Report for Lab07

Stu：金泽文

No.PB15111604

实验目的：

在实验 6 的基础上，通过实现动态内存管理，固定内存管理，以及内存有效性检查，学习并理解操作系统内存管理的原理与细节。

实验内容：

1. 检测物理内存：实现内存有效性检查方案以检查你所管理的动态内存。
    - 为简单起见，允许直接从 1MB 开始。
    - 接口：pMemStart（？=1MB）和 pMemSize。
    - 操作系统初始化过程中进行检查，并汇报 pMemStart 和 pMemSize 的值（要求 16 进制表示）
    - 附加（非强制要求）：从 0 地址开始。
2. 实现动态分区算法：dPartition
    - 给定任意大小的内存区域，按照动态分区算法对其进行初始化。接口：dPartitionInit()。（要求：若内存区域太小，小于最小大小，进行失败提示）
    - 从动态分区中分配一个指定大小的分区（允许进行 4 字节、8 字节或你自定义的某个大小对齐）。接口：dPartitionAlloc()
    - 释放一个分区。接口：dPartitionFree()
    - 将检测到的动态物理内存，按照动态分区的方式进行管理

    - 重新封装动态物理内存的分配和回收接口，提供 malloc 和 free 接口给 userApp。
    -

3. 实现等大小固定分区算法：eFPartition
    - 给定分区大小和分区个数，结合你的内部管理数据结构开销，计算出总大小。接口：eFPartitionTotalSize()
    - 将给定内存区域初始化为若干个等大小固定分区。初始化接口：eFPartitionInit()
    - 分配一个固定大小分区。接口：eFPartitionAlloc()
    - 释放一个固定大小分区。接口：eFPartitionFree()
    - 修改你的任务池分配算法：从动态物理内存中，分配一个动态分区，该动态分区能容纳指定个数的任务和内部数据管理开销，使用等大小固定分区算法管理这个动态分区。
4. 修改 osStart 原语，以增加内存管理功能

- 实验报告内容

实验分析与实验过程：

本次实验重点在于实现。

内存有效性检查：

首先是内存有效性检查。实现方式很简答， 因为老师上课说过实现方法：依次写入并读取内存，如果发现写入值与读取值不同，则认为内存无效。另外值得一提的是，我额外多写入，读取了另一个值，以防之前的读写一致是偶然所致。

在 os 开头调用，并且输出对应结果。

```
154 void pMemInit(void){ //
    内存检测并采用dPartition初始化malloc机制
155     unsigned long *i = 0x100000;
156     while(1){
157         *i = 1;
158         if(*i != 1){
159             myprintf(0x7, "\nmemory available:0x%x
                    \n", i);
160             return;
161         }
162         *i = 2;
163         if(*i != 2){
164             myprintf(0x7, "\nmemory available:0x%x
                    \n", i);
165             return;
166         }
167         *i = 0;
168         i += 0x1000;
169     }
170 }
```

函数 myprintf：

为了实现可变长参数的函数，使用 stdarg 头文件，通过维基百科-stdarg.h 中的阐述，学会实现方式。值得一提的是，实现 myprintf 过程中，我是用的 put_char 和

put_chars 函数移植于陈老师之前代码的思路，并且有较大改动。另外，为了将更多的精力放在实验上，我实现的 myprintf 只支持\n \r %d %x 的特殊字符。

```
void myprintf(int color, const char *format, ...){
    va_list ap;

    va_start(ap, format);
    int i_print, i_format;
    for(i_print = 0; i_print < 80; i_print++){
        print[i_print] = '\0';
    }
    i_print = 0;
    i_format = 0;
    int vi;
    unsigned long vl;

    while(format[i_format] != '\0'){
        if(format[i_format] == '%'){
            if(format[i_format + 1] == 'x'){
                vl = va_arg(ap, unsigned long);
                i_format += 2;
                unsigned long mask = 0xf0000000;
                int hex;
                int i = 28;
                while(mask != 0){
                    hex = (vl & mask) >> i;
                    if(hex > 9)
                        print[i_print++] = hex - 10
                            + 'a';
                    else
                        print[i_print++] = hex + '0'
                            ;
                    mask = mask / 16;
                    i = i - 4;
                }
                continue;
            }
            else if(format[i_format + 1] == 'd'){
                vi = va_arg(ap, int);
                i_format += 2;
                if(vi == 0){
                    print[i_print++] = '0';
                    continue;
                }
                int decimal[10];
                int i;
                for(i = 9; i >= 0; i--){
                    decimal[i] = vi % 10;
                    vi = vi / 10;
                }
                i = 0;
                while(decimal[i] == 0)
                    i++;
                for(i = i; i < 10; i++){
                    print[i_print++] = decimal[i] + '0';
                }
                continue;
            }
        }
        print[i_print++] = format[i_format++];
    }
    va_end(ap);
    put_chars(print, color);
}
```

align：

为了实现对齐而设计的函数。

```
unsigned long align(unsigned long size){
    if(size % 4 == 0)
        return size;
    else
        return ((size >> 2) << 2) + 4;
}
```

dPartitionInit：

我采用的不是链表或者数组的实现方式，而是针对每一个 partition，在前面分出一 head。第一个 32 位存放这个 partition 的大小。之后根据这个大小确定最多的节点个数，位 size/0x10，这样做能保证节点个数动态变化。之后的 size/0x10 个 32 位内存每个存放一个页内偏移 index。

```
12  unsigned long dPartitionInit(unsigned long start,
        unsigned long size){//初始化并返回dPartition句柄
13      size = align(size);
14      *(long *)start = size;
15      int i;
16      for(i = 0; i < size / 0x10; i++){
17          *(short *)(start + 4 + 4 * i) = 0;//index
18          *(short *)(start + 6 + 4 * i) = 0;//size
19      }//at most size / 0x10 nodes
20
21      unsigned long j = 0;
22      while(j < size / 0x10){
23          *(long *)(start + j * 4 + 4 * i + 4) = 0;
24          j += 1;
25      }
26      return start;
27  }
```

dPartitionAlloc ：

为了实现这个函数，额外写了 dPartitionInsert 函数，用来动态插入的。

```
29  unsigned long dPartitionInsert(unsigned long dp, int
        index, unsigned long size){
30      unsigned long max_size = *(unsigned long *)dp;
31      int max_nodes = max_size / 0x10;
32      int i = max_nodes - 2;
33      for(; i > index; i--){
34          *(unsigned long *)(dp + 4 * i + 8) = *(
                unsigned long *)(dp + 4 * i + 4);
35      }
36      *(short *)(dp + 4 * index + 6) = size;
37      if(i == 0){
38          *(short *)(dp + 4 * index + 4) = max_nodes *
                4 + 4;
39          return *(short *)(dp + 4 * index + 4) + dp;
40      }
41      else{
42          *(short *)(dp + 4 * index + 4) = *(short *)(
                dp + 4 * index) + *(short *)(dp + 4 *
                index + 2);
43          return *(short *)(dp + 4 * index + 4) + dp;
44      }
45  }
```

主要是根据 **dPartitionInit** 中所设计的表头，依次查看，按照 first fit 算法分配内存。需要注意块头前面那些 index 信息要实时更新，保证中间没有空 index。

```
47  unsigned long dPartitionAlloc(unsigned long dp,
        unsigned long size){//
    0失败；其他：所分配到的内存块起始地址
48      size = align(size);
49      unsigned long max_size = *(unsigned long *)dp;
50      int max_nodes = max_size / 0x10;
51      int i = 0;
52      int available;
53      for(; i < max_nodes - 1; i++){
54          if(*(short *)(dp + 4 * i + 4) == 0){
55              if(i == 0){//none malloced
56                  if(size <= max_size){
57                      return dPartitionInsert(dp, 0,
                            size);
58                  }
59                  else
60                      return 0;
61              }
62              else{
63                  unsigned long left = max_size - *(
                        short *)(dp + 4 * i) - *(short *
                        )(dp + 4 * i + 2);
64                  if(size <= left){
65                      return dPartitionInsert(dp, i,
                            size);
66                  }
67                  else
68                      return 0;
69              }
70          }
71          else if(i < max_nodes - 2){
72              if(*(short *)(dp + 4 * i + 8) == 0){
73                  unsigned long left = max_size + 4 *
```

```
62                  else{
63                      unsigned long left = max_size - *(short *)(dp
                            + 4 * i) - *(short *)(dp + 4 * i + 2);
64                      if(size <= left){
65                          return dPartitionInsert(dp, i, size);
66                      }
67                      else
68                          return 0;
69                  }
70              }
71              else if(i < max_nodes - 2){
72                  if(*(short *)(dp + 4 * i + 8) == 0){
73                      unsigned long left = max_size + 4 * max_nodes
                            + 4- *(short *)(dp + 4 * i + 4) - *(short
                            *)(dp + 4 * i + 6);
74                      if(size <= left){
75                          return dPartitionInsert(dp, i + 1, size);
76                      }
77                      else
78                          return 0;
79                  }
80                  else{
81                      unsigned long available = *(short *)(dp + 4 *
                            i + 8) - *(short *)(dp + 4 * i + 4) - *(
                            short *)(dp + 4 * i + 6);
82                      if(size <= available)
83                          return dPartitionInsert(dp, i + 1, size);
84                      else return 0;
85                  }
86              }
87          }
88      return 0;
89  }
```

dPartitionFree ：

```
90  void dPartitionDelete(unsigned long dp, int inode){
91      unsigned long max_size = *(unsigned long *)dp;
92      int max_nodes = max_size / 0x10;
93      int i = inode;
94      for(; i < max_nodes - 1; i++){
95          *(unsigned long *)(dp + 4 * i + 4) = *(
                unsigned long *)(dp + 4 * i + 8);
96      }
97  }
98
```

```
99  unsigned long dPartitionFree( unsigned long dp,
        unsigned long start){ //0失败；1成功
100     unsigned long max_size = *(unsigned long *)dp;
101     int max_nodes = max_size / 0x10;
102     int i = 0;
103     short shift = start - dp;
104     for(; i < max_nodes; i ++){
105         if(shift == *(short *)(dp + 4 * i + 4)){
106             dPartitionDelete(dp, i);
107             return 1;
108         }
109     }
110     return 0;
111 }
```

为了实现这个函数，额外写了 dPartitionDelete 函数，用来动态删除。

eFPartitionTotalSize：

为了保存节点头信息，需要为 partition 设置表头，保存 n 和 perSize。

另外为了节点标记，所以每个块需要弄一个标志位。所以，最后的 totalsize 就是 n * (perSize + 4) + 8。另外要注意对齐。

```
121 unsigned long eFPartitionTotalSize(unsigned long
        perSize, unsigned long n){ //
    根据单位大小和单位个数，计算出eFPartition所需内存大小
122     perSize = align(perSize);
123     return n * (perSize + 4) + 8;
124     // since each page needs a head and we need a
        general head.
125 }
```

eFPartitionInit：

按照 **eFPartitionTotalSize** 中所说的实现。

```
113  unsigned long eFPartitionInit(unsigned long start,
         unsigned long perSize, unsigned long n){ //
     初始化并返回eFPartition句柄
114      *(unsigned long*)start = n;
115      *(unsigned long*)(start + 4) = perSize;
116      for(int i = 0; i < n * (perSize + 4); i += 4){
117          *(unsigned long*)(start + 8 + i) = (unsigned
                 long)0;
118      }
119      return start;
120  }
```

eFPartitionAlloc 和 **eFPartitionFree**：

由于分配时只需要一个参数，所以第一个空闲的就是要分配的，无须比较 size。

Free 时依次比较地址是否相等。

```
126  unsigned long eFPartitionAlloc(unsigned long
         EFPHandler){ //0失败；其他：所分配到的内存块起始地址
127      unsigned long n = *(unsigned long*)EFPHandler;
128      unsigned long perSize = *(unsigned long*)(
             EFPHandler + 4);
129      int i = 0;
130      for(; i < n; i++){
131          unsigned long *head = (unsigned long*)(
                 EFPHandler + 8 + perSize * i);
132          if(*head == 0){
133              *head = 1;
134              return head + 4;
135          }
136      }
137      return 0;
138  }
139  unsigned long eFPartitionFree(unsigned long
         EFPHandler,unsigned long mbStart){ //0失败；1成功
140      unsigned long n = *(unsigned long*)EFPHandler;
141      unsigned long perSize = *(unsigned long*)(
             EFPHandler + 4);
142      int i = 0;
143      for(; i < n; i++){
144          unsigned long *head = (unsigned long*)(
                 EFPHandler + 8 + perSize * i);
145          if(*head == 1 && head + 4 == mbStart){
146              *head = 0;
147              return 1;
148          }
149      }
150      return 0;
151  }
```

数据结构的接口，设计说明：

由于内存的管理都是在每个 partition 头部设置对应的参数，动态管理对应 size 和索引表，固定管理对应 perSize 和 n。详细地说一下。

动态管理数据结构：（以 0x1000 的 dp，0x40 的 size 为例）

0x1000（保存 size）-- 0x40

0x1004（开始保存索引表）-- index-0 （保存对应偏移量。）

0x1006（这一部分 short 保存 size）--index-0-size

0x1008— index-1 （保存对应偏移量。）

0x100a（这一部分 short 保存 size）--index-1-size

0x100c— index-2

0x100e（这一部分 short 保存 size）--index-2-size

0x1010— index-3

0x1012（这一部分 short 保存 size）--index-3-size

（索引表到这结束，因为最多 0x40/0x10 = 4 个）

0x1014-- index-0 （偏移 0x14）

固定管理数据结构：

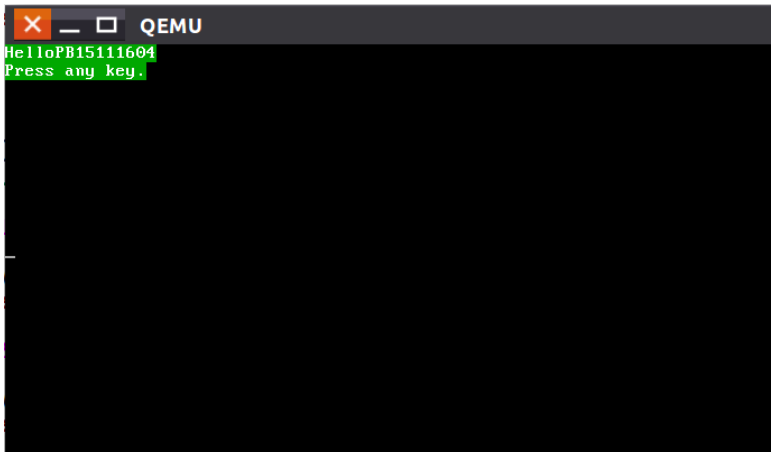
这一部分较为简单，大致说一下。

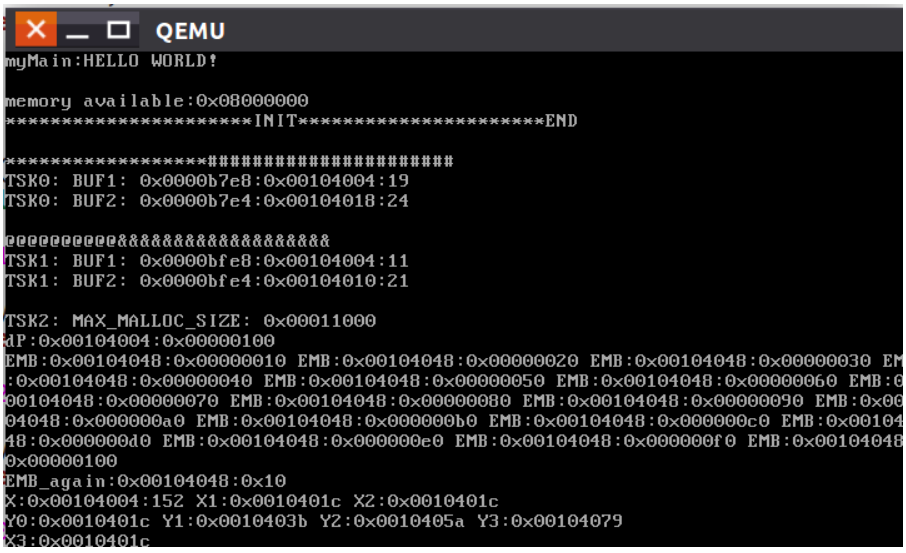
开头第一个 32 位保存块数 n，第二个 32 位保存每个块对应的大小（对齐之后）。之后每一个块第一个 32 位保存标志，0 表示这个块是 available 的，1 表示 allocated。

原语的接口、设计说明：

这一部分上面针对每个函数都有详尽的说明。

测试与运行：





可以看到，有内存有效性检查输出，以及各个 tsk 对应的正确输出。