



中山大學

SUN YAT-SEN UNIVERSITY

实 验 报 告

课程名称： 操作系统

姓 名： 方桂安

学 号： 20354027

专业班级： 2020 级智能科学与技术

任课教师： 吴贺俊

2022 年 10 月 9 日

实验报告成绩评定表

评定项目	内 容	满 分	评 分	总 分
实验态度	态度端正、遵守纪律、出勤情况	10		
实验过程	按要求完成算法设计、代码书写、注释清晰、运行结果正确	30		
实验记录	展示讲解清楚、任务解决良好、实验结果准确	20		
报告撰写	报告书写规范、内容条理清楚、表达准确规范、上交及时、无抄袭，抄袭记 0 分，提供报告供抄袭者扣分。	40		
评语：				
<div style="height: 180px;"></div>				
指导老师签字：_____ 年 月 日				

实验三 进程创建与控制调度

一、 实验目的

- 1、 创建一个进程，掌握创建进程的方法，理解进程和程序的区别。
 - a) 加深对进程概念的理解，进一步认识并发执行的实质。
 - b) 掌握 Linux 操作系统中进程的创建和终止操作。
 - c) 掌握在 Linux 操作系统中创建子进程并加载新映像的操作。
- 2、 调试跟踪进程创建的执行过程，了解进程的创建过程，理解进程是资源分配的单位。
- 3、 理解常用进程调度算法的具体实现。

二、 实验内容

1. 任务描述

- 1) Linux 的文件访问权限设置
- 2) 重定向与管道
- 3) 创建进程的系统调用
- 4) 进程调度算法模拟

2. 实验方案

- (1) 编写一个 C 程序，并使用系统调用 `fork()` 创建一个子进程。要求如下：

①在子进程中分别输出当前进程为子进程的提示、当前进程的 PID 和父进程的 PID、根据用户输入确定当前进程的返回值、退出提示等信息。

②在父进程中分别输出当前进程为父进程的提示、当前进程的 PID 和子进程的 PID、等待子进程退出后获得的返回值、退出提示等信息。

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

(3) 编写 C 程序模拟实现单处理机系统中的进程调度算法，实现对多个进程的调度模拟，要求采用常见进程调度算法（如先来先服务、时间片轮转和优先级调度等算法）进行模拟调度。

3. 实验说明

– 数据结构设计

- PCB：结构体
- 就绪队列：链表，每个节点为进程 PCB
- 进程状态

– 调度算法设计

- 具体调度算法：FCFS、SJF、PR
- 涉及多种操作：排序、链表操作

– 程序输出设计

- 调度进程的顺序、每个进程的起始时间、终止时间等
- CPU 每次调度的过程

三、实验记录

1. 实施步骤

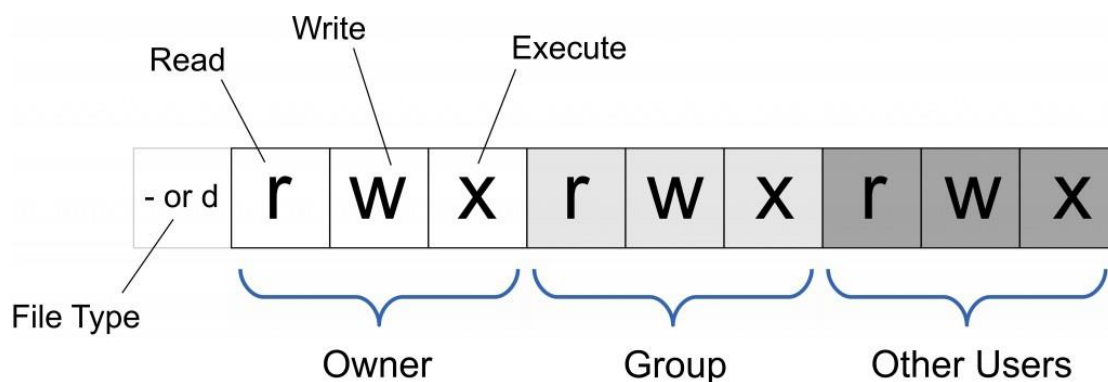
首先根据 PPT 指导进行命令的熟悉，结果如下：

```
enderfga@enderfga-virtual-machine: ~/test
enderfga@enderfga-virtual-machine:~/test$ vi file1
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 4
-rw-rw-r-- 1 enderfga enderfga 12 10月 14 15:43 file1
enderfga@enderfga-virtual-machine:~/test$ chmod o+w file1
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 4
-rw-rw-rw- 1 enderfga enderfga 12 10月 14 15:43 file1
enderfga@enderfga-virtual-machine:~/test$ chmod g-r file1
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 4
-rw--w-rw- 1 enderfga enderfga 12 10月 14 15:43 file1
enderfga@enderfga-virtual-machine:~/test$ chmod 755 file1
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 4
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
enderfga@enderfga-virtual-machine:~/test$
```

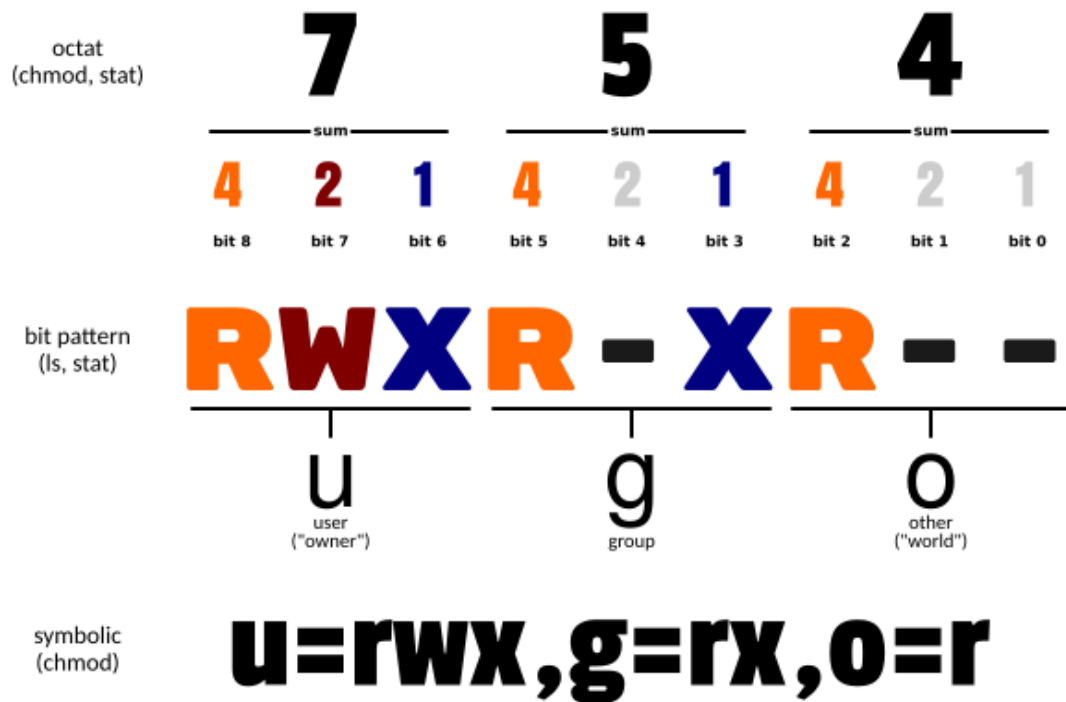
其中的 vi/ls 在实验一已经学习过，本次学习 chmod：

Linux chmod（英文全拼：change mode）命令是控制用户对文件的权限的命令

Linux/Unix 的文件调用权限分为三级：文件所有者（Owner）、用户组（Group）、其它用户（Other Users）。



只有文件所有者和超级用户可以修改文件或目录的权限。可以使用绝对模式（八进制数字模式），符号模式指定文件的权限。



使用权限：所有使用者

语法

```
chmod [-cfvR] [--help] [--version] mode file...
```

参数说明

mode：权限设定字符串，格式如下：

```
[ugoa...][[+=][rwxX]...][,...]
```

其中：

- u 表示该文件的拥有者，g 表示与该文件的拥有者属于同一个群体 (group) 者，o 表示其他以外的人，a 表示这三者皆是。
- + 表示增加权限、- 表示取消权限、= 表示唯一设定权限。
- r 表示可读取，w 表示可写入，x 表示可执行，X 表示只有当该文件是个子

目录或者该文件已经被设定过为可执行。

其他参数说明：

- `-c` : 若该文件权限确实已经更改, 才显示其更改动作
- `-f` : 若该文件权限无法被更改也不要显示错误讯息
- `-v` : 显示权限变更的详细资料
- `-R` : 对目前目录下的所有文件与子目录进行相同的权限变更(即以递归的方式逐个变更)
- `--help` : 显示辅助说明
- `--version` : 显示版本

接下来的重定向的学习, 内容如下:

```
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 4
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
enderfga@enderfga-virtual-machine:~/test$ ls -l > list
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 8
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
-rw-rw-r-- 1 enderfga enderfga 121 10月 26 10:33 list
enderfga@enderfga-virtual-machine:~/test$ cat list
总用量 4
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
-rw-rw-r-- 1 enderfga enderfga 0 10月 26 10:33 list
enderfga@enderfga-virtual-machine:~/test$ ls -l >> list
enderfga@enderfga-virtual-machine:~/test$ cat list
总用量 4
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
-rw-rw-r-- 1 enderfga enderfga 0 10月 26 10:33 list
总用量 8
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
-rw-rw-r-- 1 enderfga enderfga 121 10月 26 10:33 list
enderfga@enderfga-virtual-machine:~/test$ ls -l > list
enderfga@enderfga-virtual-machine:~/test$ cat list
总用量 4
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
-rw-rw-r-- 1 enderfga enderfga 0 10月 26 10:41 list
enderfga@enderfga-virtual-machine:~/test$
```

命令	说明
<code>command > file</code>	将输出重定向到 <code>file</code> 。
<code>command < file</code>	将输入重定向到 <code>file</code> 。
<code>command >> file</code>	将输出以追加的方式重定向到 <code>file</code> 。
<code>n > file</code>	将文件描述符为 <code>n</code> 的文件重定向到 <code>file</code> 。
<code>n >> file</code>	将文件描述符为 <code>n</code> 的文件以追加的方式重定向到 <code>file</code> 。
<code>n >& m</code>	将输出文件 <code>m</code> 和 <code>n</code> 合并。
<code>n <& m</code>	将输入文件 <code>m</code> 和 <code>n</code> 合并。
<code><< tag</code>	将开始标记 <code>tag</code> 和结束标记 <code>tag</code> 之间的内容作为输入。

注意：0 是标准输入（STDIN），1 是标准输出（STDOUT），2 是标准错误输出（STDERR）。这里的 2 和 > 之间不可以有空格，2> 是一体的时候才表示错误输出。

最后是关于管道的命令，可以利用 `explainshell` 来学习其内容：

```
enderfga@enderfga-virtual-machine:~/test$ who
enderfga tty2          2022-10-26 01:05 (tty2)
enderfga@enderfga-virtual-machine:~/test$ who | grep enderfga
enderfga tty2          2022-10-26 01:05 (tty2)
enderfga@enderfga-virtual-machine:~/test$ who | grep root
enderfga@enderfga-virtual-machine:~/test$
```

`who` 命令用于显示系统中有哪些使用者正在上面，显示的资料包含了使用者 ID、使用的终端机、从哪边连上来的、上线时间、呆滞时间、CPU 使用量、动作等等。`grep` 命令用于查找文件里符合条件的字符串。

通过管道可以在 `who` 显示出来的使用者中插座对应的用户，如 `root`；

由于虚拟机中只有我一个用户故 `who | grep root` 没有输出，改为我自己的用户名则会将 `enderfga` 高亮显示。

```
enderfga@enderfga-virtual-machine:~/test$ ls -l
总用量 8
-rwxr-xr-x 1 enderfga enderfga 12 10月 14 15:43 file1
-rw-rw-r-- 1 enderfga enderfga 121 10月 26 10:41 list
enderfga@enderfga-virtual-machine:~/test$ ls -l | wc -l
3
```

利用 `wc` 指令我们可以计算文件的 Byte 数、字数、或是列数，若不指定文件名称、或是所给予的文件名为“-”，则 `wc` 指令会从标准输入设备读取数据。

参数：

- `-c` 或 `--bytes` 或 `--chars` 只显示 Bytes 数。
- `-l` 或 `--lines` 显示行数。
- `-w` 或 `--words` 只显示字数。

结合 `ls -l` 输出的结果，故命令 `ls -l | wc -l` 的结果为 3。

2. 实验记录

编写 C 程序 1：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
/* ①在子进程中分别输出当前进程为子进程的提示、当前进程的 PID 和父进程的 PID、
根据用户输入确定当前进程的返回值、退出提示等信息。
②在父进程中分别输出当前进程为父进程的提示、当前进程的 PID 和子进程的 PID、等待
子进程退出后获得的返回值、退出提示等信息。
*/
int main() {
    pid_t pid;
    pid = fork();
```

```

    if (pid < 0) {
        printf("fork error!\n");
        return -1;
    } else if (pid == 0) {
        printf("I am child process, my pid is %d, my parent pid is %d\n",
getpid(), getppid());
        int ret;
        printf("Please input a number: ");
        scanf("%d", &ret);
        printf("I am child process, I will exit with %d\n", ret);
        return ret;
    } else {
        printf("I am parent process, my pid is %d, my child pid is %d\n",
getpid(), pid);
        int status;
        wait(&status);
        printf("I am parent process, my child process exit with %d\n",
WEXITSTATUS(status));
        return 0;
    }
}

```

编写 C 程序 2:

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
/*
编写另一个C 程序,使用系统调用fork() 以创建一个子进程,并使用这个子进程调用exec
函数族以执行系统命令ls。
*/
int main()
{
    pid_t pid;
    pid = fork();
    if (pid < 0)
    {
        printf("fork error\n");
    }
    else if (pid == 0)
    {
        printf("child process\n");
        execl("/bin/ls", "ls", NULL);
    }
}

```

```

    }
else
{
    printf("parent process\n");
    wait(NULL);
}
return 0;
}

```

编写 C 程序 3:

详见五、附：程序模块的源代码


3. 实验结果

```

● enderfga@enderfga-virtual-machine:~/os$ gcc test.c -o test
● enderfga@enderfga-virtual-machine:~/os$ ./test
I am parent process, my pid is 7735, my child pid is 7736
I am child process, my pid is 7736, my parent pid is 7735
Please input a number: 5
I am child process, I will exit with 5
I am parent process, my child process exit with 5
○ enderfga@enderfga-virtual-machine:~/os$ █

```

第一个程序实现了在 Linux 下使用 `fork()` 创建进程，并验证 `fork()` 的返回值。首先在主程序中通过 `fork()` 创建子进程，并根据 `fork()` 的返回值确定所处的进程是子进程还是父进程，然后分别在子进程和当前进程（父进程）中调用 `getpid()`、`getppid()`、`wait()` 等函数以完成具体要求。



```

资源管理器  ...  问题  输出  调试控制台  终端
v os
  test
  C test.c
  ● enderfga@enderfga-virtual-machine:~/os$ gcc test.c -o test
  ● enderfga@enderfga-virtual-machine:~/os$ ./test
  parent process
  child process
  test test.c
  ○ enderfga@enderfga-virtual-machine:~/os$ █

```

第二个程序实现了在 Linux 下使用 `fork()` 创建进程、并使用 `exec` 族函数来加载新进程的映像。这里使用到的命令是 `ls`，而如 `vscode` 中所示，

此时 os 目录下只有 test, test.c 两个文件，故在对应进程输出自己是子进程还是父进程后，终端输出了：test test.c。

第三个程序中我实现了：

1. 先来先服务（FCFS）、2. 短作业优先（SJF）、3. 优先级调度算法（PSA）

```
enderfga@enderfga-virtual-machine:~/os$ ./schedule
请输入被调度的进程数目：5
请输入调度算法序号：1.先来先服务（FCFS）、2.短作业优先（SJF）、3.优先级调度算法（PSA） 1
进程号No.0:
输入进程名:█
```

通过输入 1, 2, 3 选择对应的调度算法，

```
**** 当前正在运行的进程是:p1
qname  state  nice  ndtime  runtime
p1     R     43    12      0

****当前就绪队列状态为:
qname  state  nice  ndtime  runtime
p2     W     51    12      0
qname  state  nice  ndtime  runtime
p3     W     54    66      0
qname  state  nice  ndtime  runtime
p4     W     75     9      0
qname  state  nice  ndtime  runtime
p5     W     89    34      0
```

FCFS 算法特点

- 按照作业提交或变为后备状态的先后次序分配 CPU
- 新作业只有当当前的作业执行完成或者阻塞才能获得 CPU
- 被唤醒的作业不会立即恢复执行，默认是非抢占式，通常要等到当前的作业让出 CPU

算法的优缺点

- 有利于 CPU 繁忙型的作业，不利于 I/O 繁忙的作业，如果后备队列中作业过多，新加入的作业等待的时间会很长。

- 一般 FCFS 算法在单处理机中已很少作为主调度算法，但经常把它与其他调度算法相结合使用，形成一种更为有效的调度算法。

```

**** 当前正在运行的进程是:p2
qname  state  nice  ndtime  runtime
p2     R     54    1      0

****当前就绪队列状态为:
qname  state  nice  ndtime  runtime
p3     W     6     3      0
qname  state  nice  ndtime  runtime
p4     W     9     3      0
qname  state  nice  ndtime  runtime
p1     W     4     6      0
qname  state  nice  ndtime  runtime
p5     W     9     9      0

```

SJF 算法特点

- 按照作业的长短来计算优先级，作业越短，其优先级越高
- 作业长度是按照作业所需要占用的 CPU 时间来衡量的
- 属于非抢占式，只有当当前的作业释放出 CPU，新作业才可能获得 CPU

算法的优缺点

- SJF 算法对短作业有利，能有效的降低作业的平均等待时间，提高系统的吞吐量。
- 但未考虑作业的紧迫程度，因而不能保证紧迫性作业的及时处理；
- 还必须预知作业的运行时间，作业的运行时间需要进行估计，如果估计过低，作业未完成就会提前终止，所以一般都会偏长估计；
- 对长作业非常不利，长作业的周转时间明显偏长。
- 完全忽略作业的等待时间，可能使作业等待时间过长，出现饥饿现象；
- 人机无法交互。

```

**** 当前正在运行的进程是:p1
qname  state  nice  ndtime  runtime
p1      R      13      3        0

****当前就绪队列状态为:
qname  state  nice  ndtime  runtime
p5      W      7      9        0

qname  state  nice  ndtime  runtime
p4      W      6      2        0

qname  state  nice  ndtime  runtime
p2      W      3      4        0

qname  state  nice  ndtime  runtime
p3      W      1      4        0

```

FCFS 算法和 SJF 算法都可以看做一种特殊的 PSA 算法。

例如，在 FCFS 算法里，是将算法的到达时间当做优先级，先到达的优先级高。在 SJF 算法里，将作业的服务时间看做优先级，服务时间短的优先级高。

四、 总结与讨论

- 1 、文件 backup.tar 的权限如下：

```
-rw-r--r--  1 root  root  19274 Jul 14 11:00 backup.tar
```

请问-rw-r--r--的含义是什么？

一共十个字符

第一个字符:表示文件类型，d 是文件夹，l 是连接文件，-是普通文件。

后面的 9 个字符表示权限。

权限分为 4 种，r 表示读取权限，w 表示写入权限，x 表示执行权限，-表示无此权限。

9 个字符共分为 3 组，每组 3 个字符。第 1 组表示创建这个文件的用户的权限，第 2 组表示创建这个文件的用户所在的组的权限，第 3 组表示其他

用户的权限。

在每组中的 3 个字符里，第 1 个字符表示读取权限，第 2 个字符表示写入权限，第 3 个字符表示执行权限。如果有此权限，则对应位置为 r, w 或 x，如果没有此权限，则对应位置为-。

所以说-rw-r - r--，表示这是一个普通文件，创建文件的用户的权限为 rw-，创建文件的用户所在的组的权限为 r -，其他用户的权限为 r -。

- 2、文件 backup.tar 的所有者添加执行权限的命令是什么？

```
chmod u+x backup.tar
```

- 3、赋予所有用户读和写 backup.tar 文件权限的命令是？

```
chmod 766 backup.tar
```

- 4、如何把一个 backup 文件夹打包成 backup.tar？

```
tar -cvf backup.tar backup
```

五、附：程序模块的源代码

```
#include "stdio.h"
#include <stdlib.h>
#define getpch(type) (type*)malloc(sizeof(type))
int choice;
struct pcb { /* 定义进程控制块 PCB */
    char name[10]; //进程名
    char state;    //进程状态: "W"-就绪态, "R"-运行态
    int nice;      //进程优先级
    int ntime;     //需要运行时间
    int rtime;     //已经运行的时间
    struct pcb* link;
}*ready=NULL,*p; // ready 进程表示当前运行的进程; P 进程为新创建的进程
typedef struct pcb PCB;

// 自行选择一种调度算法, 完成进程优先级排序; FIFO、时间片轮转调度、高优先级、短作业优先、高响应比优先算法等
char PSA() /* 建立对进程进行优先级排列函数, 优先数大者优先 */
{
```

```

PCB *first, *second;
int insert=0;
if((ready==NULL) || ((p->nice)>(ready->nice))) /*优先级最大者,插入队首
*/
{
    p->link=ready;
    ready=p;
}
else /* 进程比较优先级,插入适当的位置中*/
{
    first=ready;
    second=first->link;
    while(second!=NULL)
    {
        /*若插入进程比当前进程优先数大,*/
        if((p->nice)>(second->nice))
        { /*插入到当前进程前面*/
            p->link=second;
            first->link=p;
            second=NULL;
            insert=1;
        }
        else /* 插入进程优先数最低,则插入到队尾*/
        {
            first=first->link;
            second=second->link;
        }
    }
    if(insert==0) first->link=p;
}
}

```

```

char SJF()
{
    PCB *first, *second;
    int insert=0;
    if((ready==NULL) || ((p->ntime)<(ready->ntime)))
    {
        p->link=ready;
        ready=p;
    }
    else
    {
        first=ready;

```



```

second=first->link;
while(second!=NULL)
{
    if((p->ntime)<(second->ntime))
    { /*插入到当前进程前面*/
        p->link=second;
        first->link=p;
        second=NULL;
        insert=1;
    }
    else
    {
        first=first->link;
        second=second->link;
    }
}
if(insert==0) first->link=p;
}
}

```

```

char FCFS()
{
    //使用首插入法链接链表
    PCB *first, *second;
    int insert=0;
    if(ready==NULL)
    {
        p->link=ready;
        ready=p;
    }
    else
    {
        first=ready;
        second=first->link;
        while(second!=NULL)
        {
            first=first->link;
            second=second->link;
        }
        first->link=p;
    }
}
}

```

```

char input() /* 建立进程控制块函数*/

```

```

{
    int i,num;
    printf("\n 请输入被调度的进程数目: ");
    scanf("%d",&num);
    printf("\n 请输入调度算法序号: 1.先来先服务 (FCFS)、2.短作业优先 (SJF)、
3.优先级调度算法 (PSA) ");

    scanf("%d",&choice);
    for(i=0;i<num;i++)
    {
        printf("\n 进程号 No.:%d:",i);
        p=getpch(PCB);
        printf("\n 输入进程名:");
        scanf("%s",p->name);
        printf(" 输入进程优先数:");
        scanf("%d",&p->nice);
        printf(" 输入进程运行时间:");
        scanf("%d",&p->ntime);
        printf("\n");
        p->rtime=0;
        p->state='W';
        p->link=NULL;
        //根据选择的调度算法, 调用不同的排序函数
        switch(choice){
            case 1:
                FCFS();
                break;
            case 2:
                SJF();
                break;
            case 3:
                PSA();
                break;
            default:
                printf("输入错误");
                break;
        }
    }
}

int space()
{
    int l=0; PCB* pr=ready;
    while(pr!=NULL)

```

```

    {
        l++;
        pr=pr->link;
    }
    return(l);
}

char disp(PCB * pr) /*建立进程显示函数,用于显示当前进程*/
{
    printf("\n qname \t state \t nice \tndtime\truntime \n");
    printf("%s\t",pr->name);
    printf("%c\t",pr->state);
    printf("%d\t",pr->nice);
    printf("%d\t",pr->ntime);
    printf("%d\t",pr->rtime);
    printf("\n");
}

char check() /* 建立进程查看函数 */
{
    PCB* pr;
    printf("\n ***** 当前正在运行的进程是:%s",p->name); /*显示当前运行进程*/
    disp(p);
    pr=ready;
    if (pr!=NULL)
        printf("\n *****当前就绪队列状态为:"); /*显示就绪队列状态*/
    else
        printf("\n *****当前就绪队列状态为: 空\n"); /*显示就绪队列状态为空*/
    while(pr!=NULL)
    {
        disp(pr);
        pr=pr->link;
    }
}

char destroy() /*建立进程撤消函数(进程运行结束,撤消进程)*/
{
    printf(" 进程 [%s] 已完成.\n",p->name);
    free(p);
}

char running() /* 建立进程就绪函数(进程运行时间到,置就绪状态*/
{
    (p->rtime)++;
}

```

```

if(p->rtime==p->ntime)
destroy(); /* 调用destroy函数*/
else
{
    (p->nice)--;
    p->state='W';
    switch(choice){
        case 1:

        case 2:
            SJF();
            break;
        case 3:
            PSA();
            break;
        default:
            printf("输入错误");
            break;
    }
}
}

int main() /*主函数*/
{
    int len,h=0;
    char ch;
    input();
    len=space();
    while((len!=0)&&(ready!=NULL))
    {
        ch=getchar();
        h++;
        printf("\n The execute number:%d \n",h);
        p=ready;
        ready=p->link;
        p->link=NULL;
        p->state='R';
        check();
        running();
        printf("\n 按任一键继续.....");
        ch=getchar();
    }
    printf("\n\n 所有进程已经运行完成! \n");
    ch=getchar(); }

```