CS2106 Introduction to Operating Systems
Lab 4
Contiguous Memory Allocation

1. Introduction

In this lab 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.

There are three parts to this lab:

(1) Bitmap Allocation

In the first part you will implement a bitmap-based memory first-fit allocation algorithm. You will need to implement your own algorithms to scan a bitmap to allocate a suitable stretch of memory.

(2) Linked List Allocation

In the second part you will implement linked-list based first-fit memory allocation algorithms. A set of linked-list routines have been provided to you, and you need to implement the allocation algorithms themselves.

(3) Buddy System Allocation

In the third part you will implement buddy system memory allocation algorithms. A set of linked-list routines have been provided to you, and you need to implement the allocation algorithms themselves.

The demo for this lab will be held in Week 13, the week of **14 April 2025**, and the submission deadline will be **23:59**, **Sunday 20 April 2025**. **This lab is worth 20 marks**.

2. Submissions

You may work on this lab individually or with a partner from any lab group. Fill in your names, student IDs and lab group numbers in the answer book AxxxxxxY.docx, renaming it to the student ID of the student submitting. Only one copy needs to be submitted.

Please submit to Canvas by 2359 hours on **Sunday 20 April 2025**.

3. Bitmap Memory Allocation

In this first section you will be allocating a first-fit memory allocation algorithm using bitmaps. Your bitmap will be stored in an array of unsigned char. Every bit in the bitmap represents one byte of memory to be allocated/freed.

Switch to the "bitmap" directory. You will see the following files:

Filename	Purpose
bitmap.h, bitmap.c	Header and source code files where you will implement the routines to manage the bitmap.
mymalloc.c, mymalloc.h	Header and source code files where you will implement your own versions of malloc and free, called mymalloc and myfree.
testmap.c	Test file to check that your bitmap routines work.
testmalloc.c	Test file to check that your mymalloc and myfree work.
harness.c	Demo file.

Let's now get right into what you need to do:

3.1 Implement the Bitmap Algorithms

The bitmap.c file contains the following functions. Aside from print_map, allocate_map and free_map, you need to implement the rest of the functions.

Function Name	Parameters	Description
print_map	map: The bitmap declared as an array of unsigned char. Each char is 8 bits, each bit represents 1 byte of memory to be allocated/freed. len: Length of the array in characters.	Implemented for you. Prints out the bitmap
search_map	map: The bitmap len: Length of the bitmap in characters num_zeroes: Minimum # of consecutive zeroes we need to find Returns: Index pointing to start of first series zeroes that is at least "numzeroes" long, or -1 if none found.	Searches for a stretch of 0's that is at least "num_zeroes" long. Returns the index to the start of the stretch or -1 if none are found. The first bit in the bitmap has index 0, second bit has index 1, etc.

Function Name	Parameters	Description
set_map	map: The bitmap start: Starting index of first bit to set or clear. length: # of bits to set or clear. value: Non-zero value will set the stretch of bits to 1, and a 0 value will clear the stretch of bits to 0.	Sets or clears a stretch of bits starting from index to index + length – 1.
allocate_map	map: The bitmap start: The index of the first bit to set. length: The number of bits to set.	Implemented for you. Bits index to index + length – 1 will be set to 1.
free_map	map: The bitmap start: The index of the first bit to clear. length: The number of bits to clear.	Implemented for you. Bits index to index + length – 1 will be set to 0.

Implement search_map and set_map as specified in the descriptions column in the bitmap.c file.

3.2 Testing your Bitmap Routines

To test whether your bitmap routines are written correctly, compile and run using:

```
gcc testmap.c bitmap.c -o testmap
./testmap
```

If it is implemented correctly, you will see:

```
Length: 2, Expected: 2, Actual: 2
Length: 4, Expected: 5, Actual: 5
Length: 6, Expected: 5, Actual: 5
Length: 12, Expected: 26, Actual: 26
Length: 128, Expected: -1, Actual: -1

Allocating 2 bytes
BEFORE: 11001000 00011111 00001100 11000000 00000001
AFTER:  11111000 00011111 00001100 11000000 00000001

Allocating 12 bytes
BEFORE: 11111000 00011111 00001100 11000000 00000001
AFTER:  11111000 00011111 00001100 11111111 11111101

Length: 12, Expected: -1, Actual: -1
Freeing 12 bytes
BEFORE: 11111000 00011111 00001100 11111111 11111101
AFTER:  11111000 00011111 00001100 11000000 00000001

Length: 12, Expected: 26, Actual: 26
```

If your implementation is incorrect, testmap will crash:

```
Length: 2, Expected: 2, Actual: 2
Length: 4, Expected: 5, Actual: 5
Length: 6, Expected: 5, Actual: 5
Length: 12, Expected: 26, Actual: 26
Length: 128, Expected: -1, Actual: 1
testmap: testmap.c:14: print_result: Assertion `ndx == expected' failed.
Aborted (core dumped)
```

3.3 Implementing your Memory Manager

If your bitmap routines from 3.2 are working correctly, you can now implement your memory manager!

Question 3.1a (1 mark)

Given a memory size of 64 bytes, how large would your bitmap be in bytes if we allocate in units of 1 byte?

Question 3.1b (1 marks)

Assume that we use a linked list structure instead, with each node consisting of a 4-byte forward pointer, a 4-byte backward pointer, a 4 byte-starting address, a 4-byte size field, and a 1 byte status (free/not free) field. How many bytes would we need in the worst case to manage our 64 bytes of memory, if our smallest allocation unit is 1 byte?

Question 3.1c (1 marks)

List one advantage and one disadvantage of bitmaps.

Open the mymalloc.c file and you will see the following routines:

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.
print_memlist	None	Call print_map to print the current memory map.
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)

There are also some constants in mymalloc.h that you need to know about:

Constant	Description
MEMSIZE	Size of memory in bytes. Set at 64 bytes.

You should allocate memory from the heap, simulated in mymalloc.c using an array of char:

```
char heap[MEMSIZE] = {0};
```

Question 3.2 (1 mark)

The allocated memory is held in an array for type char. Would it make a difference if the array is of type `unsigned char` instead? Why or why not?

Implement print_memlist, mymalloc and myfree using the routines in bitmap.c. See Section 4 also on a linked list library (llist.c and llist.h in the linkedlist directory) that you *might* find useful. You can copy over these files to the bitmap directory to use them.

Question 3.3 (1 mark)

Does your myfree routine need to know how many bytes of memory need to be freed? If so, where will you get this information since you only call "myfree" with a pointer to the memory to be freed with no length information?

3.4 Testing Your Memory Allocation Routines

To test your memory allocation routines, compile and run using:

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

Note: If you use the linked list library from Section 4, compile with:

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

If your mymalloc and myfree are working properly, you should see:

```
Allocating 4 bytes to ptr1
11110000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Allocating 32 bytes to ptr2
11111111 11111111 11111111 11111111 11110000 00000000 00000000 00000000

Allocating 4 bytes to ptr3
11111111 11111111 11111111 11111111 11111111 00000000 00000000 00000000

Freeing ptr2
11110000 00000000 00000000 00000000 00001111 00000000 00000000 00000000

Allocating 24 bytes to ptr2
11111111 11111111 11111111 11110000 00001111 00000000 00000000 00000000

Allocating 6 bytes to ptr4
11111111 11111111 11111111 11111111 11001111 00000000 00000000 00000000

Freeing ptr2
11110000 00000000 00000000 00001111 11001111 00000000 00000000 00000000

Allocating 32 bytes to ptr5
Allocation failed.
11110000 00000000 00000000 00001111 11001111 00000000 00000000 00000000

Freeing ptr1
00000000 00000000 00000000 00001111 11001111 00000000 00000000 00000000

Freeing ptr2
00000000 00000000 00000000 00001111 11001111 00000000 00000000 00000000

Freeing ptr3
00000000 00000000 00000000 00001111 11000000 00000000 00000000 00000000

Freeing ptr4
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Freeing ptr5
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
~
```

You can also compare with the bitmap.out file provided in the bitmap directory.

DEMO 1: (2 marks)

Compile your memory manager with harness.c using:

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

If you are using the linked list from section 4:

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

Run your harness program for the TA:

```
./harness
```

If you've done everything correctly the harness will run without crashing.

4. Linked List Memory Allocation

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

Common Files	Description
llist.c, llist.h	Linked list library
mymalloc.c, mymalloc.h	Where you will implement only the first-fit version of your memory allocation algorithm.
testlist.c	Example of how to use llist.c.
testmalloc.c	Test mymalloc.
harness.c	Demo file.

4.1 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.

4.2 Implementing the First Fit Allocation Algorithm

As before, mymalloc.c and mymalloc.h consist of get_index, print_memlist, mymalloc, and myfree, of which you need to implement print_memlist, mymalloc and myfree. The mymalloc.h file also contains the MEMSIZE constant which is set to create a heap of 64KB. Just as with the bitmap implementation, mymalloc.c contains a character array called _heap. You will allocate your memory from this array.

Using the linked list library llist.c and llist.h, implement the first-fit allocation algorithm in mymalloc, and the corresponding free in myfree.

Question 4.1 (1 mark)

What additional data did you add to TData (or TNode) to implement your first-fit manager? List down the data you added in the form of <datatype> <fieldname>. E.g.

```
int start_addr;
char status;
…
```

Question 4.2 (1 mark)

Given a total heap size of 64KB, what is the best case and worst case storage requirement for your linked list in bytes, inclusive of all the fields in TNode and TData, if we allocate memory in units of 1 byte?

Question 4.3 (1 mark)

Why does it generally not make sense to allocate memory in units of one byte?

Question 4.4 (1 marks)

Given the size of your linked list nodes, what is a sensible minimum allocation unit size?

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 2048 bytes to ptr1
Status: ALLOCATED Start index: 0 Length: 2048
Status: FREE Start index: 2048 Length: 63488

Allocating 6144 bytes to ptr2
Status: ALLOCATED Start index: 0 Length: 2048
Status: ALLOCATED Start index: 2048 Length: 6144
Status: FREE Start index: 8192 Length: 57344
```

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

DEMO 2: (2 marks)

Compile your memory manager with harness.c using:

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

Run your harness program for the TA:

```
./harness
```

If you've done everything correctly the harness will run without crashing.

5. Buddy System Memory Allocation

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

Like the two 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)

<u>Memory layout example:</u>

```
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 3. (2 marks)

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

Question 5.1a (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 5.1b (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 5.2 (1 mark)

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

Question 5.3 (2 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.

6. 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 bitmaps and linked lists.

~ END OF FILE~