情况发生时，未用完时间片也要出让 CPU。2、能够及时响应，但没有考虑作业长短等问题。3、系统的处理能力和系统的负载状态影响时间片长度。

七、 参考资料