实验二:实现动态分区分配模拟程序编程

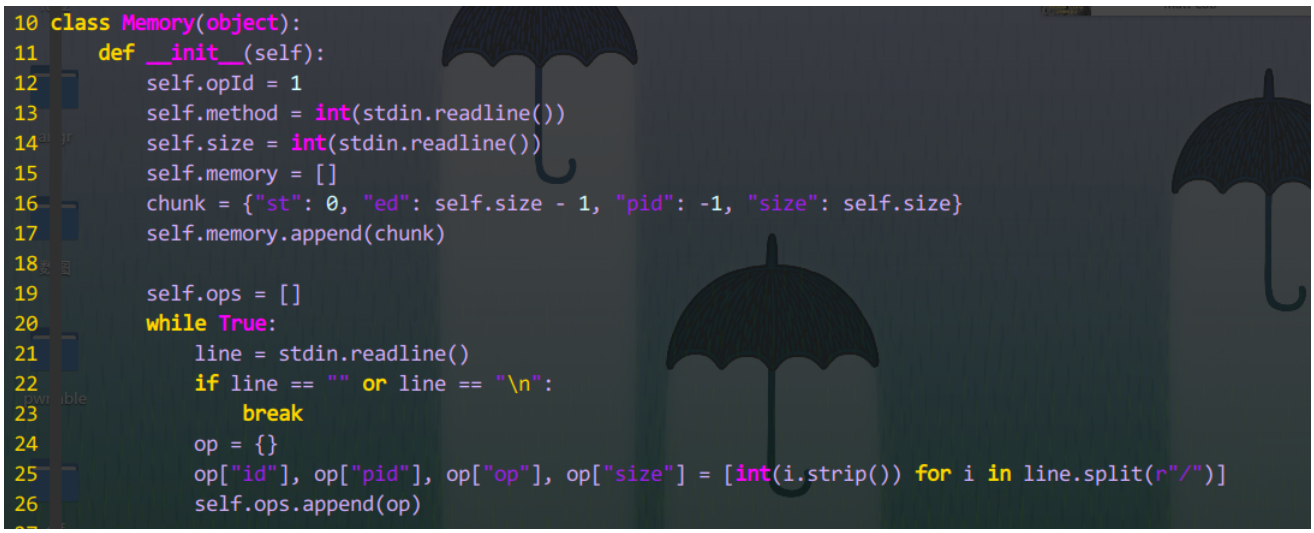
1120152035 王奥博

> 语言: python2.7
>
> OS: deepin-linux 15.5 amd64

> 本次实验只用python代码实现了编程,没有再尝试C++版的编程

数据结构及符号分析

和第一次实验类似,对整个程序定义了一个类**Memory**整体处理,在构造函数中进行了对输入进行处理如下:



```
10 class Memory(object):
11     def __init__(self):
12         self.opId = 1
13         self.method = int(stdin.readline())
14         self.size = int(stdin.readline())
15         self.memory = []
16         chunk = {"st": 0, "ed": self.size - 1, "pid": -1, "size": self.size}
17         self.memory.append(chunk)
18 
19         self.ops = []
20         while True:
21             line = stdin.readline()
22             if line == "" or line == "\n":
23                 break
24             op = {}
25             op["id"], op["pid"], op["op"], op["size"] = [int(i.strip()) for i in line.split(r"/")]
26             self.ops.append(op)
```

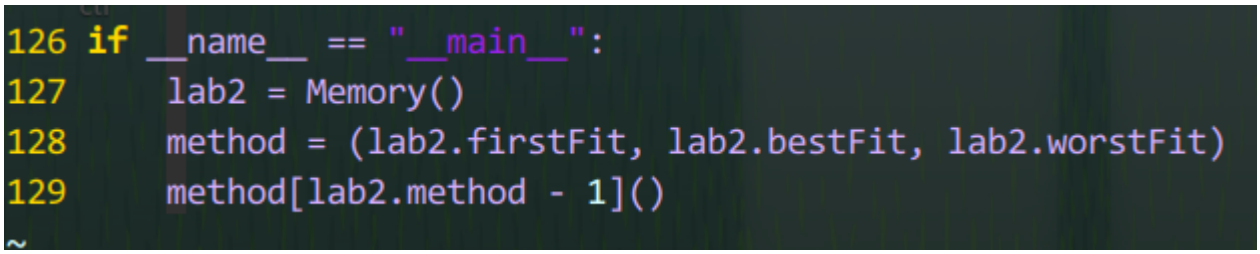
19~26行实现了对输入的处理.

其中对于每个进程,定义了一个字典进行存储,如上图25行,保存了进程的**序号,进程号,释放/申请,释放/申请的内存大小**,然后用一个列表***self.ops***保存了所有的进程.

对于内存中不同的块,也定义了一个字典进行存储,如上图16行,保存了内存块的**起始和终止地址,占用该块的进程号,该块的大小**.值得说明的是,对于空闲的内存块,**为了方便编码,设定了所有的空闲内存块的占用号为-1**.

调度算法的处理流程

同第一次实验相似,同样定义了一个函数元组存储每一种调度算法以简化编码,如下图:



```
126 if __name__ == "__main__":
127     lab2 = Memory()
128     method = (lab2.firstFit, lab2.bestFit, lab2.worstFit)
129     method[lab2.method - 1]()
```

同时定义了一个输出的函数进行输出每步的内存分配情况:

```
31  def getAns(self):
32      ans = str(self.opId)
33      self.memory = sorted(self.memory, key = lambda x: x["st"])
34      for i in self.memory:
35          if i["pid"] == -1:
36              ans += (r"/" + str(i["st"]) + "-" + str(i["ed"]) + ".0")
37          else:
38              ans += (r"/" + str(i["st"]) + "-" + str(i["ed"]) + ".1." + str(i["pid"]))
39
40      self.opId += 1
41      print ans
42
```

35行为判断该块是否被占用,对正在被占用和未被占用的内存块采取36行和38行不同的操作.

对**首次适应(firstFit),最佳适应(bestFit),最坏适应(worstFit)**三种算法而言,算法流程都是相似的,申请内存和释放内存的操作相同,仅有选取内存块的规则不同,因此**申请内存和释放内存单独定义了函数**.

申请内存:

申请内存的核心代码如下:

```
72          while True:
73              try:
74                  if self.memory[idx]["pid"] == -1 and self.memory[idx]["size"] >= i["size"]:
75                      #分割空闲块
76                      c1 = deepcopy(self.memory[idx])
77                      c1["pid"] = i["pid"]
78                      c1["ed"] = i["size"] + c1["st"] - 1
79                      c1["size"] = i["size"]
80
81                      #空闲chunk存在剩余
82                      if c1["ed"] < self.memory[idx]["ed"]:
83                          c2 = {}
84                          c2["st"] = c1["ed"] + 1
85                          c2["ed"] = self.memory[idx]["ed"]
86                          c2["pid"] = -1
87                          c2["size"] = c2["ed"] - c2["st"] + 1
88                          self.memory = self.memory[:idx] + [c1, c2] + self.memory[idx + 1:]
89                          self.getAns()
90                          break
91
92                      else:
93                          self.memory = self.memory[:idx] + [c1] + self.memory[idx + 1:]
94                          self.getAns()
95                          break
96                  idx += 1
97              except:
98                  self.getAns()
99                  break
```

对于一个新的操作,只需在排好序的内存块中寻找**第一个没被占用并且大小合适**的内存块进行分配即可,第74行即进行了此步判断.

找到满足条件的空闲内存块后,进行两步操作:

1. **分割出需占用的大小**(74~79行分割出一块空闲块进行分配)
2. **若该块还有剩余,则对剩余的内存块进行操作**(82~90行完成了对空闲块的操作)

此外,97~99行完成了分配失败时的输出信息.

释放内存:

释放内存相比申请内存更为简单,实现了两步操作:

1. **首先找到被指定进程占用的内存块,释放该块**
2. **对于相邻的空闲内存块进行合并**

具体实现代码如下:

```
43    def free(self, pid):
44        for c in self.memory:
45            if c["pid"] == pid:
46                c["pid"] = -1
47                break
48
49        idx = 0
50        while True:#合并相邻空闲chunk
51            try:
52                if self.memory[idx]["pid"] == self.memory[idx + 1]["pid"] == -1:
53                    self.memory[idx]["ed"] = self.memory[idx + 1]["ed"]
54                    self.memory[idx]["size"] = self.memory[idx]["ed"] - self.memory[idx]["st"] + 1
55                    del self.memory[idx + 1]
56                    idx -= 1#多次合并
57
58                idx += 1
59            except:
60                self.getAns()
61                break
```

其中,56行保证了对多个相邻的空闲内存块进行合并

算法调度:

实现了申请和释放内存后,对firstFit,bestFit,worstFit三种算法只需实现不同的排序即可.

```
65        if method == "firstFit":
66            self.memory = sorted(self.memory, key = lambda x: x["st"])
67        elif method == "bestFit":
68            self.memory = sorted(self.memory, key = lambda x: x["size"])
69        elif method == "worstFit":
70            self.memory = sorted(self.memory, key = lambda x: x["size"], reverse = True)
71
```

如上,**firstFit对开始地址进行排序,bestFit对申请/释放内存进行排序,worstFit对申请/释放内存进行逆序排序**

对不同算法调度的方式完全相同,如下图:

```
101    def firstFit(self):
102        #  pdb.set_trace()
103        for i in self.ops:
104            if i["op"] == 1:#申请
105                self.malloc("firstFit", i)
106            else:#释放
107                #  pdb.set_trace()
108                self.free(i["pid"])
109
110    def bestFit(self):
111        #  pass
112        for i in self.ops:
113            if i["op"] == 1:
114                self.malloc("bestFit", i)
115            else:
116                self.free(i["pid"])
117
118    def worstFit(self):
119        #  pass
120        for i in self.ops:
121            if i["op"] == 1:
122                self.malloc("worstFit", i)
123            else:
124                self.free(i["pid"])
```

完整代码：

> 代码及测试用例已上传到https://github.com/M4xW4n9/personal_repository/blob/master/OS/lab2/
>
> 使用
>
> wget https://raw.githubusercontent.com/M4xW4n9/personal_repository/master/OS/lab2/lab2.py
>
> 即可下载

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__Auther__ = 'M4x'

from sys import stdin
from copy import deepcopy
from pprint import pprint
import pdb

class Memory(object):
    def __init__(self):
        self.opId = 1
        self.method = int(stdin.readline())
        self.size = int(stdin.readline())
        self.memory = []
        chunk = {"st": 0, "ed": self.size - 1, "pid": -1, "size": self.size}
        self.memory.append(chunk)

        self.ops = []
        while True:
            line = stdin.readline()
            if line == "" or line == "\n":
                break
            op = {}
            op["id"], op["pid"], op["op"], op["size"] = [int(i.strip()) for i in
line.split(r"/")]
            self.ops.append(op)

        #  pprint(self.memory)
        #  pprint(self.ops)

    def getAns(self):
        ans = str(self.opId)
        self.memory = sorted(self.memory, key = lambda x: x["st"])
        for i in self.memory:
            if i["pid"] == -1:
                ans += (r"/" + str(i["st"]) + "-" + str(i["ed"]) + ".0")
            else:
                ans += (r"/" + str(i["st"]) + "-" + str(i["ed"]) + ".1." + str(i["pid"]))

        self.opId += 1
        print ans

    def free(self, pid):
        for c in self.memory:
            if c["pid"] == pid:
                c["pid"] = -1
                break

        idx = 0
        while True:#合并相邻空闲chunk
            try:
                if self.memory[idx]["pid"] == self.memory[idx + 1]["pid"] == -1:
```

```
                    self.memory[idx]["ed"] = self.memory[idx + 1]["ed"]
                    self.memory[idx]["size"] = self.memory[idx]["ed"] - self.memory[idx]
["st"] + 1
                    del self.memory[idx + 1]
                    idx -= 1#多次合并

                idx += 1
            except:
                self.getAns()
                break

    def malloc(self, method, i):
        idx = 0
        if method == "firstFit":
            self.memory = sorted(self.memory, key = lambda x: x["st"])
        elif method == "bestFit":
            self.memory = sorted(self.memory, key = lambda x: x["size"])
        elif method == "worstFit":
            self.memory = sorted(self.memory, key = lambda x: x["size"], reverse = True)

        while True:
            try:
                if self.memory[idx]["pid"] == -1 and self.memory[idx]["size"] >= i["size"]:
                    #分割空闲块
                    c1 = deepcopy(self.memory[idx])
                    c1["pid"] = i["pid"]
                    c1["ed"] = i["size"] + c1["st"] - 1
                    c1["size"] = i["size"]

                    #空闲chunk存在剩余
                    if c1["ed"] < self.memory[idx]["ed"]:
                        c2 = {}
                        c2["st"] = c1["ed"] + 1
                        c2["ed"] = self.memory[idx]["ed"]
                        c2["pid"] = -1
                        c2["size"] = c2["ed"] - c2["st"] + 1
                        self.memory = self.memory[:idx] + [c1, c2] + self.memory[idx + 1:]
                        self.getAns()
                        break

                    else:
                        self.memory = self.memory[:idx] + [c1] + self.memory[idx + 1:]
                        self.getAns()
                        break
                idx += 1
            except:
                self.getAns()
                break

    def firstFit(self):
        #  pdb.set_trace()
        for i in self.ops:

            if i["op"] == 1:#申请
```

```
                self.malloc("firstFit", i)
            else:#释放
                #  pdb.set_trace()
                self.free(i["pid"])

    def bestFit(self):
        #  pass
        for i in self.ops:
            if i["op"] == 1:
                self.malloc("bestFit", i)
            else:
                self.free(i["pid"])

    def worstFit(self):
        #  pass
        for i in self.ops:
            if i["op"] == 1:
                self.malloc("worstFit", i)
            else:
                self.free(i["pid"])

if __name__ == "__main__":
    lab2 = Memory()
    method = (lab2.firstFit, lab2.bestFit, lab2.worstFit)
    method[lab2.method - 1]()
```