Operating System Lab 1 : 可变分区存储管理

实 验 报 告

刘子涵　518021910690

【1】实验题目

编写一个 C 程序，用 `char *malloc(unsigned size)` 函数向系统申请一次内存空间（如 `size=1000`， 单位为字节），用**循环首次适应法**：

`addr = (char *)lmalloc(unsigned size)`

`lfree(unsigned size,char *addr)`

模拟 UNIX 可变分区内存管理，实现对该内存区的分配和释放管理。

【2】实验目的

1. 加深对可变分区的存储管理的理解；
2. 提高用 C 语言编制大型系统程序的能力，特别是掌握 C 语言编程的难点：指针和指针作为函数参数；
3. 掌握用指针实现链表和在链表上的基本操作。

【3】实验环境

- 操作系统：Ubuntu 20.04
- Linux 内核版本：5.4.0
- GCC 版本：9.3.0

【4】算法思想

1. 可变分区存储管理

可变分区分配，又称**动态分区分配。**这种分配方式不像固定分区分配那样预先划分内存分区，而是在进程装入内存时，根据进程的大小动态建立分区，使得分区的大小正好适合进程的需要。系统一般可以使用**空闲分区表**（数组）或者**动态分区链**（双向链表）来记录内存的分配情况。可变分区分配没有内部碎片，但有外部碎片，内存中会出现某些空闲分区由于太小而难以利用。

具体的分配算法主要有四种：**首次适应法（First Fit）**、**最佳适应法（Best Fit）**、**最坏适应法（Worst Fit）**、**循环首次适应法（Next Fit）**。

内存释放需要考虑四种不同情况，这里在以下的第 3 部分详细说明。

2. 循环首次适应分配算法（Next Fit）

本次实验采用循环首次适应算法进行内存分配。即把空闲分区表设计成**双向链表**的形式，各表项按起始地址从低到高的次序登记在空闲分区表中。同时需要设置两个指针，一个是指向链表头部节点的 `head`，另一个是指向下一个查找节点的 `next`。循环首次适应法总是从 `next` 指针所指的表项开始查找，找到第一个满足要求的表项进行分配，然后修改相应表项，修改指针 `next` 使其指向下一块空闲区（如果是最后一块那就指向第一块）。

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	以起始地址递增次序排列	算法开销小，释放分区后不需要对空闲分区表重新排序	
最佳适应	优先使用更小的分区，以保留更多的大分区	以分区大小递增次序排列	会有更多的大分区保留下来，满足大进程需求	会产生很多太小的、难以利用的碎片；算法开销大，释放分区后需要对空闲分区表重新排序
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	以分区大小递减次序排列	可以减少小碎片的产生	大分区容易很快被用完，不利于大进程；算法开销大，释放分区后需要对空闲分区表重新排序
循环首次适应	类似首次适应，但每次都是从上次查找的结束位置继续查找	以起始地址递增次序排列（循环链表）	算法开销小，释放分区后不需要对空闲分区表重新排序；不用每次都从低地址的小分区开始检索	会使高地址的大分区很快被用完

3. 内存释放算法

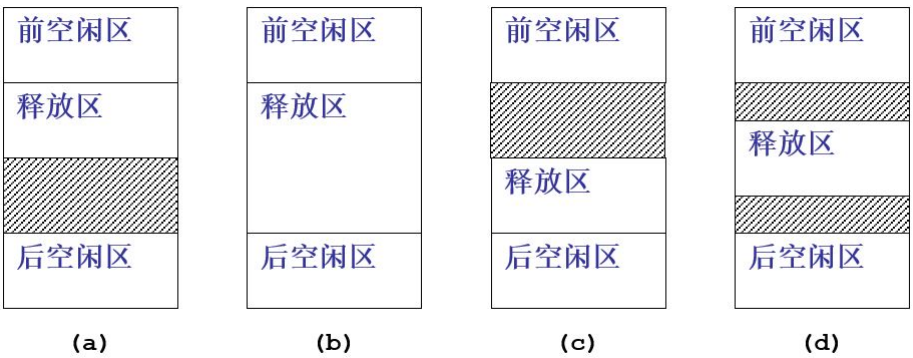
内存释放需要考虑如下四种情况：

（a）*只有释放区之前有相邻空闲分区*：合并前空闲区和释放区，修改前空闲区表项（起始地址不变、分区大小更新为两分区之和）。

（b）*释放区前后都有相邻空闲分区*：合并前空闲区、释放区和后空闲区，修改前空闲区表项（起始地址不变、分区大小更新为三分区之和），删除后空闲区表项。

（c）*只有释放区之后有相邻空闲分区*：合并释放区和后空闲区，修改后空闲区表项（起始地址更新为释放区起始地址、分区大小更新为两分区之和）。

（d）*释放区前后都没有相邻空闲分区*：在前空闲区表项和后空闲区表项之间插入一个新的表项，起始地址和分区大小均为该释放区的起始地址和分区大小。

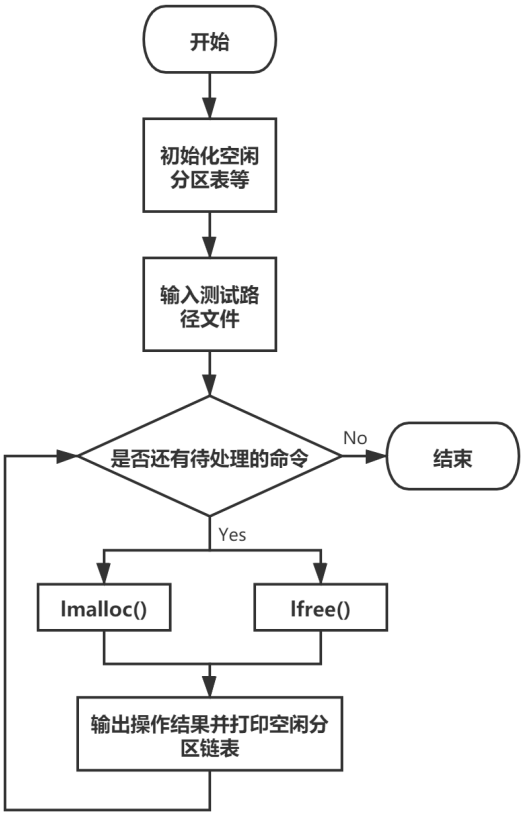


总之，释放算法的核心思想是，释放之后如果与前后邻接，则需要合并表项。但需要注意的是，空闲分区表头部和尾部只有两者情况，插入新的表项或者与相邻表项进行合并，需要专门讨论这种特殊情况。

【5】算法实现

1. 总体流程与模块设计

以下流程图主要展示了 `main.c` 主程序的执行逻辑，在同目录下的 `dpmm.h` 和 `dpmm.c` 定义并实现了上述算法。首先，主程序调用初始化函数，初始化空闲分区表数据结构；然后，通过命令行参数接收输入测试文件；根据每条指令是分配还是释放，分别调用 `lmalloc` 和 `lfree` 函数实现对空闲分区表的修改；修改完成后，输出操作结果（成功或失败，失败输出报错信息）并打印更新后的空闲分区链表。



2. 数据结构

在 `dpmm.h` 中定义了 2 种结构体数据结构：空闲区表项 `node` 和空闲区管理器 `Manager`，定义如下：

```
typedef struct map {
    unsigned m_size;
    char *m_addr;
    struct map *next, *prev;
} node;

typedef struct {
    node *head, *next;
    unsigned maxsize;
} Manager;
```

空闲区表项 `node` 包括无符号整型的分区大小 `m_size` 和字符指针类型的起始地址 `m_addr`，因为需要构造双向循环链表，所以还要指向前后节点的指针 `next` 和 `prev`。

空闲区管理器 `Manager` 包括指向链表头节点的指针 `head` 和查找的下一节点指针 `next`，以及分配的虚拟内存最大大小 `maxsize`。

3. 功能实现

在 `dpmm.h` 中定义了 5 个函数，并在 `dpmm.c` 中实现。以下主要介绍下每个函数的功能和实现的大致思路，代码详见 GitHub:zhliuworks/OS-lab 。

```
/* initialize memory manager */
Manager *dpmm_init(unsigned size);

/* free memory manager */
void dpmm_exit(Manager *manager);

/* allocate memory block with `size` */
char *lmalloc(Manager *manager, unsigned size);

/* free memory block with `size` at `addr` */
bool lfree(Manager *manager, unsigned size, char *addr);

/* display the linked list */
void display(Manager *manager);
```

- `dpmm_init` ：用于初始化空闲区管理器 `Manager` 。输入参数为 `size` 表示申请的最大内存，然后该函数建立只有一个节点的双向循环链表，起始地址为 0 ，分区大小为 size ，返回一个指向 `Manager` 类型的指针。
- `dpmm_exit` ：用于释放空闲区管理器 `Manager` 。输入参数为指向待释放 `Manager` 的指针，然后该函数将每个节点申请的内存空间进行释放，最后释放 `Manager` 。
- `lmalloc` ：用于使用 Next Fit 方法进行分区分配。输入参数为指向管理器的指针和需要申请的内存大小 `size` 。这个函数首先需要检查若干特殊情况，比如此时链表为空，或者管理器的 `next` 指针为 `NULL` 的情况。然后开始执行分配算法：首先需要判断链表是否只有一个节点，因为单节点链表和多节点链表的处理逻辑有些许差别，合并在一起写增加了代码的可读性，所以将两种情况分别讨论。如果存在满足要求的节点，需要判断表项分区大小和申请大小是否一致，如果一致需要删除该节点。如果遍历了整个链表，没有发现合适的节点，则输出分配失败的错误信息，返回 `-1` ，否则返回成功分配内存块的起始地址。
- `lfree` ：用于实现分区释放算法。输入参数为指向管理器的指针、需要释放的内存大小 `size` 和起始地址 `addr` 。成功释放返回 `true` ，否则返回 `false` 。这个函数首先检查地址是否越界，然后分两种情况：非空链表和空链表。对于空链表，简单创建一个节点即可；而对于非空链表，情况比较复杂，为了保障代码的可读性和可调试性，仍然采取分类讨论的代码实现流程。考虑：

  ① 释放头节点前的内存空间；

  ② 释放中间内存空间；

  ③ 释放尾节点后的内存空间 这三种情况。

  这三种情况。如前所述，① 和 ③ 情况分别都只有两种情形：插入新的表项 or 与相邻一个表项进行合并。而情况 ② 有四种情形。除此之外，还需要考虑若干错误的用户请求类型，比如释放本来就是空闲的内存区域，这种需要输出报错信息，并返回 `false` 。
- `display` ：用于打印整个链表。打印的格式直接以易读的形式展现，比如 `[256|512 B]-->[1024|1024B]` ，如果链表为空，则输出 `NULL` 。

【6】测试方法

1. 路径测试方法

路径测试（Path Testing），是指根据路径设计测试用例的一种技术，在程序控制流图的基础上分析各种可能的执行路径，从而按照此路径设计用例，设计出的若干测试用例需要覆盖所有可执行语句。本实验有比较清晰的流程，所以可以利用路径测试的思想来设计测试用例。这里设计了 4 个用例，包括：① 基本流程和错误；② `lmalloc` 的 “循环” 和 “首次”；③④ `lfree` 与相邻空闲分区的四种情况：包括插入新节点和各种合并情况。

2. 输入文件格式

每行是一个命令，其中分配内存命令为 `m [size]` ，表示申请大小为 `size` 的内存；释放内存命令为 `f [size] [addr]` ` 。如下例：

```
m 512
m 1024
f 1024 0
f 512 1024
m 8192
m 4096
f 4096 0
```

3. 自动化 Makefile 和 Shell 脚本

为了使测试过程更加便捷，我编写了 `Makefile` 编译 C 源文件（使用 GCC 编译器）和一个自动化测试的 Shell 脚本（测试 `tests/` 目录下所有测试样例）。

4. 测试效果截图

直接运行 `test.sh` 脚本进行测试，测试效果如图：

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab1# ./test.sh
gcc -w -O1 -std=c11 main.c dpmm.c -o main
********************************************************************************
Testing tests/path1_simple
[0|4096 B]
--------------------------------------------------------------------------------
✓ Successfully allocate 512 B memory at address [0].
[512|3584 B]
--------------------------------------------------------------------------------
✓ Successfully allocate 1024 B memory at address [512].
[1536|2560 B]
--------------------------------------------------------------------------------
✓ Successfully free 1024 B memory at address 0.
[0|1024 B]-->[1536|2560 B]
--------------------------------------------------------------------------------
✓ Successfully free 512 B memory at address 1024.
[0|4096 B]
--------------------------------------------------------------------------------
[Error] System out of memory. Try to allocate 8192 B but only 4096 B are available.
× Fail to allocate 8192 B memory.
[0|4096 B]
--------------------------------------------------------------------------------
✓ Successfully allocate 4096 B memory at address [0].
NULL
--------------------------------------------------------------------------------
✓ Successfully free 4096 B memory at address 0.
[0|4096 B]
--------------------------------------------------------------------------------
```

```
********************************************************************************
Testing tests/path2_malloc_next_fit
[0|4096 B]
--------------------------------------------------------------------------------
√Successfully allocate 2048 B memory at address [0].
[2048|2048 B]
--------------------------------------------------------------------------------
√Successfully free 1024 B memory at address 0.
[0|1024 B]-->[2048|2048 B]
--------------------------------------------------------------------------------
√Successfully allocate 1024 B memory at address [2048].
[0|1024 B]-->[3072|1024 B]
--------------------------------------------------------------------------------
√Successfully allocate 1024 B memory at address [0].
[3072|1024 B]
--------------------------------------------------------------------------------
√Successfully allocate 512 B memory at address [3072].
[3584|512 B]
--------------------------------------------------------------------------------
```

```
********************************************************************************
Testing tests/path3_free_create
[0|4096 B]
--------------------------------------------------------------------------------
√Successfully allocate 4096 B memory at address [0].
NULL
--------------------------------------------------------------------------------
√Successfully free 512 B memory at address 0.
[0|512 B]
--------------------------------------------------------------------------------
√Successfully free 512 B memory at address 1024.
[0|512 B]-->[1024|512 B]
--------------------------------------------------------------------------------
√Successfully free 1024 B memory at address 3072.
[0|512 B]-->[1024|512 B]-->[3072|1024 B]
--------------------------------------------------------------------------------
```

```
********************************************************************************
Testing tests/path4_free_merge
[0|4096 B]
--------------------------------------------------------------------------------
√Successfully allocate 4096 B memory at address [0].
NULL
--------------------------------------------------------------------------------
√Successfully free 512 B memory at address 0.
[0|512 B]
--------------------------------------------------------------------------------
√Successfully free 512 B memory at address 512.
[0|1024 B]
--------------------------------------------------------------------------------
√Successfully free 512 B memory at address 2048.
[0|1024 B]-->[2048|512 B]
--------------------------------------------------------------------------------
√Successfully free 1024 B memory at address 1024.
[0|2560 B]
--------------------------------------------------------------------------------
√Successfully allocate 2560 B memory at address [0].
NULL
--------------------------------------------------------------------------------
√Successfully free 1024 B memory at address 3072.
[3072|1024 B]
--------------------------------------------------------------------------------
√Successfully free 1024 B memory at address 2048.
[2048|2048 B]
--------------------------------------------------------------------------------
```

【7】实验心得

经过本次实验，我加深了对可变分区存储管理思想的理解，在编写代码过程中深入体会了循环首次适应法的优缺点、释放内存块的四种情况，这些思想对算法设计有很大帮助。这次实验的难点在于用指针实现链表和一些基本操作，虽然思路非常简单，在数据结构课程中也有接触，但是在实际操作中容易遇到一些因指针引起的难以调试的 bug，比如使用 free 释放了一块内存空间，指向这块空间的指针成为了野指针，没有对象，但它又并非 NULL 。所以，在最后调试过程中，我检查了每次 free 之前的链表头指针 head 和下一查找节点指针 next 是否会出现野指针的情况，如果出现将其赋值为 NULL 。这提醒我自己，在开发此类大型程序时，切记不要出现指针找不到对象的问题，可以在实现过程中多写单元测试，检查各种可能出现的异常情况。

我的程序还可以进一步从几个方面进行改进：① 支持更多的动态分区分配算法；② 完善异常处理部分的代码，可以定义一个异常类，将各种异常情况分类；③ 有些地方的逻辑是分情况讨论，但实际上可以合并，代码可以更加简洁。

总之，本次实验难度适中，既训练了我的 C/C++ 编程能力，又在 debug 过程中收获甚多，查询并学习了很多 C 标准库的 API（比如文件操作、sscanf 等）。期待下次实验！
