**大连理工大学**

**本科实验报告**

课程名称： 操作系统实验

学院（系）： 电子信息与电气工程学部

专 业： 计算机科学与技术

班 级： 电计1503班

学 号： 201585026

学生姓名： 杜鹏远

年 月 日

**实验一 进程管理**

**1.1 实验目的**

加深对于进程并发执行概念的理解。实践并发进程的创建和控制方法。观察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习Linux系统中进程创建与控制有关的系统调用的编程和调试技术。

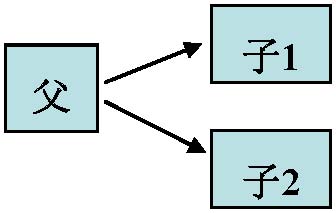
**1.2 实验原理**

进程可以通过系统调用fork()创建子进程并和其子进程并发执行.子进程初始的执行映像是父进程的一个复本.子进程可以通过exec()系统调用族装入一个新的执行程序。父进程可以使用wait()或waitpid()系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

fork成功创建子进程后将返回子进程的进程号,不成功会返回-1。

**1.3 实验内容**

1.每个进程都执行自己独立的程序，打印自己的pid，每个父进程打印其子进程的pid;

2.每个进程都执行自己独立的程序，打印自己的pid，父进程打印其子进程的pid;

3.编写一个命令处理程序，能处理max(m,n), min(m,n),average(m,n,l)这几个命令。

**1.4 实验代码及截图**

内容一：

int fork\_test1(void) {

pid\_t childPid1;

pid\_t childPid2=-1;

childPid1 = fork();

if (childPid1 == -1) {

printf("failed to create a new process!\n");

return 0;

}

if (childPid1 == 0) {

childPid2 = fork();

if (childPid2 == -1) {

printf("failed to create a new process!\n");

return 0;

}

if (childPid2 == 0) {

printf("I'm the child2 process, my process id is \033[;32;0m%d\033[0m\n", getpid());

exit(0);

}

if (childPid2 > 0) {

printf("I'm the child1 process, my process id is \033[;32;0m%d\033[0m, child2's pid is \033[;32;0m%d\033[0m\n", getpid(), childPid2);

wait(NULL);

exit(0);

}

}

if (childPid1 > 0) {

printf("I'm the father process, my process id is \033[;32;0m%d\033[0m, child1's pid is \033[;32;0m%d\033[0m\n", getpid(), childPid1);

wait(NULL);

}

return 0;

}

内容二：

int fork\_test2(void) {

pid\_t childPid1=-1;

pid\_t childPid2=-1;

childPid1 = fork();

if(childPid1 !=0) {

childPid2 = fork();

}

if (childPid1 == 0) {

printf("I'm the child1 process, my process id is \033[;32;0m%d\033[0m\n", getpid());

exit(0);

}

if (childPid2 == 0) {

printf("I'm the child2 process, my process id is \033[;32;0m%d\033[0m\n", getpid());

exit(0);

}

if(childPid1 > 0 && childPid2 > 0) {

printf("I'm the father process, my process id is \033[;32;0m%d\033[0m\n", getpid());

printf("my child1 id is \033[;32;0m%d\033[0m", childPid1);

printf("and my child2 id is \033[;32;0m%d\033[0m\n", childPid2);

wait(NULL);

}

if(childPid1 <=0) {

exit(0);

}

if(childPid2 <=0)

exit(0);

printf("\n");

return 0;

}

内容三：

void getCmd(char \*input, char \*parameterlist[]) {

char \*tmp = input;

int count = 0;

int i = 0;

for (i = 0; \*tmp != '(' && i < 20; tmp++, i++) {

parameterlist[count][i] = \*tmp;

}

tmp ++;

count ++;

while (\*tmp) {

i = 0;

while (\*tmp > 48 && \*tmp < 58) {

parameterlist[count][i] = \*tmp;

tmp ++;

i++;

}

count ++;

tmp ++;

}

}

int threeCommand(void) {

char input[20];

char \*parameterlist[10];

pid\_t pid = -1;

for(int i = 0; i < 10; i++)

parameterlist[i] = (char\*)malloc(100 \* sizeof(char));

while(1) {

for(int i = 0; i < 10; i++)

memset(parameterlist[i], 0, 100 \* sizeof(char));

for (int i = 0; i < 20; ++i)

input[i] = 0;

printf(">");

scanf("%s", input);

if(!strcmp(input, "q"))

exit(0);

pid = fork();

if (pid == -1) {

printf("Fail to create a new process!");

}

else if (pid > 0) {

wait(NULL);

}

else if (pid == 0) {

getCmd(input, parameterlist);

if(!strcmp(parameterlist[0], "max")) {

char \*argv[] = {"max", parameterlist[1], parameterlist[2], NULL};

execve("./max", argv, NULL);

}

else if(!strcmp(parameterlist[0], "min")) {

char \*argv[] = {"min", parameterlist[1], parameterlist[2], NULL};

execve("./min", argv, NULL);

}

else if(!strcmp(parameterlist[0], "average")) {

char \*argv[] = {"average", parameterlist[1], parameterlist[2], parameterlist[3], NULL};

execve("./average", argv, NULL);

}

exit(0);

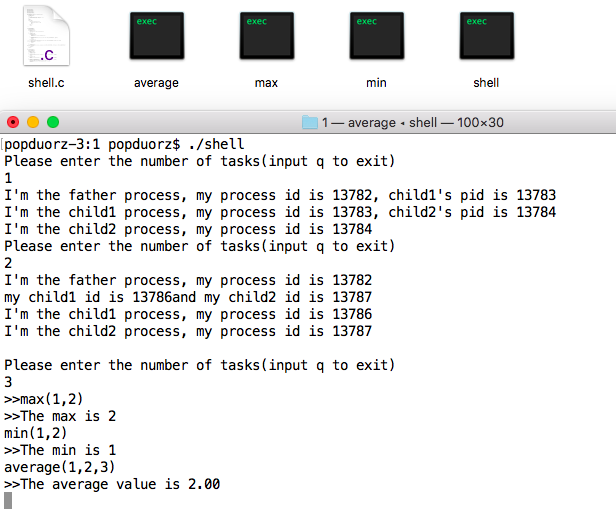
}

}

return 0;

}

截图：



**1.5 结果分析**

1.4的截图内容包含了三个分实验。

从以上结果中可以看出，在第一个分实验中，父进程13782创建了子进程13783，然后子进程13783创建了孙子进程13784。在第二个分实验中，父进程13782创建了子进程13786和13787。在第三个实验中，按照要求的格式执行命令，三分命令都反返回了正确的结果。

**实验二 处理器调度**

**2.1 实验目的**

熟悉使用各种单处理器调度的各种算法，加深对于处理机调度机制的理解。练习模拟算法的编程技巧，锻炼分析试验数据的能力。

**2.2 实验内容**

随机给出一个进程调度实例，如：

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

模拟进程调度，给出按照算法先来先服务FCFS、轮转RR（q=1）、最短进程优先SJF、最高响应比优先HRN进行调度各进程的完成时间、周转时间、带权周转时间。

**2.3 实验代码及截图**

class Process(object):

def \_\_init\_\_(self, \_p\_id, \_arrive\_time, \_service\_time):

self.p\_id = {\_p\_id: \_p\_id}

self.arrive\_time = {\_p\_id: \_arrive\_time}

self.service\_time = {\_p\_id: \_service\_time}

self.finish\_time = {\_p\_id: 0}

self.remain\_time = {\_p\_id: \_service\_time}

class Processor(object):

def \_\_init\_\_(self, \_p\_id, \_arrive\_time, \_service\_time):

self.current\_time = 0

self.result = []

self.process = Process(p\_id, \_arrive\_time, \_service\_time)

def fcfs(self):

while len(self.process.p\_id):

pid = latest\_p

for p in self.process.p\_id:

if self.process.arrive\_time[p] < self.process.arrive\_time[pid]:

pid = p

if self.current\_time < self.process.arrive\_time[pid]:

sleep(1)

self.current\_time += 1

continue

process\_finish\_time = self.run\_process(pid, 1, self.current\_time, leaf\_values)

sleep(1)

if process\_finish\_time != 0:

self.process.process\_run\_info(pid, process\_finish\_time, self.result)

self.process.process\_exit(pid)

def rr(self):

while len(self.process.p\_id):

pop\_list = []

need\_sleep = True

for p in self.process.p\_id:

if self.process.arrive\_time[p] <= self.current\_time:

need\_sleep = False

n.cursor.restore()

process\_finish\_time=self.run\_process(p,1,self.current\_time, leaf\_values)

sleep(1)

if process\_finish\_time != 0:

self.process.process\_run\_info(p, process\_finish\_time, self.result)

pop\_list.append(p)

if need\_sleep:

self.current\_time += 1

for q in pop\_list:

self.process.process\_exit(q)

def sjf(self):

while len(self.process.p\_id):

pid = longest\_p

for p in self.process.p\_id:

if self.process.arrive\_time[p] <= self.current\_time:

if self.process.service\_time[p] < self.process.service\_time[pid]:

pid = p

if self.current\_time < self.process.arrive\_time[pid]:

self.current\_time += 1

continue

process\_finish\_time = self.run\_process(pid, 1, self.current\_time, leaf\_values)

if process\_finish\_time != 0:

self.process.process\_run\_info(pid, process\_finish\_time, self.result)

self.process.process\_exit(pid)

def hrn(self):

while len(self.process.p\_id):

identifier = "p\_id"

rr = 0

for p in self.process.p\_id:

p\_rr=(self.current\_time-self.process.arrive\_time[p]+

self.process.remain\_time[p]) /\

self.process.service\_time[p]

if p\_rr > rr:

identifier = p

rr = p\_rr

if self.current\_time < self.process.arrive\_time[identifier]:

self.current\_time += 1

continue

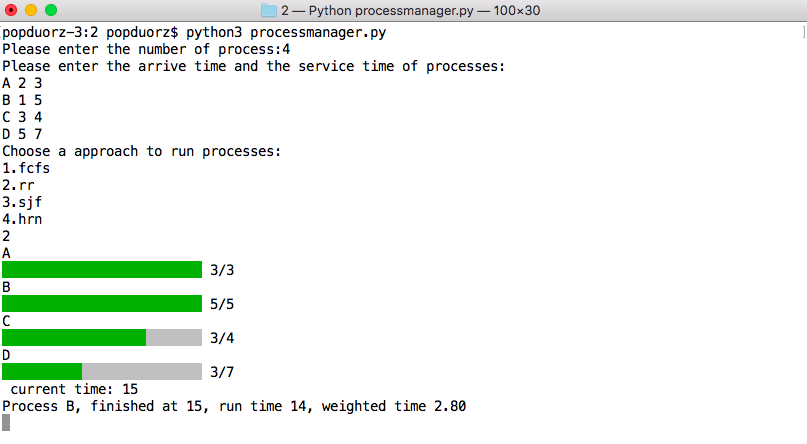
process\_finish\_time = self.run\_process(identifier, 1, self.current\_time, leaf\_values)

if process\_finish\_time != 0:

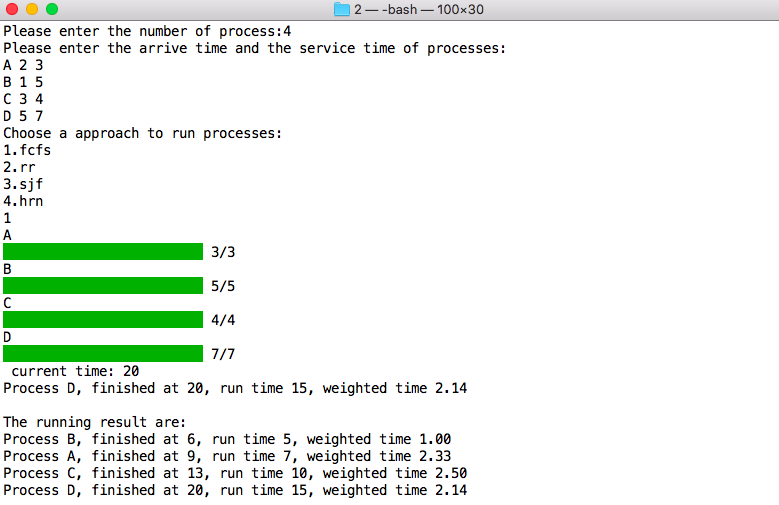
self.process.process\_run\_info(identifier, process\_finish\_time, self.result)

self.process.process\_exit(identifier)

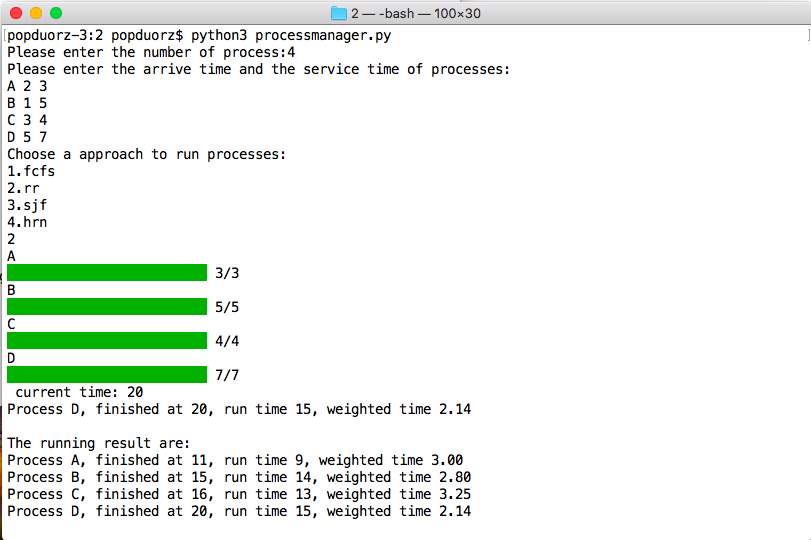
截图：



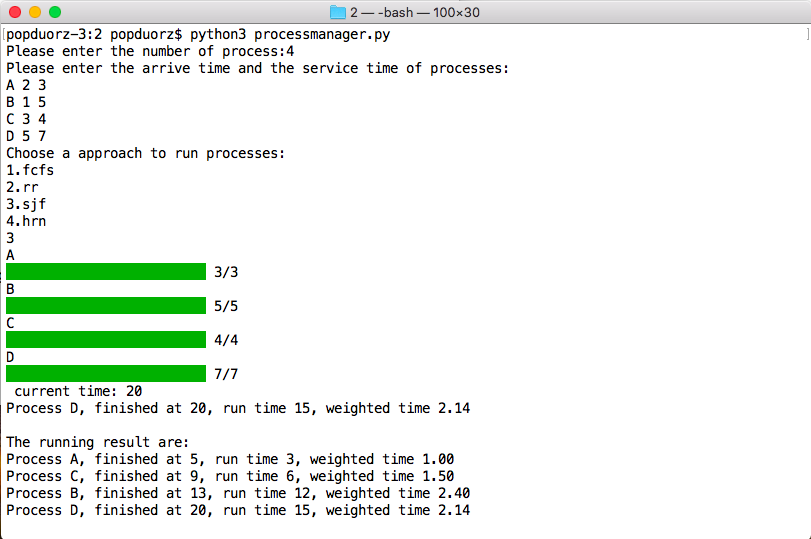
程序过程图



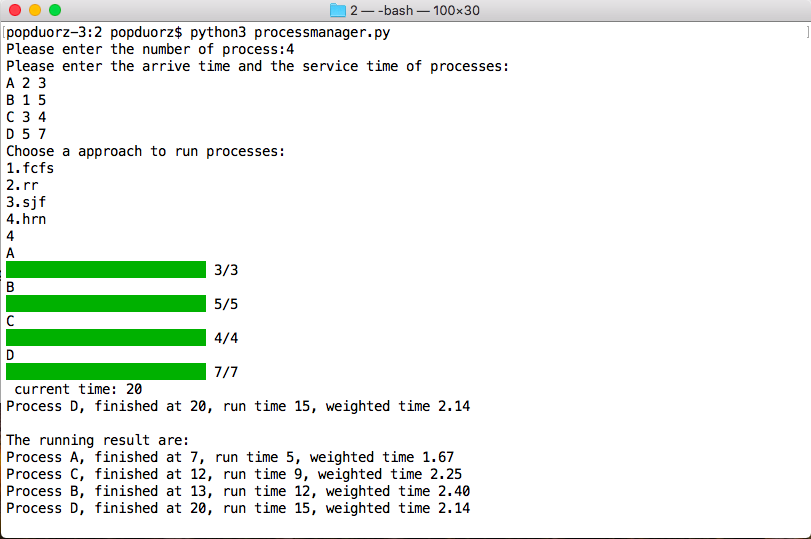
fcfs算法调度结果图



rr算法调度结果图



sjf算法调度结果图



hrn算法调度结果图

**2.5 结果分析**

图一为程序的过程图，可以看出，此程序可以把整个进程调度的过程用直观的图形化界面显示出来。

后面三张图分别是用fcfs、rr、sjf、hrn四种算法，四种算法的结果在进度条后都有给出。通过笔算可以验证程序的正确性。

**实验三 存储管理**

**3.1 实验目的**

加深对于存储管理的了解，掌握虚拟存储器的实现原理；观察和了解重要的页面置换算法和置换过程。练习模拟算法的编程技巧，锻炼分析试验数据的能力。

**3.2 实验内容**

1.示例实验程序中模拟两种置换算法：LRU算法和FIFO算法。

2.给定任意序列不同的页面引用序列和任意分配页面数目，显示两种算法的页置换过程。

3.能统计和报告不同置换算法情况下依次淘汰的页号、缺页次数（页错误数）和缺页率。

**3.3 实验代码及截图**

class VirtualStorage(object):

def \_\_init\_\_(self, num):

self.page\_num = num

self.lost = 0

self.hit = 0

self.page = []

for i in range(num):

self.page.append(0)

hit = False

def fifo(self, page\_id):

new = page\_id

old = self.page[0]

changed\_page\_num = self.page\_num

if page\_id in self.page:

self.hit = self.hit + 1

hit = True

changed\_page\_num = self.page.index(page\_id) + 1

else:

hit = False

for i in range(self.page\_num - 1):

self.page[i] = self.page[i + 1]

self.page[self.page\_num - 1] = page\_id

self.lost = self.lost + 1

self.print\_result(old, new, changed\_page\_num, hit)

def lru(self, page\_id):

new = page\_id

old = self.page[0]

changed\_page\_num = self.page\_num

if page\_id in self.page:

hit = True

changed\_page\_num = self.page.index(page\_id) + 1

self.hit = self.hit + 1

for i in range(self.page.index(page\_id), self.page\_num - 1):

self.page[i] = self.page[i + 1]

else:

hit = False

self.lost = self.lost + 1

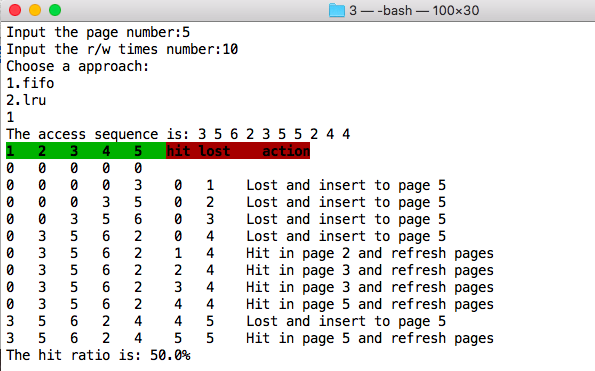
for i in range(self.page\_num - 1):

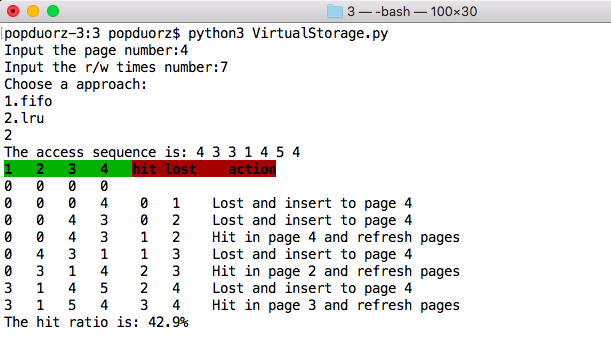
self.page[i] = self.page[i + 1]

self.page[self.page\_num - 1] = page\_id

self.print\_result(old, new, changed\_page\_num, hit)

截图：





**3.5 结果分析**

截图给出了随着访问的进行，各个物理块的储存情况。

第一行显示为最开始时各块的占用都是0。然后页面的访问，存入了相应的页号，hit表示命中的次数，lost表示缺页的次数，最右方表示每次访问时虚拟存储器所进行的动作。最后显示出了命中率。

每进行一次访问，程序都进行一次显示，整个过程被完整的记录下来。

通过手算可以验证程序的正确性。

**实验四 磁盘移臂调度算法**

**4.1 实验目的**

加深对于操作系统设备管理技术的了解，体验磁盘移臂调度算法的重要性；掌握几种重要的磁盘移臂调度算法，练习模拟算法的编程技巧，锻炼研究分析试验数据的能力。

**4.2 实验内容**

1. 示例实验程序中模拟两种磁盘移臂调度算法：SSTF算法和SCAN算法

2. 对两种算法给定序列不同的磁盘请求序列，显示响应磁盘请求的过程。

3. 能统计和报告不同算法情况下响应请求的顺序、移臂的总量。

**4.3 实验代码及截图**

class Disk(object):

def \_\_init\_\_(self, \_start\_position, \_movement\_num):

self.list\_for\_sstf = []

for m in range(\_movement\_num):

self.list\_for\_sstf.append(random.randint(1, 1000) % 500)

self.list\_for\_scan = self.list\_for\_sstf.copy()

self.position = \_start\_position

self.move\_for\_sstf = 0

self.move\_for\_scan = 0

self.x\_for\_sstf = [\_start\_position]

self.y\_for\_sstf = range(\_movement\_num + 1)

self.x\_for\_scan = [\_start\_position]

self.y\_for\_scan = range(\_movement\_num + 1)

def sstf(self):

pos = -1

print()

print('sstf route:')

while len(self.list\_for\_sstf) != 0:

for i in range(len(self.list\_for\_sstf)):

a = 100000

if abs(self.list\_for\_sstf[i] - self.position) < a:

a = abs(self.list\_for\_sstf[i] - self.position)

pos = i

self.move\_for\_sstf = self.move\_for\_sstf + abs(self.list\_for\_sstf[pos]

- self.position)

print(self.list\_for\_sstf[pos],"sstf move length:",self.move\_for\_sstf)

self.x\_for\_sstf.append(self.list\_for\_sstf[pos])

self.position = self.list\_for\_sstf[pos]

self.list\_for\_sstf.pop(pos)

def scan(self, cur\_dir):

direction = cur\_dir

print('scan route:')

while len(self.list\_for\_scan) != 0:

if direction == 0:

a = 10000000

pos = -1

for i in range(len(self.list\_for\_scan)):

if (self.list\_for\_scan[i] - self.position) < 0 and

abs(self.list\_for\_scan[i] - self.position) < a:

a = abs(self.list\_for\_scan[i] - self.position)

pos = i

if pos != -1:

self.move\_for\_scan = self.move\_for\_scan +

abs(self.list\_for\_scan[pos] - self.position)

print(self.list\_for\_scan[pos],"scan move length",self.move\_for\_scan)

self.position = self.list\_for\_scan[pos]

self.x\_for\_scan.append(self.list\_for\_scan[pos])

self.list\_for\_scan.pop(pos)

else:

direction = 1

else:

a = 10000000

pos = -1

for i in range(len(self.list\_for\_scan)):

if (self.list\_for\_scan[i] - self.position) > 0 and

abs(self.list\_for\_scan[i] - self.position) < a:

a = abs(self.list\_for\_scan[i] - self.position)

pos = i

if pos != -1:

self.move\_for\_scan = self.move\_for\_scan +

abs(self.list\_for\_scan[pos] - self.position)

print(self.list\_for\_scan[pos],"move length",self.move\_for\_scan)

self.position = self.list\_for\_scan[pos]

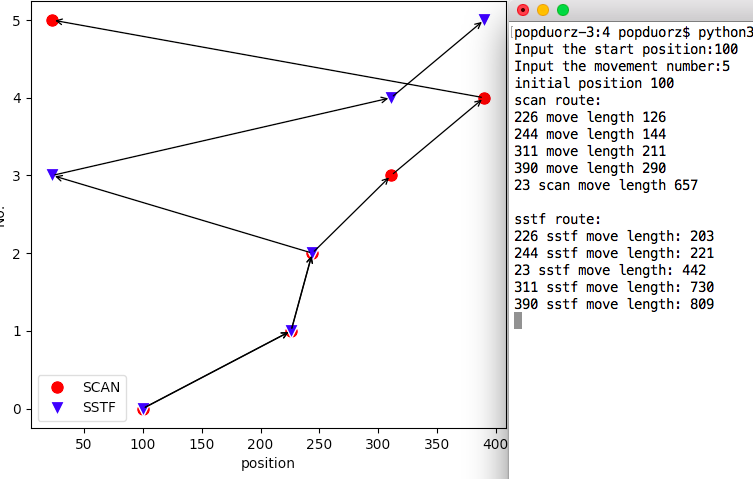
self.x\_for\_scan.append(self.list\_for\_scan[pos])

self.list\_for\_scan.pop(pos)

else:

direction = 0

截图：



**4.5 结果分析**

左边截图显示磁臂移动的图形化路径，右边截图显示磁臂移动的定量值。

两种算法的图形化结果和数值结果同时给出，便于对比分析。在图中有很好的区分标注。整个代码由python编写，图形由python代码绘制。

**实验五 文件管理**

**5.1 实验目的**

通过模拟文件的创建、删除操作，加深对于操作系统文件管理功能的了解，练习模拟算法的编程技巧，锻炼研究分析试验数据的能力。

**5.2 实验内容**

给出一个磁盘块序列：1、2、3、……、500，初始状态所有块为空的，每块的大小为2k。选择使用空闲表、空闲盘区链、位示图三种算法之一来管理空闲块。对于基于块的索引分配执行以下步骤：

1.随机生成2k-10k的文件50个，文件名为1.txt、2.txt、……、50.txt，按照上述算法存储到模拟磁盘中。

2.删除奇数.txt（1.txt、3.txt、……、49.txt）文件

3.新创建5个文件（A.txt、B.txt、C.txt、D.txt、E.txt），大小为：7k、5k、2k、9k、3.5k，按照与（1）相同的算法存储到模拟磁盘中。

4.给出文件A.txt、B.txt、C.txt、D.txt、E.txt的盘块存储状态和所有空闲区块的状态。

**5.3 实验代码及截图**

class File(object):

def \_\_init\_\_(self, \_name='untitled.txt', \_start=0, \_block\_length=0, \_size=0):

self.name = \_name

self.start = \_start

self.block\_length = \_block\_length

self.size = \_size

def get\_block\_length(self):

return self.block\_length

def get\_start(self):

return self.start

class Disk:

def \_\_init\_\_(self):

self.bitmap = [0 for i in range(500)]

def find\_block(self, \_length):

for \_start in range(0, 500):

has\_fragment = 0

for y in range(\_start, \_start + \_length):

if self.bitmap[y]:

has\_fragment = 1

break

if has\_fragment == 0:

for z in range(\_start, \_start + \_length):

self.bitmap[z] = 1

return \_start

def show\_disk(self):

for x in range(0,500):

if self.bitmap[x] == 0:

print("\033[;42m \033[0m", end='')

else:

print("\033[;41m \033[0m", end='')

if (x+1) % 100 == 0 :

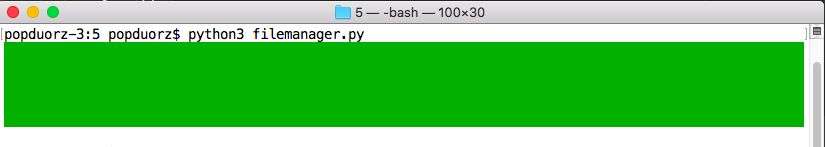
print("\n", end='')

def delete\_file(self, file):

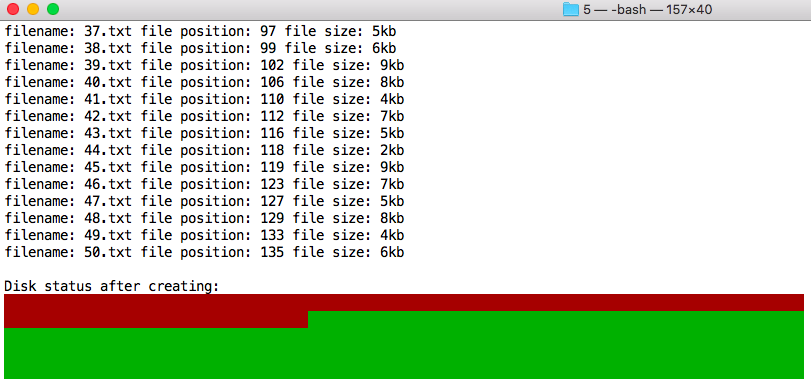
for x in range(file.start, file.get\_start() + file.get\_block\_length()):

self.bitmap[x] = 0}

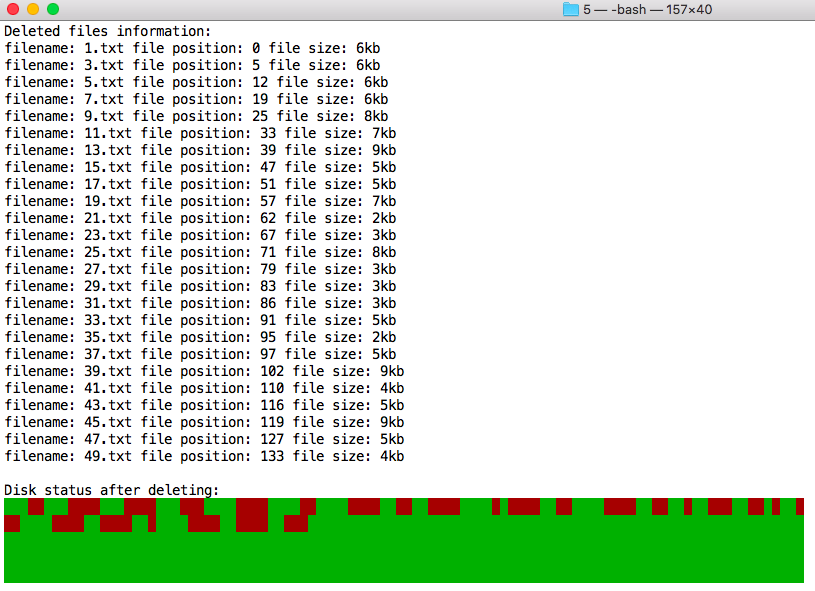
截图：



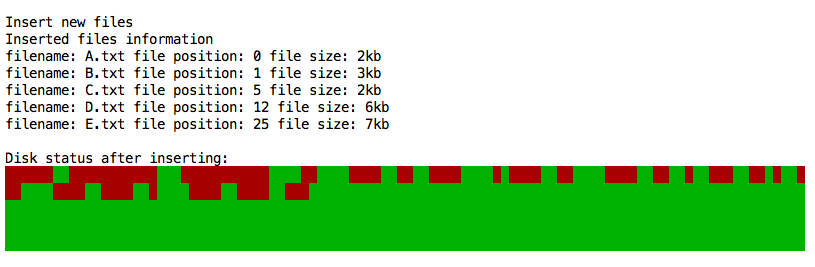
磁盘初始化



文件创建后的磁盘使用情况



文件删除后的磁盘使用情况



文件插入后的磁盘使用情况

**5.5 结果分析**

红色块表示磁盘被占用，绿色块表示磁盘可写。

第一张图全是绿色部分，随后创建了50个文件，文件的大小随机生成（由于数目太大，只有部分文件的信息给出），然后图二中显示出了50个文件创建后的磁盘占用情况。图三中显示了删除奇数个文件后的磁盘显示情况。图四显示了插入a,b,c,d,e五个文件后的磁盘状态。

在插入文件寻找空闲区时，从开头的磁盘块开始，只要找到足够大的磁盘块就将文件插入。可能会造成一些碎片。