苏州大学实验报告

院、系	计算机学院	年级专业	19 计科图灵	姓名	张昊	学号	1927405160
课程名称	操作系统课程实践					成绩	
指导教师	李培峰	同组实验者	无		实验日期	2022 年 4 月 12 日	

实 验 名 称　　实验 4　Linux 内存管理

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(首次/最佳/最坏适应算法)。
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。
3. 理解虚拟内存管理的原理和技术。
4. 掌握请求分页存储管理的常用理论——页面置换算法。
5. 理解请求分页中的按需调页机制。

二. 实验内容

1. （**实验 5.1: 动态分区分配方式的模拟**）
   编写 C 程序，模拟实现首次/最佳/最坏适应算法的内存块分配与回收，要求每次分配与回收后显示出空闲分区和已分配分区的情况。假设在初始状态下，可用的内存空间为 640 KB。
2. （**实验 5.2: 页面置换算法的模拟**）
   设计一个虚拟存储区和一个内存工作区，并使用下述常用页面置换算法计算访问命中率。
   (1) 先进先出(first in first out，FIFO)算法。
   (2) 最近最久未使用(least recently used，LRU)算法。
   (3) 最优(optimal，OPT)算法。
   要求如下。
   (1) 通过随机数产生一个指令序列，里面共 320 条指令。
   (2) 将指令序列转换成页面序列。假设：①页面大小为 1 KB；②用户内存容量为 4~32 页；③用户虚存容量为 32 KB。在用户虚存中，按每页存放 10 条指令排列虚存地址， 因此 320 条指令将存放在 32 个页面中。
   (3) 计算并输出不同页面置换算法在不同内存容量下的命中率。访问命中率的计算公式为：

   访问命中率=1-(页面失效次数/页面总数)

三. 操作方法和实验步骤

1. （**实验 5.1: 动态分区分配方式的模拟**）
   根据动态分区分配的原理，主要需要建立两个数据结构：空闲分区表和已分配分区表，使用带有头节点的链表来表示（头节点中的大小表示分区表总共的大小）。它们中需要包含分区的起始地址、长度等信息。分区信息使用结构体保存：

```
typedef struct partition {  //定义分区
  int address, size;
  struct partition *next;
} partition_t;
```

   当有新作业请求装入主存时，程序查找空闲分区表，按照指定的分配方式（首次适应/最佳适应/

最差适应）从中找出一个合适的空闲分区并将其分配给作业，并返回地址（失败返回-1）。其中分区的分配是从内存的低地址开始的。然后按照作业需要的内存大小将其装入主存，剩下的部分仍为空闲分区，将其登记到空闲分区表中，作业占用的分区则登记到已分配分区表中。

```
// 内存分配函数，查找空闲分区，成功返回地址，失败返回-1
int allocate(int size) {
  if (size > unused->size || size < 0) {
    return -1;  // 申请无效
  }
  partition_t *prev = NULL;
  if (tolower(way) == 'b') {
    prev = best_fit_find(size);  // 最佳适应
  } else if (tolower(way) == 'f') {
    prev = first_fit_find(size);  // 首次适应
  } else if (tolower(way) == 'w') {
    prev = worst_fit_find(size);  // 最差适应
  }
  if (prev == NULL) {  // 没有合适的节点
    return -1;
  }
  partition_t *curr = prev->next;
  // 有合适的节点 curr，prev 是其前驱节点
  partition_t *block = malloc(sizeof(partition_t));
  block->size = size;
  block->address = curr->address;
  // 修改 unused
  if (curr->size == size) {  // 若节点大小等于申请大小则完全分配
    prev->next = curr->next;
    free(curr);
  } else {  // 大于申请空间则截取相应大小分配
    curr->size -= size;
    curr->address += size;
  }
  unused->size -= size;
  // 修改 used（头插法）
  block->next = used->next;
  used->next = block;
  used->size += size;
  return block->address;
}
```

首次适应方法查找空闲分区表中节点的实现：

```
// 首次适应空闲分区查找，返回空闲分区，失败返回 NULL
partition_t *first_fit_find(int size) {
  partition_t *prev = unused, *curr = unused->next;
  int found = 0;
  while (curr != NULL) {  // 查找合适节点
```

```
    if (curr->size >= size) {
      found = 1;
      break;
    } else {
      prev = prev->next;
      curr = curr->next;
    }
  }
  if (!found) {  // 没有合适的节点
    return NULL;
  }
  return prev;
}
```

最佳适应方法查找空闲分区表中节点的实现：

```
// 最佳适应：搜索整个列表，找到适合条件的最小的分区进行分配，返回空闲分区
partition_t *best_fit_find(int size) {
  partition_t *prev = unused, *curr = unused->next;
  partition_t *saved_prev = NULL, *saved_curr = NULL;
  int found = 0;
  while (curr != NULL) {  // 查找合适节点
    if (curr->size >= size) {
      found = 1;
      if (saved_curr == NULL || curr->size < saved_curr->size) {//更小
        saved_prev = prev;
        saved_curr = curr;
      }
    }
    prev = prev->next;
    curr = curr->next;
  }
  if (!found) {  // 没有合适的节点
    return NULL;
  }
  return saved_prev;
}
```

最差适应方法查找空闲分区表中节点的实现：

```
// 最差适应：搜索整个列表，找到适合条件的最大的分区进行分配，返回空闲分区
partition_t *worst_fit_find(int size) {
  partition_t *prev = unused, *curr = unused->next;
  partition_t *saved_prev = NULL, *saved_curr = NULL;
  int found = 0;
  while (curr != NULL) {  // 查找合适节点
    if (curr->size >= size) {
      found = 1;
      if (saved_curr == NULL || curr->size > saved_curr->size) {//更大
```

```
      saved_prev = prev;
      saved_curr = curr;
    }
  }
  prev = prev->next;
  curr = curr->next;
 }
 if (!found) {  // 没有合适的节点
  return NULL;
 }
 return saved_prev;
}
```

作业执行完毕后，应根据地址回收作业占用的分区，具体操作为：删除已分配分区表中的相关项，然后修改空闲分区表，并根据情况增加或合并空闲分区。具体实现如下：

```
// 内存回收函数，成功返回释放的空间大小，失败返回-1
int reclaim(int addr) {
 // 检查地址是否在已分配分区表中出现
 // 若有则从表中删除该项，并获取相应分区块指针
 partition_t *prev = used, *curr = used->next;
 while (curr != NULL) {
  if (curr->address == addr) {
   prev->next = curr->next;
   used->size -= curr->size;
   break;
  }
  prev = prev->next;
  curr = curr->next;
 }
 if (curr == NULL) {  // 否则内存回收失败
  return -1;
 }
 partition_t *block = curr;
 int size = block->size;
 unused->size += block->size;
 // 找到首次 curr->address >= block->address 的位置
 // 则 prev-curr 中间为插入点
 prev = unused;
 curr = unused->next;
 while (curr != NULL && curr->address < block->address) {
  prev = prev->next;
  curr = curr->next;
 }
 if (prev != unused && prev->address + prev->size == block->address){
  // 与上一块合并
  prev->size += block->size;
```

```
    free(block);
    block = prev;
  } else {
    // 作为新块插入
    prev->next = block;
    block->next = curr;
  }
  if (curr != NULL && curr->address == block->address + block->size){
    // 与下一块合并
    block->size += curr->size;
    block->next = curr->next;
    free(curr);
  }
  return size;
}
```

2. （**实验 5.2: 页面置换算法的模拟**）

在教材示例代码的基础上进行了修改。保持原有的一个虚拟存储区和一个内存工作区，新增实现了 LRU 和 OPT 页面置换算法，并计算访问命中率。

首先分别使用随机函数随机产生指令序列和教材提供的指令产生方式，然后将指令序列转换成相应的页面序列。

页面类型、页面控制结构等数据结构定义如下：

```
typedef struct {  //页面结构
  int pn;  //页号
  int pfn;  //内存块号
  int time;  //访问时间
} page_type;
page_type page[total_page];  //所有页面
typedef struct pfc_struct { //页面控制结构
  int pn;  //页号
  int pfn;  //内存块号
  struct pfc_struct *next;
} pfc_type;
pfc_type pfc[total_page];  //所有页面的虚页控制结构
pfc_type *free_head;  //空内存页头指针
pfc_type *busy_head;  //忙内存页头指针
pfc_type *busy_tail;  //忙内存页尾指针
int invalid_count;  //页面失效次数
int instructions[total_instruction];  //指令流数据组
int pno[total_instruction];  //每条指令所属页号
int offset[total_instruction];  //每条指令的页号偏移值
```

FIFO 算法的实现：使用忙页面队列和空闲页面队列，以 FIFO 的方式掉入新页面。

```
void FIFO(int total_pf) {
  initialize(total_pf);
  busy_head = busy_tail = NULL;
```

```
  for (int i = 0; i < total_instruction; i++) {
    if (page[pno[i]].pfn == INVALID) {  //页面失效
      invalid_count++;      //失效次数
      pfc_type *p;
      if (free_head == NULL) {   //无空闲页面
        p = busy_head->next;
        free_head = busy_head;  //释放忙页面队列的第一个页面(队头)
        page[busy_head->pn].pfn = INVALID;  //该页设置为无效
        free_head->next = NULL;
        busy_head = p;
      }
      //按 FIFO 方式调入新页面到内存页面
      p = free_head->next;  //取空闲页面队列队头
      free_head->next = NULL;
      free_head->pn = pno[i];  //页号
      page[pno[i]].pfn = free_head->pfn;  //内存块号改为有效
      if (busy_tail == NULL)
        busy_head = busy_tail = free_head;
      else {
        busy_tail->next = free_head;  //空闲页面减少一个
        busy_tail = free_head;
      }
      free_head = p;
    }
  }
  printf("FIFO:%6.4f", 1 - (double) invalid_count / total_instruction);
}
```

LRU 算法的实现：每次访问页面时都记录访问时间，当有页面被调出时，找出访问时间最远的调出。

```
void LRU(int total_pf) {
  initialize(total_pf);
  for (int i = 0, time = 0; i < total_instruction; i++, time++) {
    if (page[pno[i]].pfn == INVALID) {   //页面失效
      invalid_count++;  //失效次数
      int min_time, min_pno;
      if (free_head == NULL) {   //无空闲页面
        min_time = INF;
        for (int j = 0; j < total_page; j++) {  //找出访问时间最远的（最小）
          if (page[j].pfn != INVALID && page[j].time < min_time) {
            min_time = page[j].time;
            min_pno = j;
          }
        }
        free_head = &pfc[page[min_pno].pfn]; //释放最近未访问的页面
        page[min_pno].pfn = INVALID;  //该页设置为无效
```

```
      free_head->next = NULL;
    }
    page[pno[i]].pfn = free_head->pfn;  //内存块号改为有效
    page[pno[i]].time = time;  // 记录访问时间
    free_head = free_head->next;   //空闲页面减少一个
  } else
    page[pno[i]].time = time; //命中，更新访问时间
 }
 printf(" LRU:%6.4f", 1 - (double) invalid_count / total_instruction);
}
```

OPT 算法的实现：标记页面的最远使用指令序号，对于将来不会使用的页面标记为无穷大，否则根据后续的每条指令标记最后一次访问的序号；每次调出页面时，找出最后一次访问的指令序号最大的页。

```
void OPT(int total_pf) {
 initialize(total_pf);
 for (int i = 0; i < total_instruction; i++) {
  if (page[pno[i]].pfn == INVALID) {   //页面失效
   invalid_count++;  //失效次数
   int max_dist, max_pno, dist[total_page];
   if (free_head == NULL) {   //无空闲页面
    // 1：标记将来不会被使用的页面
    // 对于所有无效页面，设置最后一次访问的指令序号为无穷大
    for (int j = 0; j < total_page; j++) {
     if (page[j].pfn != INVALID) {
      dist[j] = INF;
     } else {
      dist[j] = 0;
     }
    }
    // 2：标记最远的将来才会被使用的页面
    //对于此后所有指令所在的页 若无效记录其最后一次访问的指令序号 若有效则为 0
    for (int j = i + 1; j < total_instruction; j++) {
     if (page[pno[j]].pfn != INVALID) {
      dist[pno[j]] = j;
     }
    }
    max_dist = 1;  // 找出最后一次访问的指令序号最大的页
    for (int j = 0; j < total_page; j++) {
     if (dist[j] > max_dist) {
      max_dist = dist[j];
      max_pno = j;
     }
    }
    free_head = &pfc[page[max_pno].pfn]; //释放页面
    free_head->next = NULL;  //该页设置为无效
```

```
        page[max_pno].pfn = INVALID;
      }
      page[pno[i]].pfn = free_head->pfn;  //内存块号改为有效
      free_head = free_head->next;   //空闲页面减少一个
    }
  }
  printf(" OPT:%6.4f", 1 - (double) invalid_count / total_instruction);
}
```

计算使用指定页面置换算法时的访问命中率的方法为在运行中统计页面失效次数，并使用如下公式计算：

```
1 - (double) invalid_count / total_instruction
```

四. 实验结果和分析

1. （**实验 5.1：动态分区分配方式的模拟**）

根据如下作业请求序列运行上述代码：

(1) 作业 1 申请 80 KB。
(2) 作业 2 申请 100 KB。
(3) 作业 3 申请 180 KB。
(4) 作业 4 申请 210 KB。
(5) 作业 3 释放 180 KB。
(6) 作业 1 释放 80 KB。
(7) 作业 5 申请 60 KB。
(8) 作业 4 释放 210 KB。
(9) 作业 2 释放 100 KB。
(10) 作业 5 释放 60 KB。

具体操作与运行结果如下：

首先显示初始化后的空闲分区表与已分配分区表，并要求输入分配方式：

```
holger-405160@hao-zhang:~/codes/exp04$ ./dynamic-memory
************ 空闲分区表 ****************
> 总空间(KB): 640
 index    address    end     size
---------------------------------------
 0        0        640      640
---------------------------------------
************已分配分区表****************
> 总空间(KB): 0
没有分区！
---------------------------------------
输入分配方式 [首次适应(f)/最佳适应(b)/最差适应(w)]: f
```

这里选择**首次适应**方式。

(1) 作业 1 申请 80 KB：输入 a，接着输入 80。

```
分配(a)/回收内存(r)/其他字符退出：a
输入申请空间大小(KB): 80
空间分配成功！ADDRESS=0
************ 空闲分区表 ****************
> 总空间(KB): 560
 index    address    end     size
---------------------------------------
 0        80       640      560
---------------------------------------
************已分配分区表****************
> 总空间(KB): 80
 index    address    end     size
---------------------------------------
 0        0        80       80
---------------------------------------
```

(2) 作业 2 申请 100 KB：输入 a，接着输入 100。

```
分配(a)/回收内存(r)/其他字符退出: a
输入申请空间大小(KB): 100
空间分配成功！ADDRESS=80
************ 空闲分区表 *****************
> 总空间(KB): 460
 index    address    end     size
---------------------------------------
 0        180      640      460
---------------------------------------
************已分配分区表*****************
> 总空间(KB): 180
 index    address    end     size
---------------------------------------
 0        80       180      100
---------------------------------------
 1        0        80       80
---------------------------------------
```

(3) 作业 3 申请 180 KB：输入 a，接着输入 180。

```
分配(a)/回收内存(r)/其他字符退出: a
输入申请空间大小(KB): 180
空间分配成功！ADDRESS=180
************ 空闲分区表 *****************
> 总空间(KB): 280
 index    address    end     size
---------------------------------------
 0        360      640      280
---------------------------------------
************已分配分区表*****************
> 总空间(KB): 360
 index    address    end     size
---------------------------------------
 0        180      360      180
---------------------------------------
 1        80       180      100
---------------------------------------
 2        0        80       80
---------------------------------------
```

(4) 作业 4 申请 210 KB：输入 a，接着输入 210。

```
分配(a)/回收内存(r)/其他字符退出: a
输入申请空间大小(KB): 210
空间分配成功！ADDRESS=360
************ 空闲分区表 *****************
> 总空间(KB): 70
 index    address    end     size
---------------------------------------
 0        570      640      70
---------------------------------------
************已分配分区表*****************
> 总空间(KB): 570
 index    address    end     size
---------------------------------------
 0        360      570      210
---------------------------------------
 1        180      360      180
---------------------------------------
 2        80       180      100
---------------------------------------
 3        0        80       80
---------------------------------------
```

(5) 作业 3 释放 180 KB：输入 r，接着输入 180。

```
分配(a)/回收内存(r)/其他字符退出： r
输入要回收的内存空间的地址： 180
空间回收成功！SIZE=180
********** 空闲分区表 **********
> 总空间(KB): 250
 index    address    end     size
---------------------------------------
 0        180      360      180
---------------------------------------
 1        570      640      70
---------------------------------------
**********已分配分区表**********
> 总空间(KB): 390
 index    address    end     size
---------------------------------------
 0        360      570      210
---------------------------------------
 1        80       180      100
---------------------------------------
 2        0        80       80
---------------------------------------
```

(6) 作业 1 释放 80 KB：输入 r，接着输入 0。

```
分配(a)/回收内存(r)/其他字符退出： r
输入要回收的内存空间的地址： 0
空间回收成功！SIZE=80
********** 空闲分区表 **********
> 总空间(KB): 330
 index    address    end     size
---------------------------------------
 0        0        80       80
---------------------------------------
 1        180      360      180
---------------------------------------
 2        570      640      70
---------------------------------------
**********已分配分区表**********
> 总空间(KB): 310
 index    address    end     size
---------------------------------------
 0        360      570      210
---------------------------------------
 1        80       180      100
---------------------------------------
```

(7) 作业 5 申请 60 KB：输入 a，接着输入 60。

```
分配(a)/回收内存(r)/其他字符退出： a
输入申请空间大小(KB): 60
空间分配成功！ADDRESS=0
********** 空闲分区表 **********
> 总空间(KB): 270
 index    address    end     size
---------------------------------------
 0        60       80       20
---------------------------------------
 1        180      360      180
---------------------------------------
 2        570      640      70
---------------------------------------
```

```
************已分配分区表****************
> 总空间(KB): 370
 index    address    end     size
---------------------------------------
 0        0        60       60
---------------------------------------
 1        360      570      210
---------------------------------------
 2        80       180      100
---------------------------------------
```

可见，首次适应方式选择了空闲分区表中首次出现的符合要求的分区，地址为 0。

(8) 作业 4 释放 210 KB：输入 r，接着输入 360。

```
分配(a)/回收内存(r)/其他字符退出： r
输入要回收的内存空间的地址： 360
空间回收成功！SIZE=210
************ 空闲分区表 ****************
> 总空间(KB): 480
 index    address    end     size
---------------------------------------
 0        60       80       20
---------------------------------------
 1        180      640      460
---------------------------------------
************已分配分区表****************
> 总空间(KB): 160
 index    address    end     size
---------------------------------------
 0        0        60       60
---------------------------------------
 1        80       180      100
---------------------------------------
```

(9) 作业 2 释放 100 KB：输入 r，接着输入 80。

```
分配(a)/回收内存(r)/其他字符退出： r
输入要回收的内存空间的地址： 80
空间回收成功！SIZE=100
************ 空闲分区表 ****************
> 总空间(KB): 580
 index    address    end     size
---------------------------------------
 0        60       640      580
---------------------------------------
************已分配分区表****************
> 总空间(KB): 60
 index    address    end     size
---------------------------------------
 0        0        60       60
---------------------------------------
```

(10) 作业 5 释放 60 KB：输入 r，接着输入 0。

```
分配(a)/回收内存(r)/其他字符退出： r
输入要回收的内存空间的地址： 0
空间回收成功！SIZE=60
************ 空闲分区表 ****************
> 总空间(KB): 640
 index    address    end     size
---------------------------------------
 0        0        640      640
---------------------------------------
************已分配分区表****************
> 总空间(KB): 0
没有分区！
---------------------------------------
```

若选择**最佳适应**，上述第(7)步作业 5 申请到的分区是空闲分区表中最小的，地址为 570：

```
分配(a)/回收内存(r)/其他字符退出：a
输入申请空间大小(KB)：60
空间分配成功！ADDRESS=570
********** 空闲分区表 **********
> 总空间(KB)：270
 index    address    end     size
---------------------------------------
 0        0        80       80
---------------------------------------
 1        180      360      180
---------------------------------------
 2        630      640      10
---------------------------------------
**********已分配分区表**********
> 总空间(KB)：370
 index    address    end     size
---------------------------------------
 0        570      630      60
---------------------------------------
 1        360      570      210
---------------------------------------
 2        80       180      100
---------------------------------------
```

若选择**最差适应**，上述第(7)步作业 5 申请到的分区是空闲分区表中最大的，地址为 180：

```
分配(a)/回收内存(r)/其他字符退出：a
输入申请空间大小(KB)：60
空间分配成功！ADDRESS=180
********** 空闲分区表 **********
> 总空间(KB)：270
 index    address    end     size
---------------------------------------
 0        0        80       80
---------------------------------------
 1        240      360      120
---------------------------------------
 2        570      640      70
---------------------------------------
**********已分配分区表**********
> 总空间(KB)：370
 index    address    end     size
---------------------------------------
 0        180      240      60
---------------------------------------
 1        360      570      210
---------------------------------------
 2        80       180      100
---------------------------------------
```

2. （**实验 5.2：页面置换算法的模拟**）

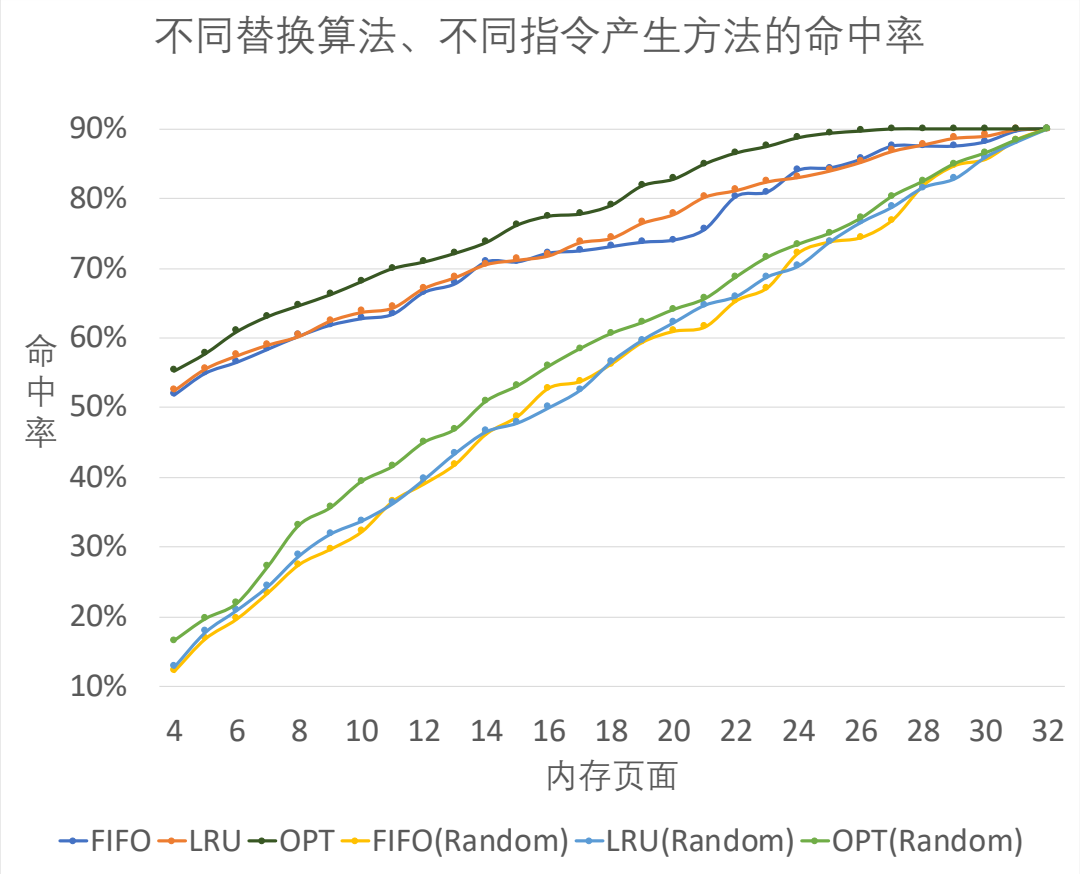
使用教材中给出的产生指令的方法，运行结果如下：

```
holger-405160@hao-zhang:~/codes/exp04$ ./page-replacement
 4 pno frames FIFO:0.5188 LRU:0.5250 OPT:0.5531
 5 pno frames FIFO:0.5500 LRU:0.5563 OPT:0.5781
 6 pno frames FIFO:0.5656 LRU:0.5750 OPT:0.6094
 7 pno frames FIFO:0.5844 LRU:0.5906 OPT:0.6312
 8 pno frames FIFO:0.6031 LRU:0.6031 OPT:0.6469
 9 pno frames FIFO:0.6188 LRU:0.6250 OPT:0.6625
10 pno frames FIFO:0.6281 LRU:0.6375 OPT:0.6813
11 pno frames FIFO:0.6344 LRU:0.6438 OPT:0.7000
12 pno frames FIFO:0.6656 LRU:0.6719 OPT:0.7094
13 pno frames FIFO:0.6781 LRU:0.6875 OPT:0.7219
14 pno frames FIFO:0.7094 LRU:0.7063 OPT:0.7375
15 pno frames FIFO:0.7094 LRU:0.7125 OPT:0.7625
16 pno frames FIFO:0.7219 LRU:0.7188 OPT:0.7750
17 pno frames FIFO:0.7250 LRU:0.7375 OPT:0.7781
18 pno frames FIFO:0.7312 LRU:0.7438 OPT:0.7906
19 pno frames FIFO:0.7375 LRU:0.7656 OPT:0.8187
20 pno frames FIFO:0.7406 LRU:0.7781 OPT:0.8281
21 pno frames FIFO:0.7562 LRU:0.8031 OPT:0.8500
22 pno frames FIFO:0.8031 LRU:0.8125 OPT:0.8656
23 pno frames FIFO:0.8094 LRU:0.8250 OPT:0.8750
24 pno frames FIFO:0.8406 LRU:0.8313 OPT:0.8875
25 pno frames FIFO:0.8438 LRU:0.8406 OPT:0.8938
26 pno frames FIFO:0.8562 LRU:0.8531 OPT:0.8969
27 pno frames FIFO:0.8750 LRU:0.8688 OPT:0.9000
28 pno frames FIFO:0.8750 LRU:0.8781 OPT:0.9000
29 pno frames FIFO:0.8750 LRU:0.8875 OPT:0.9000
30 pno frames FIFO:0.8812 LRU:0.8906 OPT:0.9000
31 pno frames FIFO:0.8969 LRU:0.9000 OPT:0.9000
32 pno frames FIFO:0.9000 LRU:0.9000 OPT:0.9000
holger-405160@hao-zhang:~/codes/exp04$
```

完全随机产生指令（指令地址无具体要求），运行结果如下：

```
holger-405160@hao-zhang:~/codes/exp04$ ./page-replacement
 4 pno frames FIFO:0.1219 LRU:0.1281 OPT:0.1656
 5 pno frames FIFO:0.1687 LRU:0.1781 OPT:0.1969
 6 pno frames FIFO:0.1969 LRU:0.2094 OPT:0.2188
 7 pno frames FIFO:0.2344 LRU:0.2438 OPT:0.2719
 8 pno frames FIFO:0.2750 LRU:0.2875 OPT:0.3313
 9 pno frames FIFO:0.2969 LRU:0.3187 OPT:0.3562
10 pno frames FIFO:0.3219 LRU:0.3375 OPT:0.3938
11 pno frames FIFO:0.3656 LRU:0.3625 OPT:0.4156
12 pno frames FIFO:0.3906 LRU:0.3969 OPT:0.4500
13 pno frames FIFO:0.4187 LRU:0.4344 OPT:0.4688
14 pno frames FIFO:0.4625 LRU:0.4656 OPT:0.5094
15 pno frames FIFO:0.4875 LRU:0.4781 OPT:0.5312
16 pno frames FIFO:0.5281 LRU:0.5000 OPT:0.5594
17 pno frames FIFO:0.5375 LRU:0.5250 OPT:0.5844
18 pno frames FIFO:0.5625 LRU:0.5656 OPT:0.6062
19 pno frames FIFO:0.5938 LRU:0.5969 OPT:0.6219
20 pno frames FIFO:0.6094 LRU:0.6219 OPT:0.6406
21 pno frames FIFO:0.6156 LRU:0.6469 OPT:0.6562
22 pno frames FIFO:0.6531 LRU:0.6594 OPT:0.6875
23 pno frames FIFO:0.6719 LRU:0.6875 OPT:0.7156
24 pno frames FIFO:0.7219 LRU:0.7031 OPT:0.7344
25 pno frames FIFO:0.7375 LRU:0.7375 OPT:0.7500
26 pno frames FIFO:0.7438 LRU:0.7656 OPT:0.7719
27 pno frames FIFO:0.7688 LRU:0.7875 OPT:0.8031
28 pno frames FIFO:0.8187 LRU:0.8156 OPT:0.8250
29 pno frames FIFO:0.8469 LRU:0.8281 OPT:0.8500
30 pno frames FIFO:0.8562 LRU:0.8594 OPT:0.8656
31 pno frames FIFO:0.8844 LRU:0.8812 OPT:0.8844
32 pno frames FIFO:0.9000 LRU:0.9000 OPT:0.9000
holger-405160@hao-zhang:~/codes/exp04$
```

将上述运行结果统计如下：



可以看到：当内存页面比较少的时候，访问命中率不高，但随着内存页面的增多，访问命中率开始提高；OPT 算法的性能最佳，LRU 算法的命中率较 FIFO 在大多数情况下有一定的提高；当指令完全随机生成时（即指令不具有局部性时），命中率会显著降低，尤其是内存页面数量较低时。

五. 讨论、心得

1. 本次实验的主要内容 Linux 内存管理，通过本次实验，我掌握了动态分区分配算法：首次/最佳/最坏适应算法，加深了对动态分区分配管理方式及其实现过程的理解。与教材不同的是，本次实验中动态分区分配算法是在分配内存空间时执行的，个人认为这样更能体现分配算法的实质。
2. 第二个小实验让我对页面置换算法有了更进一步的认识，加深了我对请求分页中的按需调页机制的理解。通过对 FIFO、LRU、OPT 算法的实现，以及对结果的分析，我对这三个算法总结如下：FIFO 实现方便，缺页率可以较高；OPT 性能最佳，但在现实中无法实现；LRU 实现时较复杂，且需要硬件支持。现实中常用近似算法如 LFU 等，但性能较靠近 OPT 算法。