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 11 月 18 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： **指导老师签字：**　　年　　月　　日				

实验报告正文

实验六　动态分区与页面置换

一、　实验目的

1. 掌握动态分区分配方式使用的数据结构和分配算法；
2. 进一步加深对动态分区分配管理方式及其实现过程的理解。

二、　实验内容

1.　任务描述

补充 C 语言程序，模拟实现首次/最佳/最坏适应算法**(样例代码使用的是首次适应算法、感兴趣同学可自行选择其它两种算法实现)**的内存块分配和回收，要求每次分配和回收后显示出空闲分区和已分配分区的情况。假设初始状态下，可用的内存空间为 640KB。

实现 LRU 算法，页容量自定，输入一段页序列，计算不同页容量下的页命中率；

拓展：实现 FIFO 算法，并且分析 Belady 现象（选做）。

2.　实验方案

数据结构设计：

- 已分配分区表、空闲分区表

分配算法设计：

- 首次适应、最佳适应、最差适应分配算法

• 根据分配算法决定空闲分区表的排序

回收算法设计：

• 考虑回收区所属的四种情况，有上空分区无下空分区、无上空分区有下空分区、上下分区都为空分区，上下都无空分区，根据情况来决定回收区的处理。

3. 实验说明

动态分区分配算法：

- 首次适配(first fit)算法

从前端开始扫描内存，直到找到一个足够大的空闲区

- 邻近适配(next fit)算法

从上次分配结束的地方开始扫描内存，直到找到一个足够大的空闲区

- 最佳适配(best fit)算法

扫描整个内存，找出一个足够大的最小的空闲区

页置换算法：

- 最优算法（OPT，OPTimal）
- 最近最少使用算法（LRU，Least Recently Used）
- 先进先出算法（FIFO，First-In-First-Out）
- 时钟算法（Clock）

三、 实验记录

1. 实施步骤

实现首先适应回收算法：

```
void firstfit(RECT *head,RECT *heada,RECT *back1)
{
  RECT *before,*after,*back2;
  int insert,del;
  back2=malloc(sizeof(RECT));
  back2->address=back1->address;
  back2->size=back1->size;
  back2->next=back1->next;
  before=head;
  after=head->next;
  insert=0;
  while(!insert)      //将回收区插入空闲区表
  {

if((after==NULL)||((back1->address<=after->address)&&(back1->addr
ess>=before->address)))
    {
      // TODO 1: 补充程序, 将回收区插入到空闲区表合适位置;
        before->next=back1;
        back1->next=after;
        insert=1;

    }
    else
    {
      before=before->next;
      after=after->next;
    }
  }

  if(back1->address==before->address+before->size) //与上一块合并
  {
    //TODO 2: 补充程序, 合并空闲区块
        before->size=before->size+back1->size;
        before->next=back1->next;
        free(back1);
        back1=before;
  }

  //TODO 3: 补充程序, 合并空闲区块, Note: 记得 free 被合并的分区
 if(after!=NULL&&(after->address==back1->address+back1->size))
  {
    back1->size=back1->size+after->size;
    back1->next=after->next;
```

```
    free(after);
  }

  if(head->size<back1->size) //修改最大块值和最大块个数
  {
    head->size=back1->size;
    maxblocknum=1;
  }
  else
    if(head->size==back1->size) maxblocknum++;

  //修改已分配分区表，删除相应节点
  before=heada;
  after=heada->next;
  del=0;
  while(!del&&after!=NULL)  //将回收区从已分配分区表中删除
  {
    //TODO 4: 补充 if 中回收已分配分区表的删除条件
    if(after->size==back2->size)
    {
      before->next=after->next;
      free(after);
      del=1;
    }
    else
    {
      before=before->next;
      after=after->next;
    }
  }
  heada->size--;
}
```

实现最佳适应回收算法：

```
void bestfit(RECT *head,RECT *heada,RECT *back1)
{
  RECT *before,*after,*back2;
  int insert,del;
  back2=malloc(sizeof(RECT));
  back2->address=back1->address;
  back2->size=back1->size;
  back2->next=back1->next;
  before=head;
  after=head->next;
```

```
insert=0;
if(head->next==NULL) /*如果可利用区表为空*/
 {
    head->size=back1->size;
    head->next=back1;
    maxblocknum++;
    back1->next=NULL;
 }
 else
 {
    while(after!=NULL) /*与上一块合并*/
    if(back1->address==after->size+after->address)
    {
        before->next=after->next;
        back->size=after->size+back1->size;
        free(after);
        after=NULL;
    }
    else
    {
        after=after->next;
        before=before->next;
    }
    before=head;
    after=head->next;
    while(after!=NULL)
    if(after->address==back1->size+back1->address) /*与下一块合并*/
    {
        back1->size=back1->size+after->size;
        before->next=after->next;
        free(after);
        after=NULL;
    }
    else
    {
        before=before->next;
        after=after->next;
    }
    before=head;/*将回收结点插入到合适的位置*/
    after=head->next;
    do{
        if(after==NULL||(after->size>back1->size))
        {
           before->next=back1;
```

```
            back1->next=after;
            insert=1;
          }
          else
          {
            before=before->next;
            after=after->next;
          }
      }while(!insert);
      if(head->size<back1->size) /*修改最大块值和最大块数*/
      {
         head->size=back1->size;
         maxblocknum++;
      }
      else
         if(head->size==back1->size)
           maxblocknum++;
   }
    before=heada;
    after=heada->next;
    del=0;
    while(!del&&after!=NULL)  /*将回收区从已分配分区表中删除*/
    {
      if(after->size==back2->size)
      {
        before->next=after->next;
        free(after);
        del=1;
      }
      else
      {
        before=before->next;
        after=after->next;
      }
    }
}
```

实现 LRU 算法：

```
void LruHitProcess() // if the replacement policy is LRU,and hit
{
    if(t_assoc == full_associative)
    {
        for(i=0; i<i_num_line; i++)
        {
```

```
            if(LRU_priority[i]<LRU_priority[current_line] &&
cache_item[current_line][30]==true)
            {
                LRU_priority[i]++; // 如果该行比正在访问的行计数器值小，并
且该行中 hit 为 true
            }
        }

        LRU_priority[current_line] = 0;
    }
    else if(t_assoc == set_associative)
    {
        for(i=(current_set*i_cache_set);
i<((current_set+1)*i_cache_set); i++)
        {
            if(LRU_priority[i]<LRU_priority[current_line] &&
cache_item[current_line][30]==true)
            {
                LRU_priority[i]++; // 如果该行比正在访问的行计数器值小，并
且该行中 hit 为 true
            }
        }

        LRU_priority[current_line] = 0;
    }
}

void LruUnhitSpace() // if the replacement policy is LRU,and not
hit,but there has a spaceline
{
    if(t_assoc == full_associative)
    {
        for(i=0; i<i_num_line; i++)
        {
            if(cache_item[current_line][30]==true)
            {
                LRU_priority[i]++; // 如果该行该行中 hit 为 true
            }
        }

        LRU_priority[current_line] = 0;
    }
    else if(t_assoc == set_associative)
    {
```

```
        for(i=(current_set*i_cache_set);
i<((current_set+1)*i_cache_set); i++)
        {
            if(cache_item[current_line][30]==true)
            {
                LRU_priority[i]++; // 如果该行该行中 hit 为 true
            }
        }

        LRU_priority[current_line] = 0;
    }
}

void LruUnhitUnspace()
{
    if(t_assoc == full_associative)
    {
        temp = LRU_priority[0];

        for(i=0; i<i_num_line; i++)
        {
            if(LRU_priority[i]>=temp)
            {
                temp = LRU_priority[i];
                j = i;
            }
        }

        current_line = j;
    }

    if(t_assoc == set_associative)
    {
        temp = LRU_priority[current_set*i_cache_set];

        for(i=(current_set*i_cache_set);
i<((current_set+1)*i_cache_set); i++)
        {
            if(LRU_priority[i]>=temp)
            {
                temp = LRU_priority[i];
                j = i;
            }
        }
```

```
        current_line = j;
    }
}
```

2. 实验记录

首先适应回收算法：

```
(base) ➜ Enderfga ./mem
index****address****end*****size****
-------------------------------------
 0         0        639      640
-------------------------------------
Enter the allocation way (best,first or next (b/f/n))
f
Enter the allocate or reclaim (a/r),or press other key to exit.
a
Input application:
130
Allocation Success! ADDRESS=  510

*********Unallocated Table*************
index****address****end*****size****
-------------------------------------
 0         0        509      510
-------------------------------------

*********Allocated Table***  **********
index****address****end*****size****
-------------------------------------
 0         510      639      130
-------------------------------------
```

```
Enter the allocate or reclaim (a/r),or press other key to exit.
r
Input address and Size:
360 60

*********Unallocated Table*************
index****address****end*****size****
-------------------------------------
 0         0        349      350
-------------------------------------
-------------------------------------
 1         360      419      60
-------------------------------------

*********Allocated Table***  **********
index****address****end*****size****
-------------------------------------
 0         510      639      130
-------------------------------------
-------------------------------------
 1         350      449      100
-------------------------------------
```

```
*********Unallocated Table*************
index****address****end*****size****
-------------------------------------
 0         0        509      510
-------------------------------------

*********Allocated Table***  **********
index****address****end*****size****
-------------------------------------
 0         510      639      130
-------------------------------------
Enter the allocate or reclaim (a/r),or press other key to exit.
Input address and Size:

*********Unallocated Table*************
index****address****end*****size****
-------------------------------------
 0         0        639      640
-------------------------------------

*********Allocated Table***  **********
NO part for print!
Enter the allocate or reclaim (a/r),or press other key to exit.
q
```

最佳适应回收算法：

```
(base) ➜ Enderfga ./mem
index****address****end*****size****
------------------------------------
 0         0        639      640
------------------------------------
Enter the allocation way (best or first (b/f))
b
Enter the allocate or reclaim (a/r),or press other key to exit.
a
Input application:
130
Allocation Success! ADDRESS=  510

*********Unallocated Table************
index****address****end*****size****
------------------------------------
 0         0        509      510
------------------------------------

*********Allocated Table***  **********
index****address****end*****size****
------------------------------------
 0         510      639      130
------------------------------------
Enter the allocate or reclaim (a/r),or press other key to exit.
```

```
Enter the allocate or reclaim (a/r),or press other key to exit.
r 450 60
Input address and Size:

*********Unallocated Table************
index****address****end*****size****
------------------------------------
 0         450      509      60
-------------------------------------
------------------------------------
 1         0        349      350
-------------------------------------

*********Allocated Table***  **********
index****address****end*****size****
------------------------------------
 0         510      639      130
-------------------------------------
------------------------------------
 1         350      449      100
-------------------------------------
Enter the allocate or reclaim (a/r),or press other key to exit.
```

LRU 算法：

```
Cache Size:1KB
Cacheline Size:32B
Way of Associativity:set_associative
Way of Replacement:LRU
Way of Write:write_back

Number of cache access:100
Number of cache load:90
Number of cache store:10

Average cache hit rate:81%
Cache hit rate for loads:83.3333%
Cache hit rate for stores:60%
```

3. 实验结果

首次适应和最佳适应算法都是用于动态分区的，管理计算机系统内存分配的方法。动态分区分配算法通过将内存划分为若干个固定大小的分区，实现了内存的高效分配。

首次适应和最佳适应都是用于选择分配给需要内存的新进程的分区的方法。首次适应算法只是选择第一个足够容纳该进程的分区。这种方法简单易行，但并不总是得到最有效的内存利用。

相比之下，最佳适应算法选择最小的足够容纳该进程的分区。这种方法的实现更复杂，但它可以通过最小化每个分区中未使用空间的数量来实现更有效的内存利用。

首次适应算法的优点之一是它简单易行。在速度至关重要的情况下，如实时系统中需要快速分配内存的进程，它可能会很有用。

首次适应算法的另一个优点是它可能不容易发生碎片化，这是在动态分区分配算法中可能发生的问题，即在已分配的分区之间留下较小的未使用内思间隙。这在某些情况下可能更有效。

首次适应算法的缺点之一是它可能导致内存利用率低下，在某些分区中留下更大的未使用空间。这可能会导致内存浪费增加，这在内存资源有限的系统中可能是个问题。

相比之下，最佳适应算法可以通过最小化每个分区中未使用空间的数量来实现更有效的内存利用。在内存有限的情况下，它可以有助于最大化可用于分配给进程的内存。

但是，最佳适应算法的实现更复杂，在速度至关重要的情况下可能不太适用。它也更容易发生碎片化，因为寻找最小的足够容纳该进程的分区可能会导致更频繁的内存分配和取消分配。

总的来说，首次适应和最佳适应算法都旨在高效地为动态分区分配算法中的进程分配内存。但是，它们的方法不同，首次适应更简单、更快，但效率可能较低，而最佳适应更复杂、更慢，但效率可能更高。选择哪种算法将取决于所使用系统的特定要求和约束。

下面是一个用Python实现FIFO算法的例子，以及对其工作原理的解释：

```
# Create a queue to store the pages in memory
queue = []

# Function to implement the FIFO algorithm
def FIFO(pages, num_frames):
  # Keep track of the number of page faults
  page_faults = 0

  # Iterate over the pages to be accessed
  for i in range(len(pages)):
    # Check if the current page is in memory
    if pages[i] not in queue:
      # If the current page is not in memory, increment the page fault
count
      page_faults += 1
```

```
        # If there is space in memory, add the page to the end of the
queue
        if len(queue) < num_frames:
            queue.append(pages[i])

        # If there is no space in memory, replace the page at the front
of the queue with the current page
        else:
            queue.pop(0)
            queue.append(pages[i])

    # Return the number of page faults
    return page_faults

# Example usage of the FIFO algorithm
pages = [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
num_frames = 3
print(FIFO(pages, num_frames)) # Output: 9
```

FIFO 算法是在 FIFO 函数中实现的，该函数接受一个要访问的页面列表和内存中的帧数作为输入。该函数对页面列表进行迭代，并检查每个页面是否在内存中。如果该页不在内存中，该函数会增加页面故障计数。

如果内存中有空间，则该页被添加到队列的末端。如果内存中没有空间，队列前面的页面就被替换成当前的页面。这就是先进先出算法的关键思想--在内存中存在时间最长的页面总是被新的页面取代。

该函数返回在页面访问过程中发生的页面故障的数量。

至于 Belady 现象，如果要访问的页面遵循某种模式，在使用先进先出算法时就会发生。例如，如果页面按照 A-B-C-D-E-F-D-E-F 的顺序访问，那么使用 FIFO 算法将导致 3 个页面故障，但使用不同的算法，如 LRU（最近使用最少的）算法，只会导致 2 个页面故障。这是因为 FIFO 算法总是替换在内存中存在时间最长的页面，不管它在未来是否会被再次访问，而 LRU 算法则是替换在最长时间内没有被访问的页面。在这种情况下，FIFO 算法替换了不再

被访问的 A 页，而 LRU 算法则替换了不再被访问的 C 页。

四、 总结与讨论

1. 首次适应和最佳适应算法都是用于动态分区分配的算法，它们的目的是选择分配给新进程的分区。首次适应算法只是选择第一个足够容纳该进程的分区，这种方法简单易行，但并不总是得到最有效的内存利用。相比之下，最佳适应算法选择最小的足够容纳该进程的分区，这种方法的实现更复杂，但它可以通过最小化每个分区中未使用空间的数量来实现更有效的内存利用。
2. FIFO 算法的一个改进方法是使用 LRU 算法（Least Recently Used，最近最少使用）。LRU 算法与 FIFO 算法类似，但它更智能，它会替换那些不会再被访问到的页面。这样可以有效避免 Belady 现象，提高系统性能。
3. 此外，我们还可以尝试使用其他算法来进一步提高性能。例如，我们可以使用基于模拟的方法来预测页面访问次数，并根据预测结果来决定是否替换页面。这种方法的实现可能更复杂，但它可以通过精确预测页面访问次数来提高内存利用率。
4. 总的来说，首次适应和最佳适应算法、FIFO 和 LRU 算法都是用于解决动态分区分配和页面置换问题的常用算法。它们各有优缺点，可以根据具体情况选择使用。此外，我们还可以尝试改进和创新，开发新的算法来解决这些问题。