CS 213, Fall 2001
Cs213,2001 年秋
Malloc Lab: Writing a Dynamic Storage Allocator
Malloc 实验室: 编写一个动态存储分配器

Assigned: Friday Nov. 2, Due: Tuesday Nov. 20, 11:59PM
分配时间: 11 月 2 日，星期五，预定时间: 11 月 20 日，星期二，11:59 PM

Cory Williams (`cgw@andrew.cmu.edu`) is the lead person for this assignment.
Cory Williams (cgw@andrew.cmu.edu)是这项任务的负责人。

1 Introduction
1 引言

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc`, `free` and `realloc` routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.
在这个实验室里，你将为 c 程序编写一个动态存储分配器，也就是说，你自己的 malloc，free 和 realloc 例程。我们鼓励你创造性地探索设计空间，实现一个正确、高效、快速的分配器。

2 Logistics
2 物流

You may work in a group of up to two people. Any clarifications a nd revisions to the assignment will be posted on the course Web page.
你可以在一个最多两个人的团队中工作。任何对作业的澄清和修改都会在课程网页上公布。

3 Hand Out Instructions
3 分发说明书

SITE-SPECIFIC: Insert a paragraph here that explains how students should download the `malloclab-handout.tar` **file.**

SITE-SPECIFIC: 在这里插入一段解释学生应该如何下载 malloclab-handout.tar 文件。

Start by copying `malloclab-handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf malloclab-handout.tar`. This will cause a number of files to be unpacked into the directory. The only file you will b e modifying and handing in is `mm.c`. The `mdriver.c` program is a driver program that allows you to evaluate the performance of your solution. Use the command `make` to generate the driver code and run it with the command `./mdriver -V`. (The `-V` flag displays helpful summary information.)
首先，将 malloclab-handout.tar 复制到您计划在其中执行工作的受保护目录。然后给出命令：tar xvf malloclab-handout.tar。这将导致许多文件被解压到目录中。你唯一需要修改和提交的文件是 mm.c。C 程序是一个驱动程序，允许你评估你的解决方案的性能。使用 make 命令生成驱动程序代码并使用以下命令运行它。/mdriver-v (- v 标志显示有用的摘要信息

Looking at the file `mm.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**
查看文件 mm.c，您将注意到一个 c 结构团队，您应该在其中插入所请求的关于组成您的编程团队的一个或两个个人的识别信息。马上这样做，这样你就不会忘记。

When you have completed the lab, you will hand in only one file (`mm.c`), which contains your solution.
完成实验后，您将只交一个文件(mm.c)，其中包含您的解决方案。

4 How to Work on the Lab
4 How to Work on the Lab 4 如何在实验室工作

Your dynamic storage allocator will consist of the following four functions, which are declared in `mm.h` and defined in `mm.c`.
您的动态存储分配器将由以下四个函数组成，它们在 mm.h 中声明并在 mm.c 中定义。

```
int   mm_init(void);
Int mm _ init (void) ;
void *mm_malloc(size_t size);
Void * mm _ malloc (size _ t size) ;
void  mm_free(void *ptr);
Void mm _ free (void * ptr) ;
void *mm_realloc(void *ptr, size_t size);
Void * mm _ realloc (void * ptr, size _ t size) ;
```

The `mm.c` file we have given you implements the simplest but still funct ionally correct malloc package that we could think of. Using this as a starting place, modify these functions (and possibly define other private `static` functions), so that they obey the following semantics:
我们给你的 mm.c 文件实现了我们能想到的最简单但仍然可以正确运行的 malloc 包。以此为起点，修改这些函数(可能还要定义其他私有静态函数)，使它们遵循以下语义:

- `mm init`: Before calling `mm malloc` `mm realloc` or `mm free`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `mm init` to perform any necessary initializations, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise.

`Mm init`: 在调用 `mm malloc` `mm realloc` 或 `mm free` 之前，应用程序(即用于评估实现的跟踪驱动驱动程序)调用 `mm init` 来执行任何必要的初始化，比如分配初始堆区域。如果初始化过程中出现问题，返回值应该为 `-1`，否则为 `0`。

- `mm malloc`: The `mm malloc` routine returns a pointer to an allocated block payload of at least `size` bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk.

`Mm malloc`: `mm malloc` 例程返回一个指向至少大小字节的已分配块有效负载的指针。整个分配的块应该位于堆区域内，不应该与任何其他分配的块重叠。

  We will comparing your implementation to the version of `malloc` supplied in the standard C library (`libc`). Since the `libc` malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.
  我们将您的实现与标准 c 库(libc)中提供的 malloc 版本进行比较。因为 libc malloc 总是返回 8 字节对齐的有效负载指针，所以你的 malloc 实现也应该这样做，并且总是返回 8 字节对齐的指针。

- `mm free`: The `mm free` routine frees the block pointed to by `ptr`. It returns nothing. This rou-tine is only guaranteed to work when the passed pointer (`ptr`) was returned by an earlier call to `mm malloc` or `mm realloc` and has not yet been freed.

毫米自由：毫米自由例程释放 `ptr` 指向的块。它不会返回任何东西。只有当传递的指针(`ptr`)通过以前对 `mm malloc` 或 `mm realloc` 的调用返回且尚未释放时，此常规程序才能保证工作。

- mm realloc: The mm realloc routine returns a pointer to an allocated region of at least size bytes with the following constraints.

Mm realloc: mm realloc 例程使用以下约束返回指向至少大小字节的分配区域的指针。

  – if ptr is NULL, the call is equivalent to mm malloc(size);

  如果 ptr 为 NULL，则调用等价于 mm malloc (size) ;

  – if size is equal to zero, the call is equivalent to mm free(ptr);

  - 如果大小等于零，则调用等于毫米自由(ptr) ;

  – if ptr is not NULL, it must have been returned by an earlier call to mm malloc or mm realloc. The call to mm realloc changes the size of the memory block pointed to by ptr (the *old block*) to size bytes and returns the address of the new block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your imple-mentation, the amount of internal fragmentation in the old block, and the size of the realloc request.

  如果 ptr 不是 NULL，那么它一定是通过早期调用 mm malloc 或 mm realloc 返回的。调用 mm realloc 将 ptr (旧块)指向的内存块的大小改为字节大小，并返回新块的地址。请注意，新块的地址可能与旧块相同，也可能不同，这取决于您的实现心理、旧块中内部碎片的数量以及 realloc 请求的大小。

  The contents of the new block are the same as those of the old ptr block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new bl ock are identical to the first 8

  新块的内容与旧 ptr 块的内容相同，直到新旧大小的最小值为止。其他的都是未初始化的。例如，如果旧块是 8 个字节，新块是 12 个字节，那么新块的前 8 个字节与前 8 个字节相同

bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.
旧块的字节和最后 4 个字节是未初始化的。同样，如果旧块为 8 字节，新块为 4 字节，则新块的内容与旧块的前 4 字节相同。

These semantics match the the semantics of the corresponding `libc malloc`, `realloc`, and `free` rou-tines. Type `man malloc` to the shell for complete documentation.
这些语义与相应的 libc malloc、 realloc 和 free routines 的语义相匹配。在 shell 中输入 man malloc 以获得完整的文档。

5 Heap Consistency Checker

5 Heap Consistency Checker 5 堆一致性检查器

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of u ntyped pointer manipulation. You will find it very helpful to write a heap checker that scans the heap and checks it for consistency.
众所周知，动态内存分配器在正确高效地编程方面非常棘手。它们很难正确编程，因为它们涉及大量未键入的指针操作。你会发现写一个堆检查器来扫描堆并检查它的一致性是非常有帮助的。

Some examples of what a heap checker might check are:
堆检查器可以检查的一些例子如下:

- Is every block in the free list marked as free?
  免费列表中的每个块都标记为免费吗？
- Are there any contiguous free blocks that somehow escaped coalescing?
  是否有任何相邻的空闲块以某种方式逃脱了合并？
- Is every free block actually in the free list?
  所有的免费街区都在免费名单上吗？
- Do the pointers in the free list point to valid free blocks?
  空闲列表中的指针是否指向有效的空闲块？
- Do any allocated blocks overlap?
  任何分配的块是否重叠？
- Do the pointers in a heap block point to valid heap addresses?
  堆块中的指针是否指向有效的堆地址？

Your heap checker will consist of the function `int mm check(void)` in `mm.c`. It will check any invari-ants or consistency conditions you consider prudent. It returns a nonzero value if and only if your heap is consistent. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when `mm check` fails.

堆检查器将由 mm.c 中的函数 int mm check (void)组成。它会检查任何你认为谨慎的不变蚂蚁或一致性条件。当且仅当堆是一致的时候，它返回一个非零值。你并不局限于列出的建议，也不需要检查所有的建议。当 mm 检查失败时，鼓励你打印出错误信息。

This consistency checker is for your own debugging during development. When you submit `mm.c`, make sure to remove any calls to `mm check` as they will slow down your throughput. Style points will be given for your `mm check` function. Make sure to put in comments and document what you are checking.
这个一致性检查器用于您自己在开发过程中的调试。当你提交 mm.c 时，一定要删除对 mm check 的任何调用，因为它们会降低你的吞吐量。你的 mm 检查功能会被赋予样式点。确保输入评论并记录你正在检查的内容。

6 Support Routines
6 支援例行公事

The memlib.c package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in `memlib.c`:
C 包模拟了动态内存分配器的内存系统。你可以在 memlib.c 中调用以下函数:

- `void *mem sbrk(int incr)`: Expands the heap by `incr` bytes, where `incr` is a positive non-zero integer and returns a generic pointer to the first by te of the newly allocated heap area. The semantics are identical to the Unix `sbrk` function, except that `mem sbrk` accepts only a positive non-zero integer argument.
`Void * mem sbrk (int incr)` ：通过 `incr` 字节扩展堆，其中 `incr` 是一个正的非零整数，并返回一个泛型指针，指向新分配的堆区域的第一个 `by te`。其语义与 `Unix sbrk` 函数相同，只是 `mem sbrk` 只接受一个正的非零整数参数。

- `void *mem heap lo(void)`: Returns a generic pointer to the first byte in the heap.

  Void * mem heap lo (void) : 返回一个指向堆中第一个字节的通用指针。

- `void *mem heap hi(void)`: Returns a generic pointer to the last byte in the heap.

  Void * mem heap hi (void) : 返回指向堆中最后一个字节的泛型指针。

- `size t mem heapsize(void)`: Returns the current size of the heap in bytes.

  Size t mem heapsize (void) : 返回堆的当前大小(以字节为单位)。

- `size t mem pagesize(void)`: Returns the system's page size in bytes (4K on Linux systems).

  Size t mem pagesize (void) : 返回系统的页面大小(以字节为单位)(Linux 系统上为 4k)。

7 The Trace-driven Driver Program
追踪驱动的驱动程序

The driver program `mdriver.c` in the `malloclab-handout.tar` distribution tests your `mm.c` pack-
Tar 发行版中的驱动程序 mdriver.c 测试 mm.c 包-
age for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files*
这个驱动程序由一组跟踪文件控制
that are included in the `malloclab-handout.tar` distribution. Each trace file contains a sequence of
每个跟踪文件包含一个序列
allocate, reallocate, and free directions that instruct the driver to call your `mm malloc`, `mm realloc`, and
分配，重新分配，以及指示驱动程序调用 mm malloc，mm realloc，和
`mm free` routines in some sequence. The driver and the trace files are t he same ones we will use when we grade your handin `mm.c` file.
以某种顺序排列的自由程序。驱动程序和跟踪文件是我们在给你的 handin mm.c 文件打分时使用的相同的文件。

The driver `mdriver.c` accepts the following command line arguments:
驱动程序 mdriver.c 接受以下命令行参数:

- `-t <tracedir>`: Look for the default trace files in directory `tracedir` instead of the default directory defined in `config.h`.

  - t < tracedir > : 查找 tracedir 目录中的默认跟踪文件，而不是 config.h 中定义的默认目录。

- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of trace-files.

  使用一个特定的跟踪文件代替默认的跟踪文件集进行测试。

- `-h`: Print a summary of the command line arguments.

  - h: 打印命令行参数的摘要。

- `-l`: Run and measure `libc` malloc in addition to the student's malloc package.

  L: 除了学生的 malloc 包之外，运行和测量 libc malloc。

- `-v`: Verbose output. Print a performance breakdown for each tracefile in a compact table.

- `v`: 详细输出。在一个紧凑的表中打印每个跟踪文件的性能细目。

- `-V`: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is c ausing your malloc package to fail.
更详细的输出。在处理每个跟踪文件时打印额外的诊断信息。在调试过程中，用于确定哪个跟踪文件是 `c`，从而导致 `malloc` 包失败。

8 Programming Rules
编程规则

- You should not change any of the interfaces in `mm.c`.
您不应该更改 mm.c 中的任何接口。

- You should not invoke any memory-management related library calls or system calls. This excludes
您不应该调用任何与内存管理相关的库调用或系统调用
the use of `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` or any variants of these calls in your code.
在你的代码中使用 malloc，calloc，free，realloc，sbrk，brk 或者这些调用的任何变体。

- You are not allowed to define any global or `static` compound data structures such as arrays, structs, trees, or lists in your `mm.c` program. However, you *are* allowed to declare global scalar variables such as integers, floats, and pointers in `mm.c`.
不允许在 mm.c 程序中定义任何全局或静态复合数据结构，如数组、结构、树或列表。但是，你可以在 mm.c 中声明全局标量变量，如整数、浮点数和指针。

- For consistency with the `libc malloc` package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

为了与 libc malloc 包保持一致，它返回按 8 字节边界对齐的块，分配器必须始终返回按 8 字节边界对齐的指针。驱动程序会强制执行这个要求。

9 Evaluation
9 评估

You will receive **zero points** if you break any of the rules or your code is buggy and crashes the driver.
如果你违反了任何一条规则，或者你的代码有问题导致驱动程序崩溃，你将得到零分。
Otherwise, your grade will be calculated as follows:
否则，你的成绩将按以下方式计算:

- *Correctness (20 points).* You will receive full points if your solution passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace.

正确性(20 分)。如果你的解决方案通过了驱动程序的正确性测试，你将获得满分。你将获得每个正确跟踪的部分学分。

- *Performance (35 points).* Two performance metrics will be used to evaluate your solution:

性能(35 分)。两个性能指标将用于评估你的解决方案:

  – *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `mm malloc` or `mm realloc` but not yet freed via `mm free`) and the size of the heap used by your allocator. The optimal ratio equals to 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.

  - 空间利用率: 驱动程序使用的内存总量(即通过 mm malloc 或 mm realloc 分配但尚未通过 mm free 释放)与分配程序使用的堆大小之间的峰值比率。最佳比率等于 1。你应该找到好的政策来尽量减少分裂，以便使这个比率尽可能接近最优。

  – *Throughput*: The average number of operations completed per second.

  - 吞吐量: 平均每秒完成的操作数。

The driver program summarizes the performance of your allocator by computing a *performance index*,
驱动程序通过计算性能指数来总结分配器的性能,
P, which is a weighted sum of the space utilization and throughput
，它是空间利用率和吞吐量的加权和

T

TLI

BC

TLI

P = WU + (1 − W) MIN 1,

p = WU + (1-w) MIN 1,

BC

where U is your space utilization, T is your throughput, and T_{LIBC} is the estimated throughput of `libc` malloc on your system on the default traces.[1] The performance index favors space utilization over throughput, with a default of $w = 0.6$.
其中 u 表示空间利用率，t 表示吞吐量，TLIBC 表示默认轨迹上系统上 libc malloc 的估计吞吐量。1 性能指数更倾向于空间利用率而不是吞吐量，默认值 w = 0.6。

Observing that both memory and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both memory utilization and throughput. Ideally, the performance index will reach $P = w + (1 - w) = 1$ or 100%. Since each metric will contribute at most w and $1 - w$ to the performance index, respectively, you should not go to extremes to optimize either the memory utilization or the throughput only. To receive a good score, you must achieve a balance between utilization and throughput.
考虑到内存和 CPU 周期都是昂贵的系统资源，我们采用这个公式来鼓励均衡地优化内存利用率和吞吐量。理想情况下，性能指数将达到 p = w + (1-w) = 1 或 100% 。由于每个度量指标对性能指数的贡献最多为 w 和 1-w，因此不应该极端地优化内存利用率或仅优化吞吐量。为了得到一个好的分数，你必须在利用率和吞吐量之间取得平衡。

- Style (10 points).

Style (10 分)。

  – Your code should be decomposed into functions and use as few global variables as possible.
  - 你的代码应该被分解成函数，并尽可能少地使用全局变量。

  – Your code should begin with a header comment that describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. each function should be preceeded by a header comment that describes what the function does.
  - 您的代码应该以头注释开始，其中描述您的空闲和分配块的结构、空闲列表的组织以及您的分配器如何操作空闲列表。每个函数前面都应该有一个描述函数功能的头注释。

[1] The value for T_{LIBC} is a constant in the driver (600 Kops/s) that your instructor established when they configured the program.
1 TLIBC 的值是指导者在配置程序时建立的驱动程序中的一个常量(600kops/s)。

– Each subroutine should have a header comment that describes what it does and how it does it.
每个子程序都应该有一个头注释，描述它做什么和如何做。

– Your heap consistency checker `mm check` should be thorough and well-documented.
- 您的堆一致性检查毫米检查应该是彻底的和良好的文件。

You will be awarded 5 points for a good heap consistency checker and 5 points for good program structure and comments.
一个好的堆一致性检查器会给你 5 分，一个好的程序结构和注释会给你 5 分。

10 Handin Instructions
10 汉字指示

SITE-SPECIFIC: Insert a paragraph here that explains how the students should hand in their solution `mm.c` files.
SITE-SPECIFIC: 在这里插入一个段落，解释学生应该如何提交他们的解决方案 mm.c 文件。

11 Hints
提示

• *Use the* `mdriver -f` *option.* During initial development, using tiny trace files will simp lify debug-ging and testing. We have included two such trace files (`short1,2-bal.rep`) that you can use for initial debugging.
使用 mdriver-f 选项。在最初的开发过程中，使用微小的跟踪文件可以简化调试和测试。我们已经包含了两个这样的跟踪文件(short1,2-bal。Rep)，可用于初始调试。

• *Use the* `mdriver -v` *and* `-V` *options.* The `-v` option will give you a detailed summary for each trace file. The `-V` will also indicate when each trace file is read, which will hel p you isolate errors.
使用 mdriver-v 和-v 选项。V 选项会为每个跟踪文件提供一个详细的摘要。V 还会显示每个跟踪文件的读取时间，这将帮助你隔离错误。

• *Compile with* `gcc -g` *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references.
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超出范围的内存引用。

• *Understand every line of the malloc implementation in the textbook.* The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
理解教科书中 malloc 实现的每一行。教科书中有一个基于隐式空闲列表的简单分配器的详细示例。Use 这是一个出发点。在你了解所有关于简单隐式 list 分配器的知识之前，不要开始处理你的分配器。

• *Encapsulate your pointer arithmetic in C preprocessor macros.* Pointer arithmetic in memory man-agers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the text for examples.

将你的指针算法封装在 c 预处理器宏中。指针算法在内存管理器是混乱和容易出错的，因为所有的转换都是必要的。你可以通过为你的指针操作编写宏来显著降低复杂度。参见文本中的例子。

- *Do your implementation in stages.* The first 9 traces contain requests to `malloc` and `free`. The last 2 traces contain requests for `realloc`, `malloc`, and `free`. We recommend that you start by getting your `malloc` and `free` routines working correctly and efficiently on the first 9 trac es. Only then should you turn your attention to the `realloc` implementation. For starters, build `realloc` on top of your existing `malloc` and `free` implementations. But to get really good performance, you will need to build a stand-alone `realloc`.

分阶段进行实现。前 9 个跟踪包含对 malloc 和 free 的请求。最后两个跟踪包含 realloc、malloc 和 free 的请求。我们建议你从让 malloc 和 free 例程在前 9 个跟踪中正确有效地工作开始。只有这样，你才能将注意力转移到 realloc 的实现上。首先，在现有的 malloc 和 free 实现的基础上构建 realloc。但是为了获得真正好的性能，你需要构建一个独立的 realloc。

- *Use a profiler.* You may find the `gprof` tool helpful for optimizing performance.

使用剖析器。你可能会发现 gprof 工具对优化性能很有帮助。

- *Start early!* It is possible to write an efficient malloc package with a few p ages of code. However, we can guarantee that it will be some of the most difficult and sop histicated code you have written so far in your career. So start early, and good luck!

早点出发！用几页代码编写一个高效的 malloc 包是可能的。然而，我们可以保证这将是你迄今为止写过的最困难、最复杂的代码。所以早点开始吧，祝你好运！