|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 | 十 | 总评 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |  |

武汉大学计算机学院

课程实验(设计)报告

专业(班)： 2011级信息安全3版

学 号： 2010302530073

姓 名： 陈子杭

课程名称： 操作系统设计

2013年 4月1日

**实习一 处理器调度**

**一、实习内容**

选择一个调度算法，实现处理器调度。

**二、实习目的**

本实习模拟在单处理器环境下的处理器调度，加深了解处理器调度的工作。

**三、实习内容**

* 设计思路

使用优先队列（这里使用的是小根堆，即优先数越小，优先级越大）作为优先调度的原理。

模拟程序运行时，每个时间单位（即每次while循环）取队首进程运行一个单位，然后将其优先级加1，时间需求减1，如果此时该进程时间需求为零则直接抛弃，否则加回到队列中，由小根堆的性质保证队列项目的取出顺序符合调度顺序。

* 主要数据结构

ProcessControlBlock

+ proc\_name: PCB的唯一标识符

+ time\_demand: 进程运行所需时间

+ priority: 进程优先级

+ status: 进程状态，只有两种，默认为STATUS\_RUNNING，进程运行完毕后，应被修改为STATUS\_FINISHED

+ run\_once(): 模拟进程运行一次，调用此方法后，self.priority自加1，self.time\_demand自减1，并根据time\_demand的值，修改self.status

+ is\_finished(): 根据self.status判断进程是否完成

IdleProcess: ProcessControlBlock

作为一个占位符，用来启动主循环。其优先级恒为65535。如果调用该类的run\_once()则会打印idle is the ultimate process提示队列为空。

* 主要代码结构

9-40行为PCB类代码，44-50行为IdleProcess类代码，60-93为命令处理函数，98-$为主程序代码。

98行建立一个优先队列对象queue，并且将一个IdleProcess对象放入queue，110行为主循环，终止条件为队列为空（但是实际上是不会为空的，因为IdleProcess的剩余时间恒为'inf'），每次循环取出队首进程，调用它的run\_once方法，然后调用它的is\_finished来判断它是否完成，如果未完成（time\_demand大于0），则放回queue中，然后打印队列。

**四、实验平台**

win7 x64，python2.7 x64

五、调试过程

以下测试数据录入3个进程，参数分别为进程名，进程所需时间，进程优先级，然后运行10次并打印每次循环的队列。

> n a 3 1

> n b 2 4

> n c 5 2

> c 10

a run

putting a back into queue

-- queue -- name; t; p

-- process a; 2; 2

-- process c; 5; 2

-- process b; 2; 4

-- process Idle; inf; 65535

a run

putting a back into queue

-- queue -- name; t; p

-- process c; 5; 2

-- process a; 1; 3

-- process b; 2; 4

-- process Idle; inf; 65535

c run

putting c back into queue

-- queue -- name; t; p

-- process a; 1; 3

-- process c; 4; 3

-- process b; 2; 4

n-- process Idle; inf; 65535

a run

a finished

-- queue -- name; t; p

-- process c; 4; 3

-- process Idle; inf; 65535

-- process b; 2; 4

c run

putting c back into queue

-- queue -- name; t; p

-- process b; 2; 4

-- process Idle; inf; 65535

-- process c; 3; 4

b run

putting b back into queue

-- queue -- name; t; p

-- process c; 3; 4

-- process Idle; inf; 65535

-- process b; 1; 5

c run

putting c back into queue

-- queue -- name; t; p

-- process b; 1; 5

-- process Idle; inf; 65535

-- process c; 2; 5

b run

b finished

-- queue -- name; t; p

-- process c; 2; 5

-- process Idle; inf; 65535

c run

putting c back into queue

-- queue -- name; t; p

-- process c; 1; 6

-- process Idle; inf; 65535

c run

c finished

-- queue -- name; t; p

-- process Idle; inf; 65535

idle is the ultimate process

putting Idle back into queue

-- queue -- name; t; p

-- process Idle; inf; 65535

>

在纸上手工模拟的结果与程序输出相同。

六、总结

如果按照题目的要求，应该使用大根堆，但是python标准库只提供了小根堆，所以把题目的要求倒转过来了（即每次运行后优先级自增，而不是题目描述的自减），但是不影响实际的结果。

**实习二 主存空间的分配和回收**

**一、实习内容**

主存储器空间的分配和回收。

**二、实习目的**

通过本实习帮助理解在不同的存储管理方式下应怎样进行存储空间的分配和回收。

三、实习内容

* 设计思路

假定系统的主存被分成大小相等的64个块，用0/1对应空闲/占用，当装入一个作业时，根据作业的主存需求量查询空闲块数能否满足作业要求，若能满足，则顺序查找位示图并修改位示图和空闲块数，然后在作业的页表中登记这次分配。当回收时，修改位示图和空闲块数，并且能够检测无效的回收请求。

* 主要数据结构

PageTable 描述每个进程的页表的类

+ register(blocks): 登记分配给进程的块

+ release(\*pages): 反登记页表

+ display(): 打印页表

Memory 描述主存的类

+ size: 主存大小，单位为kB，默认为256

+ total\_block\_amount: 总块数，恒为64

+ available\_block\_amount: 可用块数，默认为总块数的数值

+ page\_size: 页面大小，即主存大小除以总块数

+ bit\_map: 位示图

+ allocate(size): 尝试分配与请求对应大小的块

+ release(\*block\_numbers): 释放块号对应的块，即修改位示图和剩余块数

* 主要代码结构

在PageTable中，页面号即self.\_l的索引号，self.\_l中索引号对应的值即页面对应的块号，若一个页面被释放，则对应的值被改为'n/a'，一个页面是否有效就是根据self.\_l对应的值是否为'n/a'来判断的。

Memory中，位示图并不是一个二维列表，而是一个64个元素的一维列表，分配与释放的代码实现与设计思路里的描述相同。79-125行是命令处理函数。

四、实验平台

win7 x64，python2.7 x64

五、调试过程

以下测试数据模拟3个进程的主存请求（分多次请求），并最后全部释放，以检查一致性。

> a a 50

13 blocks needed, 64 available, after allocation 51 available

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] allocated

> a b 90

23 blocks needed, 51 available, after allocation 28 available

[13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35] allocated

> a c 150

38 blocks needed, 28 available, after allocation -10 available

requested size exceeded available memory

None allocated

> a c 110

28 blocks needed, 28 available, after allocation 0 available

[36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63] allocated

> d

-- a -- page | block

0 | 0 5 | 5 10 | 10

1 | 1 6 | 6 11 | 11

2 | 2 7 | 7 12 | 12

3 | 3 8 | 8

4 | 4 9 | 9

-- b -- page | block

0 | 13 5 | 18 10 | 23 15 | 28 20 | 33

1 | 14 6 | 19 11 | 24 16 | 29 21 | 34

2 | 15 7 | 20 12 | 25 17 | 30 22 | 35

3 | 16 8 | 21 13 | 26 18 | 31

4 | 17 9 | 22 14 | 27 19 | 32

-- c -- page | block

0 | 36 5 | 41 10 | 46 15 | 51 20 | 56 25 | 61

1 | 37 6 | 42 11 | 47 16 | 52 21 | 57 26 | 62

2 | 38 7 | 43 12 | 48 17 | 53 22 | 58 27 | 63

3 | 39 8 | 44 13 | 49 18 | 54 23 | 59

4 | 40 9 | 45 14 | 50 19 | 55 24 | 60

> b

available blocks: 0

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

> r a 3 5 7 9 11 13

releasing block 3

releasing page 3

releasing block 5

releasing page 5

…

releasing block 11

releasing page 11

trying to release invalid page 13, aborting

> r c 1 3 5 7 9 11 13 17 19

releasing block 37

releasing page 1

releasing block 39

releasing page 3

…

releasing block 53

releasing page 17

releasing block 55

releasing page 19

> d a c

-- a -- page | block

0 | 0 8 | 8

1 | 1 10 | 10

2 | 2 12 | 12

4 | 4

6 | 6

-- c -- page | block

0 | 36 10 | 46 18 | 54 24 | 60

2 | 38 12 | 48 20 | 56 25 | 61

4 | 40 14 | 50 21 | 57 26 | 62

6 | 42 15 | 51 22 | 58 27 | 63

8 | 44 16 | 52 23 | 59

> b

available blocks: 14

[1, 1, 1, 0, 1, 0, 1, 0]

[1, 0, 1, 0, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 0, 1, 0]

[1, 0, 1, 0, 1, 0, 1, 0]

[1, 0, 1, 1, 1, 0, 1, 0]

[1, 1, 1, 1, 1, 1, 1, 1]

> a c 50

13 blocks needed, 14 available, after allocation 1 available

[3, 5, 7, 9, 11, 37, 39, 41, 43, 45, 47, 49, 53] allocated

> d c

-- c -- page | block

0 | 36 5 | 7 10 | 46 15 | 51 20 | 56 25 | 61 30 | 49

1 | 3 6 | 42 11 | 37 16 | 52 21 | 57 26 | 62 31 | 53

2 | 38 7 | 9 12 | 48 17 | 41 22 | 58 27 | 63

3 | 5 8 | 44 13 | 39 18 | 54 23 | 59 28 | 45

4 | 40 9 | 11 14 | 50 19 | 43 24 | 60 29 | 47

> b

available blocks: 1

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 0]

[1, 1, 1, 1, 1, 1, 1, 1]

> r b

releasing block 13

releasing page 0

releasing block 14

releasing page 1

…

releasing block 34

releasing page 21

releasing block 35

releasing page 22

> d b

-- b -- page | block

empty

> b

available blocks: 24

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 0]

[1, 1, 1, 1, 1, 1, 1, 1]

> a c 80

20 blocks needed, 24 available, after allocation 4 available

[13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32] allocated

> d c

-- c -- page | block

0 | 36 5 | 7 10 | 46 15 | 51 20 | 56 25 | 61 30 | 49 35 | 16 40 | 21 45 | 26 50 | 31

1 | 3 6 | 42 11 | 37 16 | 52 21 | 57 26 | 62 31 | 53 36 | 17 41 | 22 46 | 27 51 | 32

2 | 38 7 | 9 12 | 48 17 | 41 22 | 58 27 | 63 32 | 13 37 | 18 42 | 23 47 | 28

3 | 5 8 | 44 13 | 39 18 | 54 23 | 59 28 | 45 33 | 14 38 | 19 43 | 24 48 | 29

4 | 40 9 | 11 14 | 50 19 | 43 24 | 60 29 | 47 34 | 15 39 | 20 44 | 25 49 | 30

> b

available blocks: 4

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 0, 0, 0, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 0]

[1, 1, 1, 1, 1, 1, 1, 1]

> a a 16

4 blocks needed, 4 available, after allocation 0 available

[33, 34, 35, 55] allocated

> d a

-- a -- page | block

0 | 0 5 | 55 11 | 33

1 | 1 6 | 6 12 | 12

2 | 2 8 | 8

3 | 35 9 | 34

4 | 4 10 | 10 `

> b

available blocks: 0

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1, 1, 1]

> r a

releasing block 0

releasing page 0

releasing block 1

releasing page 1

…

releasing block 33

releasing page 11

releasing block 12

releasing page 12

> r b

> r c

releasing block 36

releasing page 0

releasing block 3

releasing page 1

…

releasing block 31

releasing page 50

releasing block 32

releasing page 51

> d

-- a -- page | block

empty

-- b -- page | block

empty

-- c -- page | block

empty

> b

available blocks: 64

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

>

在纸上手工模拟的结果与程序输出相同。

六、总结

要仔细弄清楚每个方法的边界条件，特别是PageTable.release方法和Memory.release方法里，特别需要注意判断一些边界条件，比如页面号是否超过了页表长度，页面是不是对应的有一个物理块等，否则会出现意料外的情况。

**实习三 磁盘存储空间的分配和回收**

**一、实习内容**

模拟磁盘空闲空间的表示方法，以及模拟实现磁盘空间的分配和回收。

**二、实习目的**

磁盘初始化时把磁盘存储空间分成许多块（扇区），这些空间可以被多个用户共享。用户作业在执行期间常常要在磁盘上建立文件或把已经建立在磁盘上的文件删去，这就涉及到磁盘存储空间的分配和回收。一个文件存放到磁盘上，可以组织成顺序文件（连续文件）、链接文件（串联文件）、索引文件等，因此，磁盘存储空间的分配有两种方式，一种是分配连续的存储空间，另一种是可以分配不连续的存储空间。怎样有效地管理磁盘存储空间是操作系统应解决的一个重要问题，通过本实习使学生掌握磁盘存储空间的分配和回收算法。

三、实习内容

* 设计思路

维护两个表，一个是空闲区表来记录磁盘存储空间中尚未占用的部分，另一个是文件目录表来记录文件名、起始块和占用块数。当要建立顺序文件时在空闲区表中找到一个适合的条目，然后根据所需的块数修改条目（如果有必要可以删掉这个条目）。当要删除一个文件时，需要考虑空闲块的合并情况。

* 主要数据结构

HardDrive

+ allocate(filename, blocks\_required): 尝试分配文件并修改文件目录表

+ recycle(filename): 根据文件名删除文件

ATItem 文件目录表表目

+ start: 起始块

+ n: 可用块数

* 主要代码结构

代码由一个类HardDrive和一个函数dispatch\_command组成，其中HardDrive由ATItem和HardDrive的各方法组成。

其中allocate方法负责按请求的块数分配磁盘空间并在文件表中登记，当被调用后，allocate方法遍历空闲区表，如果能找到符合块数要求的空闲区，则修改表目，如果整个区被分配则删除这个表目（程序中实现的方法是过滤掉空表目），然后在文件表中登记起始块号和占用的块数，最后返回起始块号的物理地址（即扇区号，磁道号和柱面号）；如果遍历完空闲区表仍未能找到适合的起始块号，那么返回错误。

recycle方法负责根据文件名回收磁盘空间，当被调用后，首先检测文件名是否在文件表中，如果在，则取出对应文件的起始块号和占用块数，并用这两个参数新建一个表目对象，插入到空闲块表的适当位置中（即保证当前表目的起始块号在前一个和后一个表目的起始块号的中间），然后合并表目（如果有可以合并的）。合并的过程是，将空闲区表的一个一个压入一个栈中，并且每次压栈时，都与栈顶元素比较（如果有），如果处于栈顶的表目的起始块号加上块数等于将要压栈的表目的起始块号，那么这两个表目合并，否则直接压栈。

四、实验平台

win7 x64，python2.7 x64

五、调试过程

以下测试数据模拟了6个文件的申请和回收，并检查了磁盘空间的一致性。

> a a.txt 2000

allocated (track: 0, cylinder: 0, sector: 0)

> a b.exe 50000

allocation failed: not enough space

> a b.exe 3500

allocated (track: 13, cylinder: 16, sector: 2)

> a c.bin 6000

allocated (track: 16, cylinder: 45, sector: 4)

> a d.py 200

allocated (track: 16, cylinder: 95, sector: 4)

> a e.7z 5600

allocated (track: 10, cylinder: 97, sector: 0)

> d

start | blocks

17300 | 6700

> f

filename | start | blocks

a.txt | 0 | 2000

b.exe | 2000 | 3500

c.bin | 5500 | 6000

d.py | 11500 | 200

e.7z | 11700 | 5600

> r b.exe

file recycle succeeded

> d

start | blocks

2000 | 3500

17300 | 6700

> f

filename | start | blocks

a.txt | 0 | 2000

c.bin | 5500 | 6000

d.py | 11500 | 200

e.7z | 11700 | 5600

> r d.p

file not found

> r d.py

file recycle succeeded

> d

start | blocks

2000 | 3500

11500 | 200

17300 | 6700

> f

filename | start | blocks

a.txt | 0 | 2000

c.bin | 5500 | 6000

e.7z | 11700 | 5600

> a f.iso 3000

allocated (track: 13, cylinder: 16, sector: 2)

> d

start | blocks

5000 | 500

11500 | 200

17300 | 6700

> f

filename | start | blocks

a.txt | 0 | 2000

c.bin | 5500 | 6000

e.7z | 11700 | 5600

f.iso | 2000 | 3000

> r e.7z

file recycle succeeded

> d

start | blocks

5000 | 500

11500 | 12500

> f

filename | start | blocks

a.txt | 0 | 2000

c.bin | 5500 | 6000

f.iso | 2000 | 3000

> r a.txt

file recycle succeeded

> d

start | blocks

0 | 2000

5000 | 500

11500 | 12500

> f

filename | start | blocks

c.bin | 5500 | 6000

f.iso | 2000 | 3000

> r c.bin

file recycle succeeded

> d

start | blocks

0 | 2000

5000 | 19000

> f

filename | start | blocks

f.iso | 2000 | 3000

> r f.iso

file recycle succeeded

> d

start | blocks

0 | 24000

> f

filename | start | blocks

>

在纸上手工模拟的结果与程序输出相同。

六、总结

合并的时候，需要把回收的表目放到空闲区表的合适位置，但是这在python中并不是特别好实现。最后采取了比较繁琐的步骤。