

**信息科学技术学院、人工智能学院**

**课程（实习）设计**

**课程设计名称：** 操作系统实习

**专 业：** 计算机科学与技术

**学 号：** 2351610105

**学 生 姓 名：** 方泽宇

**成 绩：**

**批 改 日 期：**

**教 师 签 名：**

目 录

题目：进程调度 1

1.1 实验内容 1

1.2算法描述 1

1.3 实验结果 1

1.4 实现小结 3

1.5实验代码 3

题目：进程的同步与互斥 20

1.1 实验内容 20

1.2算法描述 20

1.3 实验结果 20

1.4 实现小结 20

1.5实验代码 20

题目：存储管理 21

1.1 实验内容 21

1.2算法描述 21

1.3 实验结果 21

1.4 实现小结 21

1.5实验代码 21

# 题目：进程调度

## 1.1 实验内容

* 实验目的：加深对进程调度的理解，熟悉进程调度的不同算法，比较其优劣性。
* 实验内容：假如一个系统中有5个进程，它们的到达时间内如表1所示，忽略I/O以及其他开销时间。若分别按抢占的短作业优先（SJF）、时间片轮转（RR，时间片=1）进行CPU调度，请按照上述2个算法，编程计算出各进程的完成时间内、周转时间、带权周转周期、平均周转周期和平均带权周转时间。

表1 进程到达和需服务时间

|  |  |  |
| --- | --- | --- |
| 进程 | 到达时间 | 服务时间 |
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

## 1.2算法描述

* 算法1:抢占的短作业优先(SJF)

SJF算法是以作业的长短来计算优先级，作业越短，其优先级越高。作业的长短是以作业所要求的运行时间来衡量的。SJF算法可以分别用于作业调度和进程调度。在把段作业优先调度算法用于作业调度时，它将从外存的作业后备队列中选择若干个估计运行时间最短的作业，优先将他们调入内存中运行。

SJF可以有多种写法，一种是采取新进入的进程优先的写法，另一种是上一次运行的进程优先。我认为新进入的进程优先的写法更符合程序运行的规则，所以本次实习中我采用新进入的进程优先的办法来写。

* 算法2:时间便流转(RR，时间片=1)

在轮转(RR)法中，系统根据FCFS策略，将所有的就绪进程排列成一个就绪队列，并可设置每隔一定时间间隔(比如30ms)完成一次中断，激活系统中的进程调度程序，完成一次调度，将CPU分配给队首进程，令其执行。当该新进程的时间片耗尽或运行完毕时，系统再次将CPU分配给心得队首进程(或者新到达的紧迫进程)。由此，可保证就绪队列中的所有进程在一个确定的时间段内，都能获得一次CPU执行。

## 1.3 实验结果

实验需要在运行程序相同目录下放置文件：filename.dat，文件内容如下：

5

A 0 3

B 2 6

C 4 4

D 6 5

E 8 2

第一种SJF抢占的短作业优先调度算法，新进入的进程的优先级别更高，存在如下执行日志：

第0-1秒，执行程序A,剩余2/3

第1-2秒，执行程序A,剩余1/3

第2-3秒，执行程序A,剩余0/3

第3-4秒，执行程序B,剩余5/6

第4-5秒，执行程序C,剩余3/4

第5-6秒，执行程序C,剩余2/4

第6-7秒，执行程序C,剩余1/4

第7-8秒，执行程序C,剩余0/4

第8-9秒，执行程序E,剩余1/2

第9-10秒，执行程序E,剩余0/2

第10-11秒，执行程序D,剩余4/5

第11-12秒，执行程序D,剩余3/5

第12-13秒，执行程序D,剩余2/5

第13-14秒，执行程序D,剩余1/5

第14-15秒，执行程序D,剩余0/5

第15-16秒，执行程序B,剩余4/6

第16-17秒，执行程序B,剩余3/6

第17-18秒，执行程序B,剩余2/6

第18-19秒，执行程序B,剩余1/6

第19-20秒，执行程序B,剩余0/6

对于第二种RR时间片轮转，采用新进入进程优先的模式，存在以下运行日志：

第0-1秒，执行程序A,剩余2/3

第1-2秒，执行程序A,剩余1/3

第2-3秒，执行程序B,剩余5/6

第3-4秒，执行程序A,剩余0/3

第4-5秒，执行程序B,剩余4/6

第5-6秒，执行程序C,剩余3/4

第6-7秒，执行程序B,剩余3/6

第7-8秒，执行程序D,剩余4/5

第8-9秒，执行程序C,剩余2/4

第9-10秒，执行程序B,剩余2/6

第10-11秒，执行程序E,剩余1/2

第11-12秒，执行程序D,剩余3/5

第12-13秒，执行程序C,剩余1/4

第13-14秒，执行程序B,剩余1/6

第14-15秒，执行程序E,剩余0/2

第15-16秒，执行程序D,剩余2/5

第16-17秒，执行程序C,剩余0/4

第17-18秒，执行程序B,剩余0/6

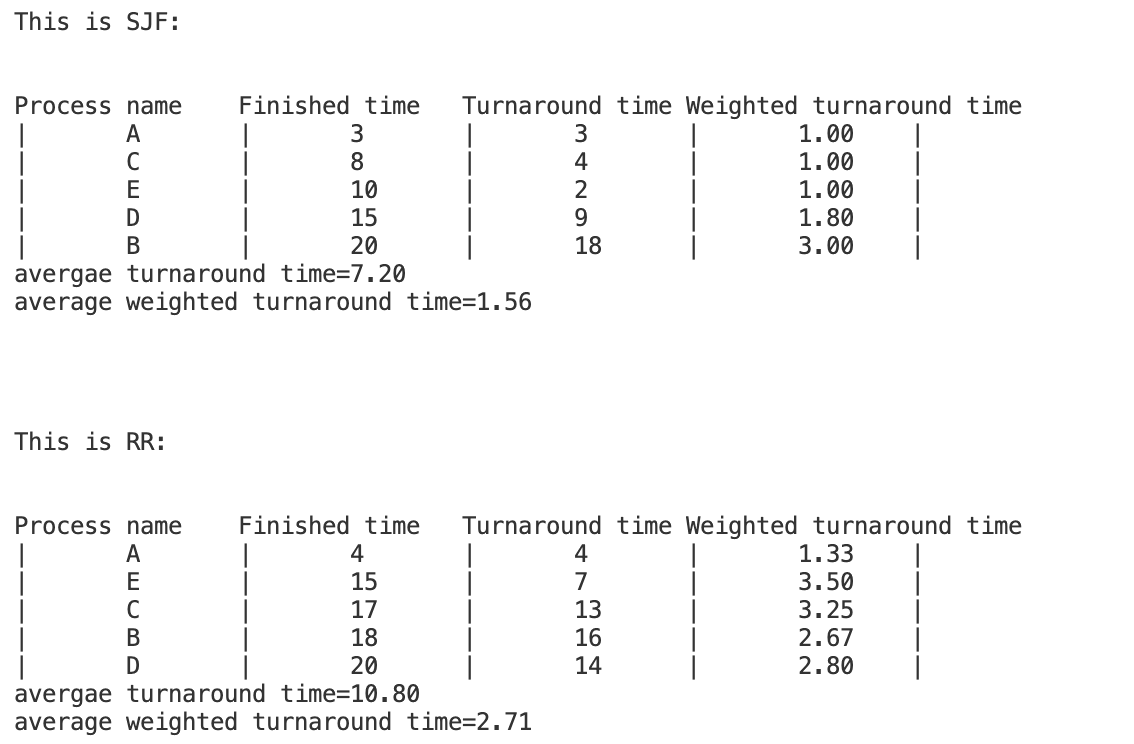
第18-19秒，执行程序D,剩余1/5

第19-20秒，执行程序D,剩余0/5

此程序在macOS14.6.1系统，采用如下编译指令：

cd "/Users/fang50253/Desktop/Files/Documents/NJFU\_Training/NJFU\_CS\_Training/ 操作系统实习/1.进程调度/" && g++ -std=c++14 01Process\_submit.cpp -o 01Process\_submit && "/Users/fang50253/Desktop/Files/Documents/NJFU\_Training/NJFU\_CS\_Training/操作系统实习/1.进程调度/"01Process\_submit

运行结果的截图如下：



## 1.4 实现小结

* 在本次实习的过程中，我原本写的答案是非抢占式的，也就是SPF(相关代码我将会打包放在文件夹下，在本实习报告中不再展现)，后来发现这样的结果和其他同学的有些许偏差，于是我重新检查自己的思路，发现代码中存在的问题，重新编程，写了SJF和RR算法。
* 由于实习过程中不可以使用C++的标准STL库，因此在本次实习中我通过C++模板类创建了queue、stack和priority\_queue三种模板类，queue用于RR，priority\_queue用于到时间以后将进程加入就绪态，以及SJF算法的实现。

## 1.5实验代码

#include<stdio.h>

#include<iostream>

#include<stdexcept>

#include<string.h>

#define MAX\_PROCESS\_NUMBER 100

namespace fzy

{

    template<class T>

    class Less

    {

    public:

        bool operator()(const T& x, const T& y)

        {

            return x < y;

        }

    };

    template<class T>

    class Greater

    {

    public:

        bool operator()(const T& x, const T& y)

        {

            return x > y;

        }

    };

    template<typename T>

    struct queuenode

    {

        T v=T();

        queuenode<T>\* next;

    };

    template<typename T>

    class queue

    {

    private:

        queuenode<T>\* head;

        queuenode<T>\* last;

        int size\_;

    public:

        queue()

        {

            head=new queuenode<T>;

            head->next=nullptr;

            last=head;

            size\_=0;

        }

        ~queue()

        {

            clear();

            delete head;

        }

        void clear()

        {

            while(!empty()) pop();

        }

        void push(T v)

        {

            queuenode<T>\* newnode = new queuenode<T>;

            newnode->v = v;

            newnode->next = nullptr;

            last->next = newnode;

            last = newnode;

            ++size\_;

        }

        void pop()

        {

            if (size\_>0)

            {

                queuenode<T>\* temp=head->next;

                head->next=temp->next;

                if (head->next==nullptr) last=head;

                delete temp;

                --size\_;

            }

        }

        T front()

        {

            if(size\_>0) return head->next->v;

            throw std::runtime\_error("Queue is empty");

        }

        T back()

        {

            if(size\_>0) return last->v;

            throw std::runtime\_error("Queue is empty");

        }

        bool empty()

        {

            return size\_==0;

        }

        int size()

        {

            return size\_;

        }

        void print()

*//定义为按照队列的入(出)队顺序进行打印*

        {

            queuenode<T> \*i;

            int j;

            for(i=head->next,j=1;j<=size\_;++j,i=i->next)

            {

                std::cout<<i->v<<" ";

            }

            std::cout<<std::endl;

        }

    };

    template<typename T>

    struct stacknode

    {

        T v;

        stacknode \*next;

    };

    template<typename T>

    class stack

    {

        private:

        stacknode<T> \*head;

        int size\_;

        public:

        stack()

        {

            head=new stacknode<T>;

            head->next = nullptr;

            size\_=0;

        }

        ~stack()

        {

            clear();

        }

        void clear()

        {

            while(size()) pop();

        }

        void pop()

        {

            if(size\_==0) throw std::runtime\_error("stack is empty");

            stacknode<T>\* tmp=head->next;

            head->next=head->next->next;

            --size\_;

            delete tmp;

        }

        int size()

        {

            return size\_;

        }

        bool empty()

        {

            return size\_==0;

        }

        T top()

        {

            if(size\_==0) throw std::runtime\_error("stack is empty");

            return head->next->v;

        }

        void push(T v)

        {

            stacknode<T> \*newnode=new stacknode<T>;

            newnode->next=head->next;

            head->next=newnode;

            newnode->v=v;

            ++size\_;

        }

        void print()

*//定义为按照栈的出栈顺序进行打印*

        {

            int j=1;

            stacknode<T> \*i;

            for(i=head->next;j<=size\_;++j,i=i->next)

            {

                std::cout<<i->v<<" ";

            }

            std::cout<<std::endl;

        }

    };

    template<typename T>

    class priority\_queue{

    public:

        priority\_queue()\

        :size\_of\_priority\_queue(0), capacity(MAX\_PROCESS\_NUMBER), compare(&\_compare)

        {

            pt = new T[capacity];

            if(nullptr==pt) throw std::runtime\_error("malloc failed");

        }

        priority\_queue(int val)\

            :size\_of\_priority\_queue(0), capacity(MAX\_PROCESS\_NUMBER), compare(&\_compare){

            while( capacity < val) capacity \*= 2;

            pt = new T[capacity];*//申请空间*

            if( nullptr == pt) throw std::runtime\_error("malloc failed");

            return;

        }

        priority\_queue(bool (\*cmp)(T&,T&))\

            :size\_of\_priority\_queue(0), capacity(MAX\_PROCESS\_NUMBER), compare(cmp)

            {

            pt = new T[capacity];

            if( nullptr == pt )throw std::runtime\_error("malloc failed");

            return;

        }

        priority\_queue( int val, bool (\*cmp)(T&,T&) )\

            :size\_of\_priority\_queue(0), capacity(MAX\_PROCESS\_NUMBER), compare(cmp){

            while( capacity < val) capacity \*= 2;

            pt = new T[capacity];

            if( nullptr == pt ) throw std::runtime\_error("malloc failed");

            return;

        }

        ~priority\_queue()

        {

            if( nullptr != pt){

                delete[] pt;

                pt = nullptr;

            }

        }

        bool empty()

        {

            return size\_of\_priority\_queue==0;

        }

        bool push(const T& t)

        {

            T \*ptt = pt;

            if( size\_of\_priority\_queue == capacity )

            {

                capacity \*= 2;

                pt = new T[capacity];

                if( nullptr == pt )

                {

                    pt = ptt;

                    capacity /= 2;

                    return false;

                }

                obj\_cpy(pt, ptt, size\_of\_priority\_queue);

                delete[] ptt;

            }

            pt[size\_of\_priority\_queue++] = t;

            heap\_up();

            return true;

        }

        bool pop(){

            if(size\_of\_priority\_queue==0) return false;

            if(size\_of\_priority\_queue==1)

            {

                size\_of\_priority\_queue = 0;

                return true;

            }

            pt[0] = pt[size\_of\_priority\_queue-1];

            size\_of\_priority\_queue--;

            heap\_down();

            return true;

        }

        T top()

        {

            if(size\_of\_priority\_queue<0) throw std::runtime\_error("queue empty");

            return pt[0];*//返回队头元素*

        }

        bool is\_empty\_pl()const

        {

            return 0==size\_of\_priority\_queue;*//返回队是否为空*

        }

        int get\_size()const

        {

            return size\_of\_priority\_queue;*//返回队元素个数*

        }

        int get\_capacity()const

        {

            return capacity;*//返回队当前容量应该为2的n次方*

        }

    private:

        void heap\_up();

        void heap\_down();

        void obj\_cpy(T\* dest, const T\* sour, int n)

        {

            for(int i=0;i<n;i++) dest[i]=sour[i];

        }

        bool static \_compare(T &t1, T &t2)

        {

            return t1 < t2;

        }

    private:

        T     \*pt;*//数据*

        int    size\_of\_priority\_queue;*// 元素个数*

        int    capacity;*//队容量*

        bool (\*compare)(T&,T&);*//比较函数*

    };

    template<typename T>

    void priority\_queue<T>::heap\_up()

    {

        T temp;

        int itr = size\_of\_priority\_queue-1;

        while( itr > 0 ){

            if( (compare(pt[itr/2], pt[itr])) )

            {

                temp = pt[itr];

                pt[itr] = pt[itr/2];

                pt[itr/2] = temp;

                itr = itr/2;

                continue;

            }

            break;

        }

        return;

    }

    template<typename T>

    void priority\_queue<T>::heap\_down(){

*//当对头出栈韩，需要将队尾数据移动到队头，向下重新调整堆*

        T temp;

        int pitr = 0, citr;

        while( pitr <= size\_of\_priority\_queue/2 -1 )

        {

            citr = pitr \* 2 + 1;

            if( citr + 1 < size\_of\_priority\_queue && compare(pt[citr], pt[citr+1]))

                citr ++;

            if( (compare(pt[pitr], pt[citr])) )

            {

                temp = pt[citr];

                pt[citr] = pt[pitr];

                pt[pitr] = temp;

                pitr = citr;*//继续将pitr指向孩子节点，进行下一次的比较*

                continue;

            }

            break;*//如果处在对的位置，直接结束，不需要继续比较下去了*

        }

        return;

    }

}

namespace os

{

    class Process

    {

        public:

        char process\_name;*//进程名称*

        int time\_arrive;*//到达时间*

        int time\_serve;*//服务时间*

        bool operator < (const Process &W) const

        {

            return time\_arrive>W.time\_arrive;

        }

        bool operator <= (const Process &W) const

        {

            return time\_arrive>=W.time\_arrive;

        }

        bool operator > (const Process &W) const

        {

            return time\_arrive<W.time\_arrive;

        }

        bool operator >= (const Process &W) const

        {

            return time\_arrive<=W.time\_arrive;

        }

        Process(char process\_name,int time\_arrive,int time\_serve)

        {

            this->process\_name=process\_name;

            this->time\_arrive=time\_arrive;

            this->time\_serve=time\_serve;

        }

        Process()

        {

            this->process\_name=0;

            this->time\_arrive=0;

            this->time\_serve=0;

        }

    };

    class Process\_finish:public Process

    {

        public:

        Process\_finish(Process a,int time\_end,int time\_turnaround,double time\_turnaround\_rights)

        {

            this->process\_name=a.process\_name;

            this->time\_arrive=a.time\_arrive;

            this->time\_serve=a.time\_serve;

            this->time\_end=time\_end;

            this->time\_turnaround=time\_turnaround;

            this->time\_turnaround\_rights=time\_turnaround\_rights;

        }

        Process\_finish()

        {

            this->process\_name=0;

            this->time\_arrive=0;

            this->time\_end=0;

            this->time\_serve=0;

            this->time\_turnaround=0;

            this->time\_turnaround\_rights=0;

        }

        int time\_end;*//完成时间*

        int time\_turnaround;*//周转时间*

        double time\_turnaround\_rights;*//带权周转时间*

        bool operator < (const Process &W) const

        {

            return time\_serve>W.time\_serve;

        }

        bool operator <= (const Process &W) const

        {

            return time\_serve>=W.time\_serve;

        }

        bool operator > (const Process &W) const

        {

            return time\_serve<W.time\_serve;

        }

        bool operator >= (const Process &W) const

        {

            return time\_serve<=W.time\_serve;

        }

    }finished[MAX\_PROCESS\_NUMBER];

    int finished\_index;

    class Process\_remain:public Process

    {

        public:

        int remain;

        Process\_remain(class Process p)

        {

            this->process\_name=p.process\_name;

            this->time\_arrive=p.time\_arrive;

            this->time\_serve=p.time\_serve;

            this->remain=p.time\_serve;

        }

        Process\_remain()

        {

            this->remain=0;

        }

        bool operator < (const Process &W) const

        {

            return time\_serve>W.time\_serve;

        }

        bool operator <= (const Process &W) const

        {

            return time\_serve>=W.time\_serve;

        }

        bool operator > (const Process &W) const

        {

            return time\_serve<W.time\_serve;

        }

        bool operator >= (const Process &W) const

        {

            return time\_serve<=W.time\_serve;

        }

    };

    class RR

    {

        private:

        fzy::priority\_queue<Process>process;

        fzy::queue<Process\_remain>doing\_process;

        int process\_cnt;

        int time;

        public:

        void read(char filename[])

        {

            FILE \*fp=fopen(filename,"r+");

            if(fp==NULL) throw std::runtime\_error("open file failed");

            fscanf(fp,"%d",&process\_cnt);

            for(int i=1;i<=process\_cnt;++i)

            {

                char process\_name;

                int time\_arrive,time\_serve;

                fscanf(fp," %c%d%d",&process\_name,&time\_arrive,&time\_serve);

                process.push(Process(process\_name,time\_arrive,time\_serve));

            }

            fclose(fp);

        }

        RR()

        {

            process\_cnt=0;

            finished\_index=0;

            char filename[]="filename.dat";

            read(filename);

            memset(finished,0,sizeof finished);

            finished\_index=0;

        }

        void conduct()

        {

            time=0;

            Process\_remain doing;

            while(finished\_index!=process\_cnt)

            {

                Process top\_process;

                if(!process.empty())

                {

                    top\_process=process.top();

                    if(top\_process.time\_arrive<=time)

                    {

                        process.pop();

                        doing\_process.push(Process\_remain(top\_process));

*//printf("push:%c\n",top\_process.process\_name);*

                    }

                }

                doing=doing\_process.front();

                doing\_process.pop();

                --doing.remain;

*//printf("%c",doing.process\_name);*

                ++time;

                if(!process.empty())

                {

                    top\_process=process.top();

                    while(top\_process.time\_arrive<=time)

                    {

                        process.pop();

                        doing\_process.push(Process\_remain(top\_process));

                        top\_process=process.top();

                        if(process.empty()) break;

                    }

                }

                if(doing.remain==0)

                {

                    ++finished\_index;

                    finished[finished\_index].process\_name=doing.process\_name;

                    finished[finished\_index].time\_arrive=doing.time\_arrive;

                    finished[finished\_index].time\_end=time;

                    finished[finished\_index].time\_serve=doing.time\_serve;

                    finished[finished\_index].time\_turnaround=time-doing.time\_arrive;

                    finished[finished\_index].time\_turnaround\_rights=1.0\*finished[finished\_index].time\_turnaround/doing.time\_serve;

                }

                else doing\_process.push(doing);

            }

        }

        void display()

        {

            printf("Process name\t");

            printf("Finished time\t");

            printf("Turnaround time\t");

            printf("Weighted turnaround time\t\n");

            for(int i=1;i<=finished\_index;++i)

            {

                printf("|\t%c\t|\t",finished[i].process\_name);*//输出进程名称*

                printf("%d\t|\t",finished[i].time\_end);*//输出进程完成时间*

                printf("%d\t|\t",finished[i].time\_turnaround);*//周转时间*

                printf("%.2lf\t|\n",finished[i].time\_turnaround\_rights);*//带权周转时间*

            }

        }

        void display\_avergae()

        {

            double average\_time\_turnaround=0;

            double average\_time\_turnaround\_rights=0;

            for(int i=1;i<=finished\_index;++i)

            {

                average\_time\_turnaround+=finished[i].time\_turnaround;

                average\_time\_turnaround\_rights+=finished[i].time\_turnaround\_rights;

            }

            average\_time\_turnaround/=finished\_index;

            average\_time\_turnaround\_rights/=finished\_index;

            printf("avergae turnaround time=%.2lf\n",average\_time\_turnaround);

            printf("average weighted turnaround time=%.2lf",average\_time\_turnaround\_rights);

        }

    };

    class SJF

    {

        private:

        public:

        void read(char filename[])

        {

            FILE \*fp=fopen(filename,"r+");

            if(fp==NULL) throw std::runtime\_error("open file failed");

            fscanf(fp,"%d",&process\_cnt);

            for(int i=1;i<=process\_cnt;++i)

            {

                char process\_name;

                int time\_arrive,time\_serve;

                fscanf(fp," %c%d%d",&process\_name,&time\_arrive,&time\_serve);

                process.push(Process(process\_name,time\_arrive,time\_serve));

            }

            fclose(fp);

        }

        int process\_cnt;

        fzy::priority\_queue<Process>process;

        fzy::priority\_queue<Process\_remain>doing\_process;

        public:

        SJF()

        {

            process\_cnt=0;

            finished\_index=0;

            char filename[]="filename.dat";

            read(filename);

            memset(finished,0,sizeof finished);

            finished\_index=0;

        }

        void conduct()

        {

            int time=0;

            Process\_remain doing;

            while(!(process.empty()&&doing\_process.empty()))

            {

                Process top\_process;

                if(!process.empty())

                {

                    top\_process=process.top();

                    while(top\_process.time\_arrive<=time)

                    {

                        process.pop();

                        doing\_process.push(Process\_remain(top\_process));

                        top\_process=process.top();

                        if(process.empty()) break;

                    }

                }

                doing=doing\_process.top();

                doing\_process.pop();

                --doing.remain;

                ++time;

                if(doing.remain==0)

                {

                    ++finished\_index;

                    finished[finished\_index].process\_name=doing.process\_name;

                    finished[finished\_index].time\_arrive=doing.time\_arrive;

                    finished[finished\_index].time\_end=time;

                    finished[finished\_index].time\_serve=doing.time\_serve;

                    finished[finished\_index].time\_turnaround=time-doing.time\_arrive;

                    finished[finished\_index].time\_turnaround\_rights=1.0\*finished[finished\_index].time\_turnaround/doing.time\_serve;

                }

                else doing\_process.push(doing);

            }

        }

        void display()

        {

            printf("Process name\t");

            printf("Finished time\t");

            printf("Turnaround time\t");

            printf("Weighted turnaround time\t\n");

            for(int i=1;i<=finished\_index;++i)

            {

                printf("|\t%c\t|\t",finished[i].process\_name);*//输出进程名称*

                printf("%d\t|\t",finished[i].time\_end);*//输出进程完成时间*

                printf("%d\t|\t",finished[i].time\_turnaround);*//周转时间*

                printf("%.2lf\t|\n",finished[i].time\_turnaround\_rights);*//带权周转时间*

            }

        }

        void display\_avergae()

        {

            double average\_time\_turnaround=0;

            double average\_time\_turnaround\_rights=0;

            for(int i=1;i<=finished\_index;++i)

            {

                average\_time\_turnaround+=finished[i].time\_turnaround;

                average\_time\_turnaround\_rights+=finished[i].time\_turnaround\_rights;

            }

            average\_time\_turnaround/=finished\_index;

            average\_time\_turnaround\_rights/=finished\_index;

            printf("avergae turnaround time=%.2lf\n",average\_time\_turnaround);

            printf("average weighted turnaround time=%.2lf",average\_time\_turnaround\_rights);

        }

    };

}

int main()

{

    printf("\n\nThis is RR:\n\n\n");*//使用新来进程优先的算法*

    os::RR \*rr=new os::RR();

    rr->conduct();

    rr->display();

    rr->display\_avergae();

    delete(rr);

    printf("\n\n\n");

    printf("\n\nThis is SJF:\n\n\n");*//使用新来进程优先的算法*

    os::SJF \*sjf=new os::SJF();

    sjf->conduct();

    sjf->display();

    sjf->display\_avergae();

    delete(sjf);

    printf("\n\n\n");

    return 0;

}

# 题目：进程的同步与互斥

## 1.1 实验内容

* 实验目的：分析进程争用资源的现象，学习解决进程互斥的方法。
* 设计内容：

用程序实现生产者—消费者问题。具体问题描述：一个仓库可以存放K件物品。生产者每生产一件产品，将产品放入仓库，仓库满了就停止生产。消费者每次从仓库中去一件物品，然后进行消费，仓库空时就停止消费。

数据结构：

Producer - 生产者进程，Consumer - 消费者进程

buffer: array [0..k-1] of integer;

in, out: 0..k-1; in记录第一个空缓冲区，out记录第一个不空的缓冲区

s1,s2,mutex: semaphore; s1控制缓冲区不满,s2控制缓冲区不空,mutex保护临界区；

初始化s1=k,s2=0,mutex=1

原语描述：

producer（生产者进程）：

item\_Type item;

{

while (true)

{

produce(&item);

p(s1);

p(mutex);

buffer[in]:=item;

in:=(in+1) mod k;

v(mutex);

v(s2);

}

}

consumer（消费者进程）：

item\_Type item;

{

while (true)

{

p(s2);

p(mutex);

item:=buffer[out];

out:=(out+1) mod k;

v(mutex);

v(s1);

}

}

## 1.2算法描述

* 对算法的原理进行系统描述。

## 1.3 实验结果

* 展示运行部分，把结果统计为表格展示；
* 需要对结果有一定的解释说明。

## 1.4 实现小结

* 不少于150字，主要写关于在实现过程中遇到的问题，以及如何解决的。

## 1.5实验代码

* 实现代码，需要提供清晰的注释；
* 采用CSDN网站代码的显示方式，可以通过网址实现：<https://highlightcode.com>；
* 代码格式：小五，1.0行间距。

# 题目：存储管理

## 1.1 实验内容

* 实验的具体内容；
* 参考任务书；
* 格式要求：正文，五号，宋体

## 1.2算法描述

* 对算法的原理进行系统描述。

## 1.3 实验结果

* 展示运行部分，把结果统计为表格展示；
* 需要对结果有一定的解释说明。

## 1.4 实现小结

* 不少于150字，主要写关于在实现过程中遇到的问题，以及如何解决的。

## 1.5实验代码

* 实现代码，需要提供清晰的注释；
* 采用CSDN网站代码的显示方式，可以通过网址实现：<https://highlightcode.com>；
* 代码格式：小五，1.0行间距。