15-213/18-213/15-513, Fall 2020
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: Mon, November 02, 2019

This lab requires submitting two versions of your code: one as an initial checkpoint, and the second as your final version. The dates and weights for your course grade are indicated in the following table:

Version	Due Date	Max. Grace Days	Last Date	Weight in Course
Checkpoint	Mon, November 09, 11:59pm	0	Mon, Novermber 09, 11:59pm	3%
Final	Mon, November 16, 11:59pm	2	Wed, November 18, 11:59pm	9%

1 Introduction

In this lab you will be writing a *general purpose* dynamic storage allocator for C programs; that is, your own version of the `malloc`, `free`, `realloc`, and `calloc` functions. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient, and fast.

In order to get the most out of this lab, we *strongly* encourage you to start early. The total time you spend designing and debugging can easily eclipse the time you spend coding.

Bugs can be especially pernicious and difficult to track down in an allocator, and you will probably spend a significant amount of time debugging your code. *Buggy code will not get any credit. You cannot afford to get score of 0 for this assignment.*

This lab has been heavily revised from previous versions. Do not rely on advice or information you may find on the Web or from people who have done this lab before. Be sure to read all of the documentation carefully.

2 Logistics

This is an individual project. You should do this lab on one of the Shark machines.

3 Hand Out Instructions

Start by downloading `malloclab-handout.tar` from the *Malloc Lab* (not Malloc Lab Checkpoint) assignment on Autolab 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 be turn in is `mm.c`*, which contains your solution.* The provided code allows you to locally evaluate your implementations in the same way that will be used to generate a score for your submissions. Using the command `make` will generate two *driver* programs: `mdriver` and `mdriver-emulate`. Use these to evaluate the correctness and performance of your implementations.

4 Required Functions

Your dynamic storage allocator will implement the following functions, declared in `mm.h` and defined in `mm.c`:

```
bool  mm_init(void);
void *malloc(size_t size);
void  free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
bool  mm_checkheap(int);
```

We provide you three versions of memory allocators:

`mm.c`**:** A placeholder that compiles correctly, but does not run.

`mm-naive.c`**:** A functional implementation that runs fast but gets very poor utilization, because it never reuses any blocks of memory.

`mm-baseline.c`**:** A fully-functional implicit-list allocator. We recommend that you use this code as your starting point.

Your allocator must run correctly on a 64-bit machine. It must support a full 64-bit address space, even though current implementations of x86-64 machines support only a 48-bit address space. The driver `mdriver-emulate` will evaluate your program's correctness using benchmark traces that require the use of a full 64-bit address space.

Your submitted `mm.c` must implement the following functions:

- `mm_init`: Performs any necessary initializations, such as allocating the initial heap area. The return value should be `false` if there was a problem in performing the initialization, `true` otherwise. You must reinitialize all of your data structures in this function, because the drivers call your `mm_init` function every time they begin a new trace to reset to an empty heap.

- `malloc`: The `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 block.

  Your `malloc` implementation must always return 16-byte aligned pointers.

- `free`: The `free` routine frees the block pointed to by `ptr`. It returns nothing. This routine is only guaranteed to work when the passed pointer was returned by an earlier call to `malloc`, `calloc`, or `realloc` and has not yet been freed. `free(NULL)` has no effect.

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

  - if `ptr` is `NULL`, the call is equivalent to `malloc(size);`
  - if `size` is equal to zero, the call is equivalent to `free(ptr)` and should return `NULL`;
  - if `ptr` is not `NULL`, it must have been returned by an earlier call to `malloc` or `realloc` and not yet have been freed. The call to `realloc` takes an existing block of memory, pointed to by `ptr` — the *old block*. Upon return, the contents of the new block should be the same as those of the old block, up to the minimum of the old and new sizes. Everything else is uninitialized. Achieving this involves either copying the old bytes to a newly allocated region or reusing the existing region.

    For example, if the old block is 16 bytes and the new block is 24 bytes, then the first 16 bytes of the new block are identical to the first 16 bytes of the old block and the last 8 bytes are uninitialized. Similarly, if the old block is 16 bytes and the new block is 8 bytes, then the contents of the new block are identical to the first 8 bytes of the old block.

    The function returns a pointer to the resulting region. The return value might be the same as the old block—perhaps there is free space after the old block, or `size` is smaller than the old block size—or it might be different. If the call to `realloc` does not fail and the returned address is different than the address passed in, the old block has been freed and should not be used, freed, or passed to `realloc` again.

  Hint: Your `realloc` implementation will have only minimal impact on measured throughput or utilization. A correct, simple implementation will suffice.

- `calloc`: Allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero before returning.

  Note: Your `calloc` will not be graded on throughput or performance. A correct, simple implementation will suffice.

- `mm_checkheap`: The `mm_checkheap` function (the *heap consistency checker*, or simply *heap checker*) scans the heap and checks it for possible errors (e.g., by making sure the headers and footers of each block are identical). Your heap checker should run silently until it detects some error in the heap. Then, and only then, should it print a message and return `false`. If it finds no errors, it should return `true`. It is very important that your heap checker run silently; otherwise, it will produce too much output to be useful on the large traces.

A quality heap checker is essential for debugging your `malloc` implementation. Many `malloc` bugs are too subtle to debug using conventional `gdb` techniques. The only effective technique for some of these bugs is to use a heap consistency checker. When you encounter a bug, you can isolate it with repeated calls to the consistency checker until you find the operation that corrupted your heap. Because of the importance of the consistency checker, it will be graded. If you ask members of the course staff for help, the *first thing we will do is ask to see your checkheap function*, so please write this function before coming to see us!

The `mm_checkheap` function takes a single integer argument that you can use any way you want. One very useful technique is to use this argument to pass in the line number of the call site:

```
mm_checkheap(__LINE__);
```

If `mm_checkheap` detects a problem with the heap, it can print the line number where `mm_checkheap` was called, which allows you to call `mm_checkheap` at numerous places in your code while you are debugging.

The semantics for `malloc`, `realloc`, `calloc`, and `free` match those of the corresponding `libc` routines. Type `man malloc` to the shell for complete documentation.

5 Support Routines

The `memlib.c` package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in `memlib.c`:

- `void *mem_sbrk(intptr_t incr)`: Expands the heap by `incr` bytes, where `incr` is a non-negative integer, and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix `sbrk` function, except that `mem_sbrk` will fail for negative arguments. (Data type `intptr_t` is defined to be a signed integer large enough to hold a pointer. On our machines it is 64-bits long.)
- `void *mem_heap_lo(void)`: Returns a generic pointer to the first byte in the heap.
- `void *mem_heap_hi(void)`: Returns a generic pointer to the last byte in the heap.
- `size_t mem_heapsize(void)`: Returns the current size of the heap in bytes.

You are also allowed to use the following `libc` library functions: `memcpy`, `memset`, `printf`, `fprintf`, and `sprintf`. Other than these functions and the support routines, your `mm.c` code may not call any externally-defined function.

6 The Trace-driven Driver Programs

The two driver programs generated when you run `make` test your `mm.c` code for correctness, space utilization, and throughput. These driver programs are controlled by a set of *trace files* that are included in the `traces` subdirectory of the `malloclab-handout.tar` distribution. Each trace file contains a sequence of commands that instruct the driver to call your `malloc`, `realloc`, and `free` routines in some sequence. The drivers and the trace files are the same ones we will use when we grade your handin `mm.c` file.

When the driver programs are run, they will run each trace file multiple times: once to make sure your implementation is correct, once to determine the space utilization, and between 3 and 20 times to determine the throughput.

The drivers accept the following command-line arguments. The normal operation is to run it with no arguments, but you may find it useful to use the arguments during development.

- `-p`: Apply the scoring standards for the checkpoint, rather than for the final submission.
- `-t` *`tracedir`*: Look for the default trace files in directory *`tracedir`* instead of the default directory `traces`
- `-f` *`tracefile`*: Use one particular *`tracefile`* instead of the default set of tracefiles for testing correctness, utilization, and throughput.
- `-c` *`tracefile`*: Run a particular *`tracefile`* exactly once, testing only for correctness. This option is extremely useful if you want to print out debugging messages.
- `-h`: Print a summary of the possible command line arguments.
- `-l`: Run and measure the `libc` version of `malloc` in addition to your `malloc` package. This is interesting if you want to see how fast a real `malloc` package runs.
- `-V`: Verbose output. Print additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your `malloc` package to fail.
- `-v` *`level`*: This optional feature lets you manually set your verbosity level to a particular integer. Verbosity levels 0–2 are supported.
- `-d` *`level`*: At debug level 0, very little validity checking is done. This is useful if you are mostly done but just tweaking performance.

  At debug level 1, every array the driver allocates is filled with random bytes. When the array is freed or reallocated, the driver checks to make sure the bytes have not been changed. This is the default.

  At debug level 2, every time any operation is done, all of the allocated arrays are checked to make sure they still hold their randomly assigned bytes, and your implementation of `mm_checkheap` is called. This mode is slow, but it can help identify the exact point at which an error gets injected.
- `-D`: Equivalent to `-d2`.

- `-s` *s*: Time out after *s* seconds. The default is to never time out.
- `-T`: Print the output in a tabular form. This can be useful if you want to convert the results into a spreadsheet for further analysis.

For most of your testing, you should use the `mdriver` program. It checks the correctness, utilization, and throughput of the standard benchmark traces. It applies all of the tests required for the checkpoint. The `mdriver-emulate` program uses special compilation and memory-management techniques to allow testing your program with a heap making use of the full, 64-bit address space. In addition to the standard benchmark traces, it will run a set of *giant* traces with very large allocation requests. If your final submission fails to pass any of its checks, you will be given a penalty of 30 points.

7 Programming Rules

- You are writing a general purpose allocator. You may not solve specifically for any of the traces—we will be checking for this. Any allocator that attempts to explicitly determine which trace is running (e.g., by executing a sequence of tests at the beginning of the trace) and change its behavior based on that trace's pattern of allocations will receive a penalty of 20 points.
- You should not change any of the interfaces in `mm.h`, and your program must compile with the provided Makefile. However, we strongly encourage you to use `static` helper functions in `mm.c` to break up your code into small, easy-to-understand segments.
- You are not allowed to define any large global data structures such as large arrays, trees, or lists in your `mm.c` program. However, you *are* allowed to declare small global arrays, structs and scalar variables such as integers, floats, and pointers in `mm.c`. In total, your global data should sum to at most 128 bytes. Global values defined with the `const` qualifier are not counted in the 128 bytes.

  The reason for this restriction is that the driver cannot account for such global variables in its memory utilization measure. If you need space for large data structures, you can put them at the beginning of the heap.
- The use of macro definitions (using `#define`) in your code is restricted to the following:

  1. Macros with names beginning with the prefix "`dbg_`" that are used for debugging purposes only. See, for example, the debugging macros defined in `mm-baseline.c`. You may create other ones, but they must be disabled in any version of your code submitted to Autolab.
  2. Definitions of constants. These definitions must not have any parameters.

  Explanation: It is traditional in C programming to use macros instead of function definitions in an attempt to improve program performance. This practice is obsolete. Modern compilers (when optimization is enabled) perform *inline substitution* of small functions, eliminating any inefficiencies due to the use of functions rather than macros. In addition, functions provide better type checking and (when optimization is disabled) enable better debugging support.

Here are some examples of allowed and disallowed macro definitions:

`#define DEBUG 1`	OK	Defines a constant
`#define CHUNKSIZE (1<<12)`	OK	Defines a constant
`#define WSIZE sizeof(uint64_t)`	OK	Defines a constant
`#define dbg_printf(...) printf(__VA_ARGS__)`	OK	Debugging support
`#define GET(p) (*(unsigned int *)(p))`	Not OK	Has parameters
`#define PACK(size, alloc) ((size)|(alloc))`	Not OK	Has parameters

When you run `make`, it will run a program that checks for disallowed macro definitions in your code. This checker is overly strict—it cannot determine when a macro definition is embedded in a comment or in some part of the code that has been disabled by conditional-compilation directives. Nonetheless, your code must pass this checker without any warning messages.

- The code shown in the textbook (Section 9.9.12, and available from the CS:APP website) is a useful source of inspiration for the lab, but it does not meet the required coding standards. It does not handle 64-bit allocations, it makes extensive use of macros instead of functions, and it relies heavily on low-level pointer arithmetic. Similarly, the code shown in K&R does not satisfy the coding requirements. You should use the provided code `mm-baseline.c` as your starting point.

- It is okay to look at any *high-level* descriptions of algorithms found in the textbook or elsewhere, but it is *not* acceptable to copy or look at any code of `malloc` implementations found online or in other sources, except for the allocators described in the textbook, in K&R, or as part of the provided code.

- It is okay to copy code for useful generic data structures and algorithms (e.g., hash tables, search trees, priority queues, etc.) from Wikipedia and other repositories. (This code must not have already been customized as part of a memory allocator.) You must include (as a comment) an attribution of the origins of any borrowed code.

- Your allocator must always return pointers that are aligned to 16-byte boundaries. The driver will check this requirement.

- Your code *must* compile without warnings. Warnings often point to subtle errors in your code; whenever you get a warning, you should double-check the corresponding line to see if the code is really doing what you intended. If it is, then you should eliminate the warning by tweaking the code (for instance, one common type of warning can be eliminated by adding a type-cast where a value is being converted from one type of pointer to another). We have added flags in the Makefile to force your code to be error-free. You may remove those flags during development if you wish, but please realize that we will be grading you with those flags activated.

- Your code will be compiled with the `clang` compiler, rather than `gcc`. This compiler is distributed by the LLVM Project (`llvm.org`). Their compiler infrastructure provides features that have enabled us to implement the 64-bit address emulation techniques of `mdriver-emulate`. For the most part, `clang` is compatible with `gcc`, except that it generates different error and warning messages.

8 Evaluation

Malloc Lab is worth 12% of your final grade in the course: 3% for the checkpoint and 9% for the final version. Assuming your solution passes all correctness tests, and the graders do not detect any errors in your source code, two metrics are used to evaluate performance:

- *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `malloc` but not yet freed via `free`) and the size of the heap used by your allocator. The optimal (but unachievable) ratio equals 100%. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
- *Throughput*: The average number of operations completed per second, expressed in *kilo-operations per second* or KOPS. A trace that takes T seconds to perform n operations will have a throughput of $n/(1000 \cdot T)$ KOPS.

8.1 Performance Points

Observing that both memory and CPU cycles are expensive system resources, we combine these two measures into a single performance index P, with $0 \leq P \leq 100$, computed as a weighted sum of the space utilization and throughput:

$$P\left(U,T\right) = 100\left(w \cdot \mathit{Threshold}\left(\frac{U - U_{min}}{U_{max} - U_{min}}\right) + (1-w) \cdot \mathit{Threshold}\left(\frac{T - T_{min}}{T_{max} - T_{min}}\right)\right)$$

where U is the space utilization (averaged across the traces), and T is the throughput (averaged across the traces, but weighted by the number of operations in each trace). U_{max} and T_{max} are the estimated space utilization and throughput of a well-optimized `malloc` package, and U_{min} are T_{min} are minimum space utilization and throughput values, below which you will receive 0 points. The weight w defines the relative weighting of utilization versus performance in the score. Function *Threshold* is defined as

$$\mathit{Threshold}(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x \leq 1 \\ 1, & x > 1. \end{cases}$$

The values of the parameters differ for the checkpoint and the final versions as follows:

Version	w	U_{min}	U_{max}	T_{min}	T_{max}
Checkpoint	0.2	55%	58%	2,000	12,000
Final	0.6	55%	74%	5,000	15,000

The `mdriver` driver program will assign a performance index of 0 if it detects an error in any of the traces. In addition, we will run a separate test of the driver `mdriver-emulate` on your final submission. If it detects an error, we will deduct 30 points from P.

The `traces` subdirectory contains a number of traces. Some of them are short traces that do not count toward your memory utilization or throughput but are useful for detecting errors and for debugging. In the driver's output, you will see these marked without a '*' next to them. The traces that count towards both your memory utilization and throughput are marked with a '*' in the output of `mdriver`.

Note: You should be able to get reliable performance measurements running on a Shark machine. When you submit your code via Autolab, you may find the throughput measurements do not match those of a Shark machine, because Autolab runs on a variety of different machines. As part of the grading process, we will run your code on a lightly-loaded Shark machine to determine its official throughput performance. It should be at least as good as the throughput you will get when running the program yourself. *Do not trust the throughtput results you get from Autolab. Evaluate throughput by running* `mdriver` *on a Shark machine.*

8.2 Additional Points

Your score for the checkpoint will be determined solely by the performance index P computed by running `mdriver` with command-line option `-p`.

Your score for the final version will equal P (as computed by `mdriver` minus any penalty due to failing a test with `mdriver-emulate`), plus up to 20 additional points as follows:

- *Heap Consistency Checker (10 points).* Ten points will be awarded based on the quality of your implementation of `mm_checkheap`. It is up to your discretion how thorough you want your heap checker to be. The more the checker tests, the more valuable it will be as a debugging tool. However, to receive full credit for this part, we require that you check *all* of the invariants of your data structures. Some examples of what your heap checker should check are provided below.
  - Checking the heap (implicit list, explicit list, segregated list):
    - Check epilogue and prologue blocks.
    - Check each block's address alignment.
    - Check heap boundaries.
    - Check each block's header and footer: size (minimum size, alignment), previous/next allocate/free bit consistency, header and footer matching each other.
    - Check coalescing: no two consecutive free blocks in the heap.
  - Checking the free list (explicit list, segregated list):
    - All next/previous pointers are consistent (if A's next pointer points to B, B's previous pointer should point to A).
    - All free list pointers are between `mem_heap_lo()` and `mem_heap_high()`.
    - Count free blocks by iterating through every block and traversing free list by pointers and see if they match.
    - All blocks in each list bucket fall within bucket size range (segregated list).
- *Style (10 points).* Your code should follow the Style Guidelines posted on the course Web site. In addition:

- Your code should be decomposed into functions and use as few global variables as possible. You should use static functions and declared structs and unions to minimize pointer arithmetic and to isolate it to a few places.
- You should avoid sprinkling your code with numeric constants. Instead, use declarations via `#define` or static constants. Try, as much as possible, to use C data types, and the operators `sizeof` and `offsetof` to define the sizes of various fields and offsets, rather than using fixed numeric values.
- Your `mm.c` file **must** begin with a header comment that gives an overview of the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list.
- In addition to this overview header comment, each function **must** be preceded by a header comment that describes what the function does.
- You will want to use inline comments to explain code flow or code that is tricky.

Study the code in `mm-baseline.c` as an example of the desired coding style.

9 Handin Instructions

Make sure you have included your name and Andrew ID in the header comment of `mm.c`. Make your code does not print anything during normal operation, and that all debugging macros have been disabled.

Hand in your `mm.c` file by uploading it to Autolab. You may submit your solution as many times as you wish until the due date.

Only the last version you submit will be graded.

For this lab, you must upload your code for the results to appear on the class status page. Remember that the throughput value shown on the status page may be inaccurate.

10 Useful Tips

- We have included several macros to check assertions (see the Linux man page for `assert`) and to print information when debugging is enabled. As described earlier, they all have names that start with "`dbg_`." These let you keep some of your debugging code in your programs, without incurring any performance penalty when debugging is not enabled. Think about a strategy for providing useful debugging information, while keeping your code readable.
- *Use the drivers' `-c` and `-f` options.* During initial development, using short trace files will simplify debugging and testing.
- *Use the drivers' `-V` option.* The `-V` option will also indicate when each trace file is processed, which will help you isolate errors.
- *Use the drivers' `-D` option.* This does a lot of checking to quickly find errors.

- *Use a debugger.* A debugger will help you isolate and identify out-of-bounds memory references. Modify the Makefile to set the parameter `COPT` to `-O0` (big-Oh, numeral zero) to disable optimizations and then recompile your code. But, do not forget to restore the Makefile to the original, and then recompile the code (with the command "`make clean ; make`"), when doing performance testing. Also, you'll find that `mdriver-emulate` will give errors when the code has been generated with optimization disabled.

- *Use* `gdb`*'s* `watch` *command* to find out what changed some value you did not expect to have changed.

- Don't try to use `gdb` to examine the heap when running `mdriver-emulate`. It cannot account for our memory emulation tricks. On the other hand, you can use `printf`'s in your code to print the memory contents.

- *Encapsulate your pointer arithmetic in functions.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can significantly reduce the complexity by writing functions for your pointer operations. See the code in `mm-baseline.c` for examples. **Remember:** you are not allowed to define macros—the code in the textbook does not meet our coding standards for this lab.

- *Your allocator must work for a 64-bit address space.* The `mdriver-emulate` will specifically test this capability. You should allocate a full eight bytes for all of your pointers and size fields. Make sure you do not inadvertently invoke 32-bit arithmetic with an expression such as `1<<32`, rather than `1L<<32`.

- *Use your heap consistency checker.* We are assigning ten points to the `mm_checkheap` function in your final submission for a reason. A good heap consistency checker will save you hours and hours when debugging your `malloc` package. You can use your heap checker to find out where exactly things are going wrong in your implementation (hopefully not in too many places!). Make sure that your heap checker is detailed. To be useful, your heap checker should only produce output when it detects an error. Every time you change your implementation, one of the first things you should do is think about how your `mm_checkheap` will change, what sort of tests need to be performed, and so on. Although we will not examine the heap checker you implement for the checkpoint, you should have a good implementation of it right from the start. Do not even think about asking for debugging help from any of the course staff until you have implemented and tried using your heap checker.

- *Keep backups.* Whenever you have a working allocator and are considering making changes to it, keep a backup copy of the last working version. It is very common to make changes that inadvertently break the code and then have trouble undoing them.

- *Versioning your implementation.* You may find it useful to manage a couple of different versions of implementation (e.g., explicit list, segregated list) during the assignment. Since `mdriver` looks for `mm.c`, creating a symbolic link between files is useful in this case. For example, you can create a symbolic link between `mm.c` and your implementation such as `mm-explicit.c` with command line `ln -s mm-explicit mm.c`. Now would also be an great time to learn an industrial-strength version control system like Git (`http://git-scm.com`).

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

11 Strategic Advice

You must design algorithms and data structures for managing free blocks that achieve the right balance of space utilization and speed. This involves a trade-off—it is easy to write a fast allocator by allocating blocks and never freeing them, or a high-utilization allocator that carefully packs blocks as tightly as possible. You must seek to minimize wasted space while also making your program run fast.

As described in the textbook and the lectures, utilization is reduced below 100% due to *fragmentation*, taking two different forms:

- *External fragmentation:* Unused space between allocated blocks or at the end of the heap
- *Internal fragmentation:* Space within an allocated block that cannot be used for storing data, because it is required for some of the manager's data structures (e.g., headers, footers, and free-list pointers), or because extra bytes must be allocated to meet alignment or minimum block size requirements

To reduce external fragmentation, you will want to implement good block placement heuristics. To reduce internal fragmentation, it helps to reduce the storage for your data structures as much as possible.

Maximizing throughput requires making sure your allocator finds a suitable block quickly. This involves constructing more elaborate data structures than is found in the provided code.

Here's a strategy we suggest you follow in meeting the checkpoint and final version requirements:

- *Checkpoint.* The provided code already meets the required utilization performance, but it has very low throughput. You can achieve the required throughput by converting to an explicit-list allocator. You will want to experiment with allocation policies (e.g., first-fit, next-fit), as well as where on the free list you place newly freed blocks.
- *Final Version.* With the throughput achieved by the checkpoint version, you must greatly increase the utilization and slightly improve the throughput. You must reduce both external and internal fragmentation. Reducing external fragmentation requires achieving something closer to best-fit allocation, e.g., by using segregated lists. Reducing internal fragmentation requires reducing data structure overhead. There are multiple ways to do this, each with its own challenges. Possible approaches and their associated challenges include:
  - Eliminate footers in allocated blocks. But, you still need to be able to implement coalescing. See the discussion about this optimization on page 852 of the textbook.
  - Decrease the minimum block size. But, you must then manage free blocks that are too small to hold the pointers for a doubly linked free list.
  - Reduce headers below 8 bytes. But, you must support all possible block sizes.

  - Set up special regions of memory for small, fixed-size blocks. But, you will need to manage these and be able to free a block when given only the starting address of its payload.

Some advice on how to implement and debug your packages will be covered in the lectures and recitations.

Good luck!