CS2106: Operating Systems
Lab 4 – Building your own `malloc()`

Important:
- The deadline for LumiNUS submission: **April 16th, 2359**
- The total weightage: **10%**
  - Exercise 1: 1% **[Lab Demo Exercise]**
  - Exercise 2: 1%
  - Exercise 3:
    - Task 1: 1% **[Lab Demo Exercise]**
    - Task 2: 2%
    - Task 3: 2%
    - Task 4: 1%
  - Exercise 4: 1%
  - Exercise 5: 1%

Reminder:
- Ensure your code works properly on the SoC Compute Cluster Nodes.
- **Ensure your submission follow the specified format.**

Section 0. Basic Information

Please refer to the "Introduction to OS Labs" document if you have forgotten how to unpack / setup the lab exercises.

You only need to do this if you have just formed a team for lab 04. If you have already registered a team for previous labs, there is **no need** to re-register. Please use the google form to register your team:

https://forms.gle/2XGm3zFYpXdFTgpc6

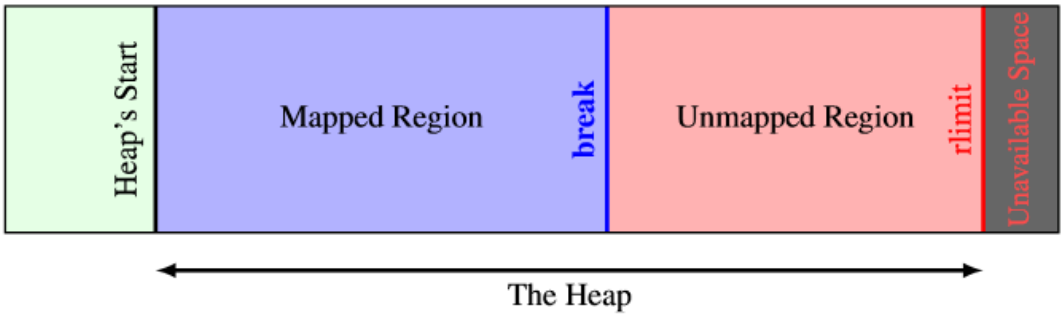
Select your name and your buddy's name from the dropdown list to form a team.

Section 1. Overview

1.1 Heap region Basics

Heap memory region is used for dynamically allocated data. In C, library calls like **`malloc()`**, **`free()`**, **`realloc()`**, etc., manipulate the heap region. Behind the scene, the heap region can be managed using **contiguous memory allocation scheme** as discussed in lecture 7. For this lab, we are going to implement our very own **`malloc()`** and **`free()`** functions. For "mysterious reasons" (see *Historical Lesson* at the end ☺), we are going to call our own version ***`mymalloc()`*** and ***`myfree()`*** respectively. To manage the complexity of this lab, substantial amount of helper / debug code has already been written for you. Your tasks are to understand, expand and improve the current implementation.

On *nix systems, the heap region is defined by three important parameters:



1. **Start:** Starting address of the heap region.
2. **Break:** The boundary of currently **usable** heap region.
3. **rLimit:** The maximum boundary of heap region, i.e. **break** can only grows up to **rLimit**. Once **break** == **rLimit**, we have run out of heap memory.

In C, we can use the "set break" **`sbrk(`*`size`*`)`** system call to increase the **break** boundary by ***`size`*** bytes. The **`sbrk(0)`** is a special usage which returns the **current address of the break** boundary.

For our usage, we will acquire a contiguous piece of heap region at the program start via **`setupHeap()`** function. This piece of memory is then used to fulfil all user **`mymalloc()`** requests. We have already coded the portion using **`sbrk()`** for you in the given skeleton code.

With this piece of "heap" memory, we can manage it using any of the contiguous memory allocation schemes. Program that manages memory allocation is also known as **memory allocator**.

1.2 Code Structure Overview

In this section, we will give a quick overview of the source files and code structure for the exercises.

For each of the exercises, you will find the following files:

ex?.c	Implementation of the `mymalloc()` and `myfree()`. You may need to modify the `setupHeap()` in some exercises. [? is the exercise number.]
mmalloc.h	Key definition for the heap meta information (i.e. book keeping information). Additional definition for data structure required for the allocation scheme. A good number of printing functions are provided to ease your coding effort. Browse around before you start.
mmalloc_driver.c	An automated tester to stress-test your implementation. There is no need to modify.
mmalloc_manual.c	A manual tester to allow more targeted testing, e.g. for testing specific corner cases etc. There is no need to modify.

Most of your work will be limited to the `ex?.c` (only ex3/ex4 may requires minor modification of the `mmalloc.h`). Some of the key structure and function definitions can be found in `mmalloc.h`:

heapMetaInfo (C Structure)	Keep bookkeeping information of the "heap" as well as information needed to perform memory operations, e.g. a. Total Size (in bytes) b. Starting address (aka **base**) of the region being managed c. Additional data structure / variables depending on the allocation scheme.
partInfo (C Structure)	Information for a partition (piece of heap allocated to user `mymalloc()` requests). Depending on the allocator, the information kept can be different.

A good place to start for each exercise would be to read "`mmalloc.h`", followed by "`ex?.c`". In most cases, "`//TODO:`" flags are added to indicate code insertion / modification point.

Makefile is also included so that when you type "`make`", the following executables will be built for you:

`ex?.exe`	Auto-tester with minimal printing. This is the version we will use for grading eventually. You may want to use this as a final check when everything seems to be working.
`exD.exe`	Essentially the same auto-tester but with more debug information. Can be used to trace and locate bug.
`exM.exe`	A manual tester that uses user input for testing. Useful when you want to test specific cases (e.g. force a certain corner case to happen, etc).

All testers take in a number of command line arguments to configure the tests. **A simple usage message will be printed if you just run the tester without providing any command line argument.**

1.3 Exercise Overview

For this lab, we are going to explore the following allocators:

a. **Linear List:**
   - Allocator keep track of the partition using a singly linked list.
   - Different fitting algorithms:
     - Exercise 1: first-fit
     - Exercise 2: best-fit

b. **Buddy System:**
   - Standard buddy system (as covered in lecture and tutorial 7) in Exercise 3.
   - Modified buddy system in Exercise 4.

In addition, we will explore ***thread-safe*** allocator, where the allocator can be called safely by multiple threads in concurrent fashion in Exercise 5 (details in Section 6).

Note that exercises for the same allocator are should be completed in order (i.e., you should finish exercise 3 before attempting exercise 4), but exercises related to different allocators are independent, e.g. you can work on Linear Allocator and Buddy System in any order. Exercise 5 is independent from any other exercises (though understanding of exercise 1 can help). Although there are many exercises, the amount of coding is quite minimal (with the possible exception in exercise 3 ☺). Most exercises just requires to read and understand (lots of) code with minor addition / modification.

Section 4. Linear List Allocator [Ex1 and Ex2]

4.1 First-Fit Linear List Allocator with Statistics [Demo Exercise]

For this exercise, the **entire allocator is already coded,** i.e. you have a working linear list allocator that handles `mymalloc`() and `myfree`() using **best-fit algorithm.** Note that this basic allocator **does not perform merging of adjacent holes during `myfree()`**.

Your tasks is to understand the allocator and calculate the following statistics for the `printHeapStatistic`() function:

- Number of occupied partition
- Total size of occupied partition in bytes.
- Number of free partition (aka hole).
- Total size of holes in bytes.

Don't panic ☺. You will spend more time to understand the given code rather than coding yourself. (hint: you need only about 10 lines of code for this exercise).

Sample Output
Command: `./ex1M.exe 1024 5 < TC/manual_1024_5.in`
Configuration: Heap Size 1024 | Active Partitions 5

```
<....... Other debug info not shown ......>
<....... At the bottom of the print out...>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x564d6bbab000   //Your address may differ
Partition list:
[+    0 |    40 bytes | 1]
[+   40 |    80 bytes | 1]
[+  120 |   120 bytes | 0]
[+  240 |   160 bytes | 1]
[+  400 |   624 bytes | 0]

Heap Usage Statistics:
======================
Total Space: 1024 bytes
Total Occupied Partitions: 3
        Total Occupied Size: 280 bytes
Total Number of Holes: 2
        Total Hole Size: 744 bytes
```

You can see that the total occupied size and total hole should add up to the total size of the heap (**1024** bytes in this test).

We have included a number of additional sample output for you in the "TC/" (test case) subfolder. In that folder, you will find:

manual_***X***_***Y***.in manual_***X***_***Y***.out	***X*** and ***Y*** are configuration parameters (heap size and number of active allocation respectively). You should execute command: `./ex1M.exe X Y < TC/manual_X_Y.in > my.out` Compare `TC/manual_X_Y.out` with `my.out`. There may be differences in the addresses (Linux randomize heap starting point ☺), but the statistics at the bottom should match exactly.
auto_***X***_***Y***_***Z***.out	***X***, ***Y*** and ***Z*** are configuration parameters (random seed, heap size and number of active allocation respectively). Since the autotester randomly generate the memory requests, there is no additional input command (i.e. no ".in" files) You should execute command as: `./ex1.exe X Y Z > my.out` Compare `auto_X_Y_Z.out` with `my.out`. There may be differences in the "Start Address" output (Linux randomize heap starting point ☺), but everything else should match exactly. This is the most succinct test output and should be used when you are quite sure about your solution and just want to stress test.
debug_***X***_***Y***_***Z***.out	Same stress test as "`auto_X_Y_Z.out`" except the output is generated via **`ex1D.exe`** which produces (lots!) of debug traces. Only use this when your stress test goes wrong and you want to find out why. You should execute command: `./ex1D.exe X Y Z > my.out`

Note that we have provided additional test cases for exercise 2 to 4 in the respective "TC/" subfolder in similar fashion.

4.2 Exercise 2 – Best-fit Algorithm

The given **`mymalloc()`** from exercise 1 implements the **first-fit** algorithm, i.e. we allocate the first large enough free partition for user allocation requests. Change the **`mymalloc()`** function in **`ex2.c`** to use **best-fit** algorithm instead.

Best-fit algorithm **minimizes the left-over after allocation**, i.e. allocate the smallest free partition that is large enough for the allocation request. **If there are multiple free partitions of the same size, chose the one with smallest starting address.**

You are not allowed to modify the existing functions and declarations, but can implement helper function(s) if needed. **Remember to copy over your completed `printHeapStatistic`()** from exercise 1 so that the final statistic is reported correctly.

Sample Output
Command: `./ex2M.exe 1024 10 < TC/manual_1024_10.in`
Configuration: Heap Size 1024 | Active Partitions 10

```
<....... Other debug info not shown ......>
<....... Near the bottom of the print out...>

<... The partition list before the last mymalloc()...>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x5599f3717000
Partition list:
[+    0 |    40 bytes | 1]
[+   40 |    80 bytes | 0] //This is a hole of 80 bytes
[+  120 |   120 bytes | 1]
[+  240 |   160 bytes | 0] //This is a hole of 160 bytes
[+  400 |   200 bytes | 1]
[+  600 |   240 bytes | 0] //This is a hole of 240 bytes
[+  840 |   184 bytes | 0] //This is a hole of 184 bytes

<....... Other debug info not shown ......>
<... The final printout at the bottom...>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x5599f3717000
Partition list:
[+    0 |    40 bytes | 1]
[+   40 |    80 bytes | 0]
[+  120 |   120 bytes | 1]
[+  240 |    88 bytes | 1]       // myalloc(88) choose this hole
[+  328 |    72 bytes | 0]
```

```
[+  400 |   200 bytes | 1]
[+  600 |   240 bytes | 0]
[+  840 |   184 bytes | 0]

Heap Usage Statistics:
======================
Total Space: 1024 bytes
Total Occupied Partitions: 4
        Total Occupied Size: 448 bytes
Total Number of Holes: 4
        Total Hole Size: 576 bytes
```

There are a number of other test cases in "TC/" subfolder which you can utilize in the same way.

Section 5. Buddy System Allocator [Ex3 and Ex4]

Please refer to the lecture set #7 and tutorial 7 for the buddy system description and algorithm if you are not very familiar about the Buddy System.

5.1 Exercise 3 - Standard Buddy System

In this exercise, you have 3 sub-tasks. You are recommended to complete the tasks in the specified order to simplify the coding process.

Task 1 – Setting up the book keeping information. [Demo "Task"]

You will work mainly in the `setupHeap()` function for this task. We need to setup the array of partition lists used by the buddy system among other configurations. Following the discussion in lecture and tutorial, we will simply call this array `A[]`.

Given the initial memory size (**can be assumed to be a power of 2**), do the following:

- Figure out the number of levels for `A[]`.
- Dynamically allocate the array `A[]`.
- Initialize each level `A[lvl]` to be an empty partition list.
- Finally, place a single partition at the appropriate level to represent the initial free partition.

If the setup is performed correctly, you should see output similar to the following:

Sample Output
Command: `./ex3.exe 1234 256 0` //"heap" size is 256 bytes
Configuration: Random Seed 1234 | Heap Size 256 | nRequest = 0 (ends immediately)

```
Heap Meta Info:
===============
Total Size = 256 bytes
Start Address = 0x55ea1bf82000
A[8]: [+    0]          //one free partition of 256 bytes
A[7]:
A[6]:
A[5]:
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:
Heap Usage Statistics: //You can ignore the stats for the demo task
======================
<...... Not Checked for this Task......>
```

Sample Output
Command: `./ex3.exe 1234 4096 0`
Configuration: Random Seed 1234 | Heap Size 4096 | nRequest = 0

```
Heap Meta Info:
===============
Total Size = 4096 bytes
Start Address = 0x56132db6b000
A[12]: [+    0] //one free partition of 4096 bytes
A[11]:
A[10]:
A[9]:
A[8]:
A[7]:
A[6]:
A[5]:
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:
Heap Usage Statistics:
======================
<..... Not Checked for this Task ......>
```

As it is **critical** to get this setup correctly, we have designated ONLY this task as the second demonstration tasks during your lab session, i.e. you only need to show that this setup is working properly without the rest of the allocation / free / stat code.

Note that the printing functions are provided, i.e. you need about ~10 lines of code in the `setupHeap`() function only.

Task 2. Allocation and Splitting

For this task, you need to handle memory allocation requests, i.e., to implement `mymalloc()` for the buddy allocator. [See lecture slide for the algorithm]

Additional note on tie-breaker: If there are multiple suitable free partitions, you should choose the one with the **smallest address**.

Suggestions to reduce the coding complexity:

- Consider implementing a helper function "`removePartitionAtLevel(…)`" which look for a free partition at a specified level, removes the partition from the list and returns to caller. An empty function is provided for you in `ex3.c`. [Note: You can choose NOT to implement this helper function.]
- Handle only allocation with NO splitting first.
- Don't be afraid of using a little recursion ☺.
- Use the manual tester "`ex3M.exe`" to test slowly and check all scenarios before you stress test using the auto tester.

Sample Output
Command: `./ex3M.exe 1024 10 < TC/manual_1024_10.in`

```
<..... we start with one free partition of 1024 bytes ...>
Heap Meta Info
===============
Total Size = 1024 bytes
Start Address = 0x55b1beeec000
A[10]: [+    0]
A[9]:
A[8]:
A[7]:
A[6]:
A[5]:
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:

<....... after mymalloc(4) .......>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x55b1beeec000
A[10]:
```

```
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]: [+   16]
A[3]: [+    8]
A[2]: [+    4]        //Multiple splits happens from A[10] to A[2]
A[1]:
A[0]:

<....... after mymalloc(16) .......>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x55b1beeec000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]:          //free partition at size 16 allocated
A[3]: [+    8]
A[2]: [+    4]
A[1]:
A[0]:

<....... after mymalloc(4) .......>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x55b1beeec000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]:
A[3]: [+    8]
A[2]:            //free partition at size 4 allocated
A[1]:
A[0]:

<....... after mymalloc(4) .......>
Heap Meta Info:
===============
```

```
Total Size = 1024 bytes
Start Address = 0x55b1beeec000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]:
A[3]:
A[2]: [+   12]   //split from A[3..2], one partition at size 4 allocated
A[1]:
A[0]:
```

Task 3. Deallocation and Merging

You can now proceed to finish off the buddy system allocator by handling **`myfree()`**. For simplicity, we includes the **memory size** in addition to the partition address as the parameters for **`myfree()`**. Note that the memory size passed in is the same as the original request size (i.e. the value user passes in for the earlier **`mymalloc`**()), not the partition size you give to the user. (e.g. if user do **`ptr`** = **`mymalloc`**(**`240`**), they will perform a corresponding **`myfree`**(**`addr`**, **`240`**) later even though you gave them a partition of size 256).

The suggested approach is similar to Task 2:

- Consider implementing a helper function "`addPartitionAtLevel`(…)" which add a free partition at a specified level into the `A[]` array. An empty function is provided for you in **`ex3.c`**. [Note: Similarly, this helper function is just a suggestion that you can ignore.]
- Handle only deallocation with NO merging first. [i.e. Just add the free partition to the right level.]
- Don't be afraid of using a little recursion ☺.
- Use the manual tester "`ex3M.exe`" to test slowly and check all scenarios before you stress test using the auto tester.

Additional note on ordering of partitions:
If there are multiple free partitions at a level, you should maintain them in **ascending order w.r.t. starting offset.**

Sample Output
Command: `./ex3M.exe 1024 15 < TC/manual_1024_15.in`
This is essentially a continuation of the sample test from task 2.

```
<....... after mymalloc(4) .......>
<....... after mymalloc(16) .......>
<....... after mymalloc(4) .......>
<....... after mymalloc(4) .......>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x55b1beeec000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]:
A[3]:
A[2]: [+   12]
A[1]:
A[0]:
<....... after myfree( +0, 4 ) .......>
< Note: we show the address offset instead of absolute
address for readability.>

********* Menu *********
Allocate= 1 <nInt>  | Deallocate = 2        |
Print Heap Meta = 3 | Print Heap Stat = 4 |
Print Entire Heap = 5 | Exit = 0
Your Choice =>
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x556704b82000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]:
A[3]:
A[2]: [+    0] [+   12] //Free partition at [+0] size 4 added, no buddy
A[1]:
A[0]:
<....... after myfree( +4, 4 ) .......>
```

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x556704b82000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]:
A[3]: [+    0]      //2. Merged buddy blocks from A[2]
A[2]: [+   12]      //1. Free partition at [+4] size 4 added, merge with [+0]
A[1]:
A[0]:
<..... after myfree( +16, 16 ) ....->

Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x556704b82000
A[10]:
A[9]: [+  512]
A[8]: [+  256]
A[7]: [+  128]
A[6]: [+   64]
A[5]: [+   32]
A[4]: [+   16]  //Free partition at [+16] size 16 added, no buddy
A[3]: [+    0]
A[2]: [+   12]
A[1]:
A[0]:
<..... after myfree( +8, 4 ) ....->

Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x556704b82000
A[10]: [+    0]  //At the end. We are back with one free partition of 1024
A[9]: //…. Same merging happens from A[5 to A10]
A[8]: //…. Same merging happens from A[5 to A10]
A[7]: //…. Same merging happens from A[5 to A10]
A[6]: //…. Same merging happens from A[5 to A10]
A[5]: //…. Same merging happens from A[5 to A10]
A[4]: //3. Free partition at [+0] size 16 added, merge with [+16]
```

```
A[3]: //2. Free partition at [+12] size 8 added, merge with [+0]
A[2]: //1. Free partition at [+8] size 4 added, merge with [+12]
A[1]:
A[0]:
```

Task 4. Statistics!

Congratulation of reaching this task! The major pain (task 2 and 3) are now over. You just need to compute a few statistics for `printHeapStatistic`().

Sample Output
Command: `./ex3M.exe 1024 5 < TC/manual_1024_5.in`

```
<..after mymalloc(12) >
   => partition given is size 16
   => internal frag = 4 bytes (wasted!)
<..after mymalloc(40)=> internal frag = 24 bytes >
<..after mymalloc(320)=> internal frag = 192 bytes >
<..after mymalloc(160)=> internal frag = 96 bytes >

Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x5627c8677000
A[10]:
A[9]:
A[8]:
A[7]: [+  128]
A[6]:
A[5]: [+   32]
A[4]: [+   16]
A[3]:
A[2]:
A[1]:
A[0]:

Heap Usage Statistics:  //Task 4
======================
Total Space: 1024 bytes
Total Free Partitions: 3 //See meta info above for the 3 partitions
Total Free Size: 176 bytes  //16 + 32 + 128
Total Internal Fragmentation: 316 bytes //4 + 24 + 192 + 96
```

Hint: Calculate the internal fragmentation in `mymalloc()` and `myfree()`.
Hint 2: The **Free Size** can be easily tallied on the fly.

5.2 Exercise 4 - Modified Buddy System

You need the completed code from exercise 3 for this exercise. Please copy over **`ex3.c`** as **`ex4.c`** into the **`ex4/`** subfolder before starting.

For this exercise, we are going to introduce a few minor modifications to the standard setup of buddy system. The changes are quite minor (~10 lines of code each).

Change No.1: Non-Power-Of-2 Memory Size

The **s**tandard **b**uddy **s**ystem (**SBS**) has a pretty restricting setup, the piece of "heap" memory we manage must have power-of-2 size. Imagine if you have 1200 bytes of memory, the SBS can only manages at most 1024 bytes, wasting 176 bytes! This is a form of external fragmentation for **SBS**.

To handle non-power-of-2 memory size is surprisingly simple. We simple chop the initial piece of memory into **multiple power-of-2 free partitions.**

For example, 1200 bytes can be handled as "1024 + 128 + 32 + 16", i.e. instead of starting with just one piece of free memory of 1024 at address 0, we have 4 pieces:

- Size = 1024, Starts at 0.
- Size = 128, Starts at 1024.
- Size = 32, Starts at 1152.
- Size = 16, Starts at 1184.

Once the initial partitions are added, **there is NO change to `mymalloc()` and `myfree()`**! [Take a moment to convince yourself why this is the case!].

Using this approach, we can now manage memory region of "any size"!

Your task is to **update `setupHeap()`** such that we can handle non-power-of-2 memory size. You should **add partitions in descending order w.r.t. partition size** (e.g. in the example above, 1024 partition should be added first). Of course, **you should always add the largest possible partitions** (e.g. instead of adding two size 256 partitions, you should add **one** size 512 partition).

Sample Output
Command: `./ex4.exe 1234 1200 1 1024 0`
RanSeed 1234 | HeapSize 1200 | minPartSize 1 | maxPartSize 1024 | nReq 0

```
Heap Meta Info:
===============
Total Size = 1200 bytes
Start Address = 0x55850fc89000
```

```
A[10]: [+    0]
A[9]:
A[8]:
A[7]: [+ 1024]
A[6]:
A[5]: [+ 1152]
A[4]: [+ 1184]
A[3]:
A[2]:
A[1]:
A[0]:
```

Change No.2: Smallest and largest allocatable size

The standard buddy system (SBS) allocates partition from size 1 (2^0) all the one to the largest partition as determined by the initial memory size. However, in real allocator, it is usually desirable to put a limit on:

- Smallest allocatable partition size (**SS**): Giving out a partition as small as one byte does not make sense (the book keeping overhead far outweigh the "convenience"). So, we commonly will set a lower limit on the partition size to be allocated. For example, if the SS is set to 64 bytes, **any requests < 64 bytes will get a partition of 64 bytes instead, subjecting to availability.** This increases internal fragmentation, but reduces the number of partitions (and the book keeping overhead).
- Largest alloctable partition size (**LS**): Similarly, it is desirable to keep the largest partition in checks. (e.g. to match with page size). We will similarly set an upper limit on the largest partition we can (or willing to) allocate. **Any request > LS** will be rejected (i.e. "NULL" is returned).

Modify the relevant functions (likely to be **`setupHeap`**(), **`mymalloc`**() and **`myfree`**()) to handle the lower and upper limit.

Note that change No.2 will impact **change No.1 too!** For example, if we set **LS** to be 512 bytes, and the **SS** to be 32 bytes, then an initial memory size of 1200 has to be handles as 512 + 512 (we cannot handle partition size of 1024 now) + 128 + 32 (the last 16 bytes cannot be utilized as we do not give out partition < 32 bytes).

Sample Output
Command: `./ex4.exe 1234 1200 32 512 0`
RanSeed 1234 | HeapSize 1200 | minPartSize 32 | maxPartSize 512 | nReq 0

```
Heap Meta Info:
===============
Total Size = 1184 bytes
Start Address = 0x564964f7d000
```

```
A[9]: [+    0] [+  512]
A[8]:
A[7]: [+ 1024]
A[6]:
A[5]: [+ 1152]
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:
```

You can verify your code by printing the heap meta information for change No.1 and use specific allocation requests to test out change No.2 (e.g. when request size > LS, request size < SS, etc).

Sample Output
Command: `./ex4M.exe 1200 32 512 5`
HeapSize 1200 | minPartSize 32 | maxPartSize 512 | Active Partitions 5

```
<..Initial Free Partitions >
Heap Meta Info:
===============
Total Size = 1184 bytes
Start Address = 0x565184913000
A[9]: [+    0] [+  512]
A[8]:
A[7]: [+ 1024]
A[6]:
A[5]: [+ 1152]
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:
<..after mymalloc(4) >

Heap Meta Info:
===============
Total Size = 1184 bytes
Start Address = 0x565184913000
A[9]: [+    0] [+  512]
A[8]:
A[7]: [+ 1024]
A[6]:
A[5]:   //Partition of size 32 is given (This is the smallest alloctable size)
A[4]:
A[3]:
A[2]:
```

```
A[1]:
A[0]:
<..after mymalloc(4) >
Total Size = 1184 bytes
Start Address = 0x565184913000
A[9]: [+    0] [+  512]
A[8]:
A[7]:
A[6]: [+ 1088]
A[5]: [+ 1056] //After split A[7..5], Partition of size 32 is given
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:
<..after mymalloc(520) >
Total Size = 1184 bytes
Start Address = 0x565184913000
A[9]: [+    0] [+  512] //No change as we cant allocate > 512 bytes
A[8]:
A[7]:
A[6]: [+ 1088]
A[5]: [+ 1056]
A[4]:
A[3]:
A[2]:
A[1]:
A[0]:
```

Section 6. Thread-Safe Memory Allocator [Exercise 5]

So, what can be worse than coding memory allocator? Making the allocator ***thread-safe***, of course! "Thread-Safe" functions / data structures can be used in multi-threaded programs with the guarantee of correctness (or more precisely, "free from race conditions").

Let us use the simple first-fit linear allocator (i.e., from ex1) to illustrate the issue. Use "`make`" to prepare the auto-tester. The auto-tester has been upgraded to spawn multiple tester threads where each threads will perform a number of mymalloc() / myfree() randomly. [Note that there is no manual tester ex5M.exe for this exercise.]

If you run the auto-tester with ~5 threads, ~50 requests multiple times, you will encounter the following two types of errors:

A. Memory Corruption Error

Two or more tester threads somehow are allocated to the same partition and they overwrite each other's result and trigger this error during validation.

Sample Error
Command: `./ex5.exe 1234 4096 500 5`
Configuration: Random See 1234 | HeapSize 4096 | nRequest 500 | nThread 5

```
Tester Thread[140636620441344] with [500] requests
Tester Thread[140636612048640] with [500] requests
Tester Thread[140636603655936] with [500] requests
Tester Thread[140636595263232] with [500] requests
Tester Thread[140636586870528] with [500] requests
Memory corruption detected!
Memory region at [0x55d314a308d8] should have 61 magic values
Memory corruption detected!
<.. other warnings not shown..>
```

B. Concurrent memory operations error

Multiple parallel memory operations are detected. By "luck" (i.e. non-destructive interleaving), the end result seems ok but there is definite chance for the interleaving to cause issues (i.e. (A)) on some other runs.

You will see "`** FAILED: Memory operation is NOT thread-safe`" at the end of the execution.

Your task is very simple, protect the memory operation `mymalloc`() and `myfree`() to eliminate (A) and (B) errors from happening.

Notes:

- Use the **simplest approach.**
- For simplicity, we will **not reward concurrency**, i.e. we care about correctness only and do not grade on concurrency allowance.
- However, if you are able to consider concurrency, we will award 2 ***impress points*** to the top 5 solutions that provide the highest concurrency. Submit a **separate source code `ex5_impress.c`** if you want to give this challenge a try.
- Use only **general semaphores**.
- Do not modify / move the additional semaphore codes we inserted into `mymalloc`() and `myfree`(). Those are for detecting type (B) errors. **[Violating this rule can invalidate your attempt!]**

Just like all other race condition related exercise, there is no standard test case we can give. A correct solution should always produce: `"** Memory operation is thread-safe"` at the end of execution.

Section 7. Pondering and Historical Lesson

This lab should shed lights on some "weird" behaviours you may have encountered before with the standard **`malloc()`**, **`free()`**. For example:

1. Why data in recently freed memory space seems intact (for a while)?

2. Why exceeding the range of allocated memory space **sometimes** does not cause segmentation fault?

3. Why dynamically allocated memory space contains "random" garbage values?

You should be able to give a pretty good guess / answer to the above questions after this lab.

Now, one last surprise. Once you have the complete code, you can try to link any code that uses dynamic memory with the **`ex?.c`**, e.g. linked list code (e.g. from lab 1) would be a good showcase. The linked list code should work perfectly with your very own **`mymalloc()`** and **`myfree()`**, fun eh? ☺

[Note: you need to remove the "**`<stdlib.h>`**" library, as it provides the standard malloc/free and can conflict with your own implementations.]

[Note2: Of course, you need to rename all **`malloc()`** to **`mymalloc()`** and **`free()`** to **`myfree()`** in the user code to utilize your own version.]

[Note 3: For buddy system allocator, you need to keep the partition size allocated to user request internally, so that the **`free`**() can take in only a memory address.]

Historical Lesson

It seems intuitive that we should directly implement **`malloc`**() and **`free`**() instead of our own "**`myXYZ`**()" version. That would be much more realistic and potentially useful (i.e. you can have your own allocator instead of the default!).

Well, we tried that (~4 years ago) and concluded that's a (somewhat hilarious) disaster. For a start, do you know that **`printf`**() actually uses **`malloc`**() internally (to allocate the output string before writing on screen)? So, if you use **`printf`**() to debug your **`malloc`**() implementation……. ☺

Additionally, the allocator itself usually needs dynamic memory allocation (e.g. to build a linked list etc). So, it is definitely not a good idea trying to fix the vehicle while it is moving at high speed!

Section 8. Submission

Zip the following folders and files as `A0123456X.zip` (**use your student id with "A" prefix and the alphabet suffix**). Remember to modify the archive name and the content accordingly as needed.

a. **ex2/**
   a. **ex2.c**
b. **ex3/**
   a. **ex3.c**
   b. **mmalloc.h** (submit regardless of whether you change this file)
c. **ex4/**
   a. **ex4.c**
   b. **mmalloc.h** (submit regardless of whether you change this file)
d. **ex5/**
   a. **ex5.c**
   b. **ex5_impress.c** (Optional, only for impress points)

Do **not** add additional folder structure during zipping, e.g., do not place the above in a "**lab4/**" folder etc.

3.1 Do a quick self-check

We have provided a self-checking shell script to help validating your zip archive. The script checks the following for this lab:

a. The name or the archive you provide matches the naming convention mentioned above.
b. Your zip file can be unarchived, and the folder structure follows the structure described above.
c. All files for each exercise with the required names are present.
d. Each exercise can be compiled.

Once you have produce the zip file, you will be able to check it by performing the following steps at the **L4/** folder:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh A0123456X.zip (replace with your zip file name)
```

The **check_zip** script will print out the status of each of the checks. Successfully passing checks enable the lab TA to focus on grading your assignment instead of performing lots of clerical tasks (checking filename, folder structure, etc). Please note that **points might be deducted if you fail these checks.**

Expected Successful Output
Checking zip file.... Unzipping file: A0123456X.zip Transferring necessary skeleton files ex2: Success

```
ex3: Success
ex4: Success
ex5: Success
```

3.2 Upload to LumiNUS before deadline!

After verifying your zip archive, please upload the zip file to the "Student Submission→Lab 4" file folder on LumiNUS. Please note that **late penalty is quite steep,** so submit early to avoid last minute issues.

Also, remember that if you are in a team, **only one member need to submit into LumiNUS.** The archive name should be in the format `A0123456X.zip`, where `A0123456X` stands for the student number of *any of the team members*; we will do "background processing" to match with your team members automatically. Both team members will receive the same score for the lab.

Reference:

1. "A Malloc Tutorial" by Marwan Burelle, 2009, http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf
    - Good exploration material.