Malloc Lab

实验目标

编写一个动态内存分配器（即自己实现 `malloc`，`free` 以及 `realloc` 方法）。

实验要求

修改下发的 `mm.c` 文件，最终实现一个正确高效的分配器设计。

实验步骤

1. 登陆服务器
    - 地址：ics.ayaya.in
    - 用户名与密码：与之前实验相同
2. 使用 `cp ~/../malloclab-handout.tar ~/` 将实验文件复制到自己的用户目录下
3. 使用 `tar -xvf malloclab-handout.tar` 解压。
4. 按要求完成作业。

你的动态内存分配器将由以下四个函数组成，它们的定义和实现可以分别在 `mm.h` 和 `mm.c` 中找到。

```
int   mm_init(void);
```

```
void *mm_malloc(size_t size);
```

```
void  mm_free(void *ptr);
```

```
void *mm_realloc(void *ptr, size_t size);
```

`mm.c` 中提供了我们能想到的最简单但仍然功能正确的 `malloc` 包。以此为起点修改这些函数（可能还要定义其他的私有 `static` 函数），使它们遵守以下要求：

- `mm_init`：在调用 `mm_malloc`，`mm_realloc` 或者 `mm_free` 之前，应用程序（例如用来评估实现的程序）调用 `mm_init` 来进行任何需要的初始化操作，例如分配初始堆空间。如果在初始化过程中出现问题，返回值需要被设置为 `-1`，否则为 `0`。
- `mm_malloc`：`mm_malloc` 返回一个指针，该指针指向一个大小至少为 `size` 字节的已分配块。整个已分配块需要在堆区域并且不与其他的已分配块重叠。我们会将你的实现与 C 标准库（libc）的版本作比较。因为 libc 的 `malloc` 总是返回 `8` 字节对齐的指针，因此你的 `malloc` 实现也应该像这样并总是返回 `8` 字节对齐的指针。
- `mm_free`：`mm_free` 释放 `ptr` 指向的块，没有返回值。只有当传递的指针（`ptr`）是由之前的 `mm_malloc` 或者 `mm_realloc` 分配并且还没有被释放的时候，该方法才能保证工作。
- `mm_realloc`：`mm_realloc` 返回一个指向大小至少为 `size` 字节的已分配区域的指针，并且有以下限制：
    - 如果 `ptr` 是 `NULL`，该调用等价于 `mm_malloc(size)`。
    - 如果 `size` 为 `0`，该调用等价于 `mm_free(ptr)`。

- 如果 `ptr` 不是 `NULL`，它必须是由之前的 `mm_malloc` 或者 `mm_realloc` 调用返回。对 `mm_realloc` 的调用改变 `ptr` 指向的内存块的大小（旧块）到 `size` 字节并且返回新块的地址。注意，新块和旧块的地址可能一样，也可能不一样，这取决于你的实现细节、旧块的内部碎片大小以及 `realloc` 要求的大小。新块的内容需要与旧块相同（大小为新块和旧块之间的最小值）。其他的都是未被初始化的。例如，如果旧块是 `8` 字节而新块是 `12` 字节，那么新块的头 `8` 字节需要与旧块的头 `8` 字节相同，剩下的 `4` 字节未初始化。类似的，如果旧块是 `8` 字节而新块是 `4` 字节，则新块的内容应该与旧块的头 `4` 字节相同。

这些要求与对应的 libc `malloc`， `realloc` 以及 `free` 相同。在 shell 中输入 `man malloc` 可以获得完整文档。

`memlib.c` 为你的动态内存分配器模拟了内存系统，你可以调用 `memlib.c` 中的以下函数：

- `void *mem_sbrk(int incr)`：使堆增加 `incr` 字节，参数 `incr` 是一个正整数，函数返回一个指向新分配的堆区域的第一个字节的指针。该语义与 Unix 的 `sbrk` 函数相同，但是 `mem_sbrk` 只接受正整型参数。
- `void *mem_heap_lo(void)`：返回一个指向堆的第一个字节的指针。
- `void *mem_heap_hi(void)`：返回一个指向堆的最后一个字节的指针。
- `size_t mem_heapsize(void)`：返回当前堆的大小（字节）。
- `size_t mem_pagesize(void)`：返回系统的页大小（字节），Linux 上为 `4K`。

`mdriver.c` 程序测试你的 `mm.c` 的正确性、空间利用率以及吞吐量。该程序由一些 trace 文件控制，每一个 trace 文件包含一系列的分配、重分配以及释放指令，指示程序按一定顺序调用你的 `mm_malloc`， `mm_realloc` 以及 `mm_free`。我们使用同样的程序和 trace 文件对你上交的代码评分。`mdriver.c` 支持下列命令行参数：

- `-t <tracedir>`：在目录 `tracedir` 下查找默认 trace 文件，而不是在 `config.h` 中定义的默认目录中查找。
- `-f <tracefile>`：使用一个特定的 trace 文件测试。
- `-h`：打印命令行参数摘要。
- `-l`：在你的 `malloc` 包之外，运行并测试 libc 的 `malloc` 包。
- `-v`：详细输出。打印每一个 trace 文件的性能数据。
- `-V`：更详细的输出。在处理每一个 trace 文件时打印额外的诊断信息。在确定是哪一个 trace 文件导致你的 `malloc` 包失败时可能有用。

实验说明

堆一致性检查器

动态内存分配器难以正确且高效地实现，一个原因是其中包含了大量的没有类型的指针操作。你可能会发现编写一个堆检查器用来扫描堆并检查一致性很有帮助。

这些例子是堆检查器可能需要检查的东西：

- 空闲链表里的每一个块都被标记为空闲了吗？
- 是否有连续的空闲块因为某种原因没有被合并？
- 每一个空闲块都在空闲链表里吗？
- 空闲链表的指针指向的是有效的空闲块吗？
- 有重合的分配块吗？
- 在一个堆块中的指针指向了有效的堆地址吗？

你的堆检查器会由 `mm.c` 中的 `int mm_check(void)` 函数组成。它应该会检查你仔细考虑过的任何变量和一致性条件。当且仅当你的堆满足一致性的时候返回非零值。你既不需要局限于上述建议，也不需要完全检查所有建议。鼓励在 `mm_check` 失败时打印错误信息。

一致性检查器仅在你自己调试时使用。当你提交 `mm.c` 时，请确保移除了所有对于 `mm_check` 的调用，因为它们会影响你的吞吐量。

代码规范

- 不能更改 `mm.c` 定义的任何接口。
- 不能进行任何内存管理相关的库调用或者系统调用。
- 在 `mm.c` 中，不允许定义任何全局的或 `static` 的复合数据结构，例如数组、结构体、树或者链表。但是，可以定义全局的 scalar 变量，例如整数、浮点数和指针。
- 为了与 libc `malloc` 包保持一致，你的分配器必须总是返回 `8` 字节对齐的指针。

一些提示

- **使用 `mdriver -f` 选项。**在最初的开发过程中，使用小的 trace 文件可以简化调试和测试过程。有两个这样的 trace 文件（`short1,2-bal.rep`）可以用来做最初的调试。
- **使用 `mdriver -v` 和 `-V` 选项。** `-v` 选项可以针对每一个 trace 文件提供一个细节摘要；`-V` 选项还会指明每一个 trace 文件被读取的时间，这可以帮助你隔离错误。
- **使用 `gcc -g` 编译并使用调试器。**一个调试器可以帮助你隔离和识别越界的内存引用。
- **理解课本上 `malloc` 实现的每一行。**课本有一个基于隐式空闲链表的简单分配器的详细例子，可以以此为起点。在你理解这个简单的隐式链表分配器之前，不要开始编写你的分配器。
- **使用 C 预处理宏封装你的指针算术运算。**内存管理中使用的指针算术运算很容易出错，因为涉及很多必要的转换。你可以通过为你的指针操作编写宏来显著降低复杂度。
- **分阶段实现。**头 9 个 trace 包含 `malloc` 和 `free` 请求，最后的两个 trace 包含 `realloc`，`malloc`和 `free` 请求。我们推荐从让你的 `malloc` 和 `free` 能够正确并有效地通过前 9 个 trace 开始。在那之后再将注意力放到 `realloc` 的实现上。对于初学者，可以在现有的 `malloc` 和 `free` 实现上构建 `realloc`，但是如果想得到良好的性能，你需要构建一个独立的 `realloc`。
- **使用分析器。** `gprof` 等工具或许能够帮助你优化性能。
- **尽早开始！**

评分标准

你需要提交**实验报告**和**代码**。

- 代码（70%）
  - 正确性和性能
  - 代码风格
- 报告（30%）

性能评分依据两个指标：

- 空间利用率：使用的内存总量（分配但未释放的内存）与使用的堆大小的峰值比率。需要使其接近最优解 1。
- 吞吐量：每秒钟平均完成的操作数。

测试程序使用下述公式计算性能指标 P：

$$P = wU + (1 - w)\min(1, \frac{T}{T_{\text{libc}}})$$

其中 U 代表空间利用率，T 代表吞吐量，T_{libc} 代表测试系统上 libc `malloc` 在默认 trace 上的估计吞吐量（一个常数）。性能指标更偏向于空间利用率，因此 `w` 默认值为 `0.6`。各参数的具体值在最终评测时可能会发生变化。想要得到一个好的分数，需要在利用率和吞吐量之间取得平衡。