CS2106 Introduction to Operating Systems
Lab 4
Contiguous Memory Allocation

1 Introduction

Imagine every *malloc()* in your programs as a tiny "help!, please give me more memory" request flying into the OS. The OS usually will answer these request without fail and returns a valid pointer to the newly assigned memory. In this lab, you'll step into the OS's shoes and build two strategies that power everything from the foundation: a **linked-list** memory allocator and the **buddy system** memory allocator. We will look at implementing our own version of malloc and free, called "mymalloc" and "myfree", so that you can explore some of the issues in creating memory allocation algorithms.

Deadline and Submission

a. You can do this lab alone or with a partner.
b. Ensure that both of you fill in your names, student IDs and group numbers in the answer book AxxxxxxY.docx.
c. Rename AxxxxxxY.docx to the student ID of the submitter before submitting.
d. The demo for this lab will be held in Week 12, the week of **3 November 2025**
e. Zip up the following files into a single ZIP file called AxxxxxxY.zip, where AxxxxxxY is your student ID:
    - Your answer book, properly renamed.
    - Your mymalloc.c and mymalloc.h in Part 2
    - Your mymalloc.c and mymalloc.h in Part 3
f. Upload your ZIP file, properly named, to Canvas by **11.59 pm, Sunday 9 November 2025.**
g. The folder has been set to close very shortly after midnight on Monday 10 Nov 2025. **Once the folder is closed no submissions will be accepted**.
h. This lab is worth 20 marks in total (8 marks for the report, 12 marks for the two demos).

2 Linked List Memory Allocation

We will now create best-fit memory allocation algorithms using linked lists. Open the linkedlist directory and you will see:

Common Files	Description
mymalloc.c, mymalloc.h	Where you will implement only the best-fit version of your memory allocation algorithm.
testmalloc.c	Test mymalloc.
harness.c	Demo file.

The ***mymalloc.h*** file also contains the MEMSIZE constant which is used to create a heap of 64KB. You may check *`setupHeapRegion()`* function to see how the memory region was setup. After calling *`setupHeapRegion()`*, you can get the base address of your heap through a static variable _heap in ***myalloc.h***. ***mymalloc.c*** and ***mymalloc.h*** consist of *`get_index`*, *`print_memlist`*, *`mymalloc`*, and *`myfree`*, of which you would need to implement *`print_memlist`*, *`mymalloc`* and *`myfree`*.

Implement the best-fit allocation algorithm in ***mymalloc.c***, and the corresponding free in *`myfree()`*. You are **not allowed** to allocate extra memory in this exercise and may only use the _heap variable as well as the heap region allocated for you to maintain the heap information.

Question 1.1 (1 mark)

Explain how setupHeapRegion() works? What does sbrk() do?

Question 1.2 (1 mark)

What content should a linked list node store? What is the overhead?

Question 1.3 (2 mark)

Explain how you would maintain the linked list meta data in the same memory region as your data. What are the pros and cons of your scheme?

Hint: Linked list nodes are usually created with malloc(), which dynamically allocates new memory. However, in this case all the memory allocated is at your disposal. How you would you split the given memory to accommodate for your meta data.

You can verify your implementation by doing:

```
gcc mymalloc.c testmalloc.c -o testmalloc
./testmalloc
```

If all goes well you will see an output like this:

```
Allocating 2048 bytes to ptr1
Total Size = 65536 bytes
Start Address = 0x100430000
Partition Meta Size = 16 bytes
Status: ALLOCATED Start index: + 0 Length: 2048
Status: FREE Start index: + 2064 Length: 63456

Allocating 6144 bytes to ptr2
Total Size = 65536 bytes
Start Address = 0x100430000
Partition Meta Size = 16 bytes
Status: ALLOCATED Start index: + 0 Length: 2048
Status: ALLOCATED Start index: + 2064 Length: 6144
Status: FREE Start index: + 8224 Length: 57296
```

You can see the full output that you should get in ll.out provided. If your implementation is correct you will get an identical output.

DEMO 2: (6 marks)

Compile your memory manager with harness.c using:

```
gcc testmalloc.c mymalloc.c -o testmalloc
```

Run your harness program for the TA:

```
./testmalloc
```

3 Buddy System Memory Allocation

In this exercise, to facilitate easier implementation, you are allowed to use *malloc()* to create your linked list. Please see the details below.

a. The Linked List Library

The linked list library is available in llist.c and llist.h. All routines have been implemented for you. The basic linked list structure TNode is defined as:

```
typedef struct tn {
    unsigned int key;
    TData *pdata; // Pointer to the data you want to store

    struct tn *trav; // Only used in the root for traversal
    struct tn *tail; // Only used in the root for finding the end of the list
    struct tn *prev;
    struct tn *next;
} TNode;
```

It consists of a key that is used sort the nodes into ascending or descending order, a pointer of type TData (see below) to point to a data node, a prev and next pointer to point to the previous and next nodes, and two pointers trav and tail that are used only by the "succ" and "pred" iterator functions, and to allow reverse traversal of the list.

There is a TData structure that you can use to define the type of data you want to put into the node. It is currently defined as:

```
typedef struct td {
    int val;
} TData;
```

You should modify TData to hold the data that you need to manage your memory. Note that you **CAN** choose to modify TNode directly to put in the data you want to store in the node, instead of using TData.

The following library calls are available in llist.c. Note again that ALL of these have already been implemented for you. See testlist.c for how to use each function.

Function Name	Parameters	Description
dbprintf	Same parameters as printf	A debug version of printf that prints to the screen only if the DEBUG macro in llist.h is defined.
make_node	key: The key value for sorting the list. data: Pointer to the data to add to the node. NULL if you are not using this.	Creates a new linked list node.

insert_node	llist: Pointer to the linked list node: The node to be inserted created using make_node. dir: Sort direction. ASCENDING or DESCENDING	Inserts a new node created by make_node into the linked list in the specified sort order.
delete_node	llist: Pointer to the linked list node: The node to delete	Deletes node from the linked list.
find_node	llist: The linked list key: Value to search for	Searches the linked list for key and returns the node holding key. Returns NULL if key is not found.
merge_node	llist: The linked list node: The node to merge dir: PRECEDING or SUCCEEDING (previous or next)	Between the provided node and the PRECEDING or SUCCEEDING node, the node with the larger key is deleted.
purge_list	llist: Pointer to the linked list.	Purges the linked list and sets it to NULL.
process_list	llist: Linked list func: Function to call for each node of the linked list.	Traverse the linked list and call func for each node.
reset_traverser	llist: The linked list where: FRONT or REAR	Resets the traverser to the front or rear of the linked list.
succ	llist: The linked list	Returns the current node and advances the traverser to the next node.
pred	llist: The linked list	Returns the current node and moves the traverser to the previous node.

You can test the linked list library compiling and running testlist:

```
gcc llist.c testlist.c -o testlist
./testlist
```

Hit return to see the numbers inserted in ascending order, do a series of deletes, and purge the list. Hit return again to repeat with the numbers in descending order.

Note: It may seem a little strange that we are using a library that uses malloc to implement our own malloc, but Operating Systems would have routines to manage their own private memory where they create and use data structures to manage various services. Rather than try to implement our own memory management just for the linked list, we will simply use malloc as a proxy for internal routines.

b. The Buddy System Memory Allocator

We will now create buddy system memory allocation algorithms. Open the buddy directory and you will see:

Common Files	Description
llist.c, llist.h	Linked list library
testlist.c	Example of how to use llist.c.
mymalloc.c, mymalloc.h	Where you will implement buddy-system version of your memory allocation algorithm. Within the free list of the chosen block size, pick the first free block.
testmalloc.c	Test mymalloc.
harness.c	Demo file.

Like the previous sections, mymalloc.c and mymalloc.h consist of get_index, print_memlist, mymalloc, and myfree. Additionally, get_size is added to be used in harness.c.

Function Name	Parameters	Description
get_index	ptr: Pointer to a memory region returned by mymalloc.	Implemented for you. Returns an index corresponding to the memory region created by mymalloc. Used by the test harness but you can also use it in your code if you find it useful.
get_size	ptr: Pointer to a memory region returned by mymalloc.	Returns the size of the corresponding to the memory region created by mymalloc. Used by the test harness but you can also use it in your code if you find it useful.
print_memlist	None	Print out the memory layout for each block size like the example below. (The format is not strict. As long you can show how each block is laid out, it is fine.)
mymalloc	size: Number of bytes to allocate	Returns a pointer to the allocated memory, or NULL if no suitable memory is found.
myfree	ptr: Pointer to block of memory to free	Frees memory pointed to by ptr. Fails silently if ptr is NULL or does not point to a memory region created by mymalloc.

		(Note: This is different from free which crashes under such circumstances)

Memory layout example:

```
Block size 1024 KB: ALLOCATED, 0, 1024 ->
Block size 512 KB: ALLOCATED, 0, 512 -> FREE, 512, 512 ->
Block size 256 KB:
Block size 128 KB:
Block size 64 KB:
Block size 32 KB:
Block size 16 KB:
Block size 8 KB:
Block size 4 KB:
Block size 2 KB:
Block size 1 KB:
```

The mymalloc.h file also contains the MEMSIZE constant which is set to create a heap of **1024 KB**. For the sake of simplicity, the minimum block size that you can allocates is **1 KB**. Just as with the bitmap and linked list implementation, mymalloc.c contains a character array called _heap. You will allocate your memory from this array.

You can verify your implementation by doing:

```
gcc mymalloc.c llist.c testmalloc.c -o testmalloc
./testmalloc
```

If all goes well you will see an output like this:

```
Allocating 512 KBytes to ptr1
Allocated 512 KB successfully to ptr1
Block size 1024 KB: ALLOCATED, 0, 1024 ->
Block size 512 KB: ALLOCATED, 0, 512 -> FREE, 512, 512 ->
Block size 256 KB:
Block size 128 KB:
Block size 64 KB:
Block size 32 KB:
Block size 16 KB:
Block size 8 KB:
Block size 4 KB:
Block size 2 KB:
Block size 1 KB:

Allocating 120 KBytes to ptr2
Allocated 120 KB successfully to ptr2
Block size 1024 KB: ALLOCATED, 0, 1024 ->
Block size 512 KB: ALLOCATED, 0, 512 -> ALLOCATED, 512, 512 ->
Block size 256 KB: ALLOCATED, 512, 256 -> FREE, 768, 256 ->
Block size 128 KB: ALLOCATED, 512, 128 -> FREE, 640, 128 ->
Block size 64 KB:
Block size 32 KB:
Block size 16 KB:
Block size 8 KB:
Block size 4 KB:
Block size 2 KB:
Block size 1 KB:
```

You can see the full output that you should get in buddy.out in the buddy directory. If your implementation is correct you will get an identical output. (Note that printing out the memory layout is not required to be exactly the same).

There is a test harness program called harness.c. Compile the harness using:

```
gcc harness.c mymalloc.c llist.c -o harness
```

DEMO 2. (6 marks)

Your TA will ask you to run one of the test harnesses to show that your buddy system algorithm works.

Question 2.1a (1 mark)

Given only the allocated pointer to get_size(), how do your function find out how large your memory block is? Copy and paste your code here and explain it.

Question 2.1b (1 mark)

Given only the pointer to the myfree(), how do your function find out which memory block in the buddy system was allocated to free it? Copy and paste your code here and explain it.

Question 2.2 (1 mark)

Given a starting address of a block, how do you find its buddy address?

Question 2.3 (1 mark)

How do you detect if there is a free buddy to perform a merge? How do you merge it? Copy and paste your code here and explain it.

4 Conclusion

In this lab we've explored the practical aspects of implementing a memory manager, like one that we would find in an operating system. We've looked at how to do this using both linked lists and buddy system.

~ END OF FILE~