CS2106: Introduction to Operating Systems
Lab Assignment 4 – Memory Management

Important:

- **The deadline of submission through LumiNUS is Wed, 4 November, 2pm**
- The total weightage is 7% (+1% bonus):
  - Exercise 0: 1 % **[Lab Demo Exercise]**
  - Exercise 1: 2 %
  - Exercise 2: 3 %
  - Exercise 3: 1 %
  - Exercise 4: 1 % [Bonus mark]

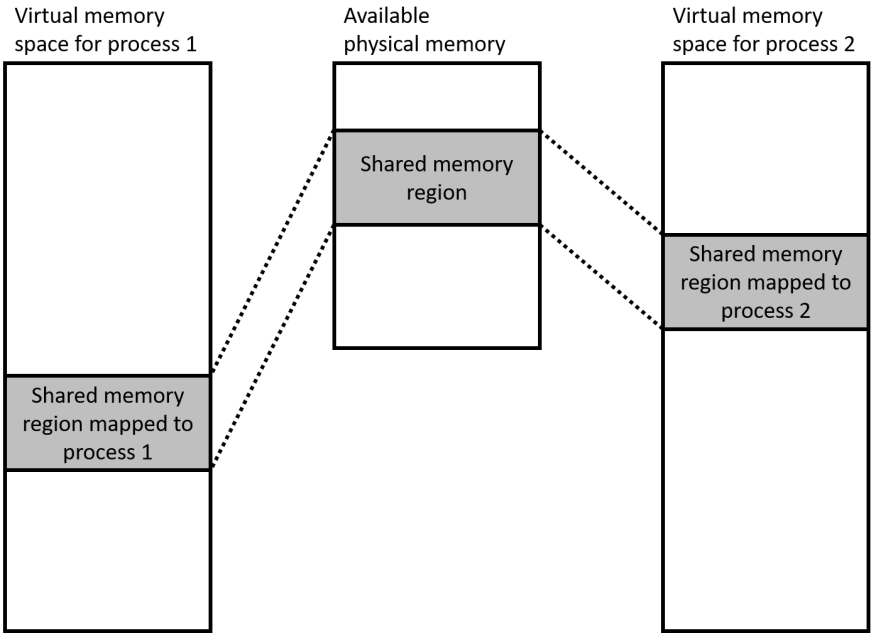
You must ensure the exercises work properly on the SoC Compute Cluster, hostnames: xcne0 – xcne7, Ubuntu 20.04, x86_64, GCC 9.3.0.

Background

Shared memory (or specifically *interprocess* shared memory) refers to memory that is simultaneously mapped into the virtual address space of multiple processes. This means that the writes to the shared memory region carried out by one process are visible to other processes.

Shared memory is especially useful for data that is intended to be used by multiple processes at once, avoiding the redundant copies needed by other modes of interprocess communication, such as pipes and sockets. Shared memory is generally very fast, because the shared memory region may be read from and written to much like normal virtual memory.

For example, in the diagram below, there is a shared memory region that is simultaneously mapped to two processes:



Reads and writes to the respective virtual addresses mapped to process 1 and process 2 will modify the underlying shared memory region in physical memory.

There are two main concerns for processes that use shared memory:

1. The virtual address of the shared memory region in process 1, in general, will be different from that of process 2. As such, pointers to the shared memory region cannot be directly shared between processes, and the content of the shared memory region usually should not contain pointers.
2. Reads and writes occur on different threads, and as such, programs must put in place some synchronization mechanism when accessing and modifying the data in shared memory.

Your Task

Your task in this lab is to write a heap allocator in shared memory, which will be used by multiple processes.

This heap allocator will support the equivalent of malloc and free operations, and such operations may be executed on any process connected to the shared memory. You will also implement a synchronization mechanism to ensure that concurrent operations from multiple processes will be executed correctly.

For each exercise, a runner has been provided for you.

Implementation instructions

The lab archive contains the following files:

- **ex0** directory:
  - **Makefile**: the Makefile (will be replaced when grading)
  - **main.c**: the file containing **main** for exercise 0 (will be replaced when grading)
  - **mmf.h**: the file containing **struct** declarations and function declarations
  - **mmf.c**: the source file containing empty function definitions for you to implement
- **ex123** directory:
  - **Makefile**: the Makefile (will be replaced when grading)
  - **runner_ex1.c**: the file containing **main** for exercise 1 (will be replaced when grading)
  - **runner_ex2.c**: the file containing **main**, as well as the code handling user commands for exercise 2 (will be replaced when grading)
  - **runner_ex3.c**: the file containing **main**, as well as the code handling user commands for exercise 3 (will be replaced when grading)
  - **ex2_sample1.in**: sample input file for exercise 2 (will be removed when grading)
  - **ex2_sample1.out**: sample output file for exercise 2 (will be removed when grading)
  - **ex3_sample1.in**: sample input file for exercise 3 (will be removed when grading)
  - **shmheap.h**: the file containing **struct** declarations and function declarations
  - **shmheap.c**: the source file containing empty function definitions for you to implement

You may modify the struct definitions or add new structs in **mmf.h** and **shmheap.h**, but you should **NOT** change the function signatures of the function definitions that are provided as the runner expects those functions to have those signatures.

To compile your program, either run **make** or invoke GCC directly. Here is an example for ex0:

```
$ gcc -std=c99 -Wall -Wextra -D_POSIX_C_SOURCE=200809L –D_GNU_SOURCE -o main main.c mmf.c
```

You may use **-Werror** to make all warnings errors. You will not be penalised for warnings, but you are strongly encouraged to resolve all warnings. You are also advised to use **valgrind** to make sure your program is free of memory errors.

Handle types

You need to write definitions for two types used in the function definitions in **`shmheap.h`** – **`shmheap_memory_handle`** and **`shmheap_object_handle`**.

The **`shmheap_memory_handle`** represents a type that points to the shared heap. This type should be pointer-like, in the sense that duplicating the handle results in a new handle to the same shared heap.

The **`shmheap_object_handle`** represents a type that points to an object in the shared heap, in a representation that may be transferred between processes (which may have mapped the shared memory at a different address). Similarly, duplicating the handle results in a new handle to the same object in shared memory. The **`shmheap_object_handle`** must additionally be designed such that retrieving an object from a different process via the handle is possible.

These handles are *opaque*, in the sense that there is no specific requirement of the content of these types, and that no pointer arithmetic will be carried out on objects of these types. It is your job to define these types appropriately.

Multiple heaps

It is possible that multiple shared heaps are used in the same process, possibly shared with different sets of processes. For all exercises, you should ensure that your implementation works even if multiple shared heaps are used simultaneously by the same process. This requirement is usually easily satisfied by avoiding the use of global variables.

Exercise 0: Memory-mapped files (1% demo)

Before we start work on the shared heap, we would like to introduce the concept of memory-mapped files. Memory-mapped files are regular files that have been "linked" to some portion of a process's virtual memory, so that all changes to that memory region are propagated into the backing file, and all changes to the file (by other processes) are propagated into the memory region.

As an example, suppose that there are two processes, A and B, and the following steps are carried out:

1. Process A creates and opens the file "test.bin", resizes it to the desired size, and maps it into memory.
2. Process A writes some data to the obtained memory address (e.g. `ptr[0] = 1;`).
3. Process B opens the existing file "test.bin", and maps it into memory.
4. Process B will see the same data in the mapped region.

5. If process B modifies any data to the mapped region, process A will also observe the changes.

Note also that it is valid for steps 2 and 3 to be swapped.

Your task for this exercise is to implement a simple wrapper to create a file (or open an existing file) and map/unmap the file into memory.

Functions that you need to implement:

`void *mmf_create_or_open(const char *name, size_t sz);` This function opens a file with the given name, or creates a new file with size **`sz`** if no such file exists yet. It then maps the file contents into memory, returning a pointer to the beginning of the file content. It is guaranteed that **`sz`** will be an integer multiple of the system page size, and that if a file already exists, **`sz`** will be equal to the file size.
`void mmf_close(void *ptr, size_t sz);` This function unmaps a file that was previously mapped into memory (**`ptr`**). It is guaranteed that **`sz`** will be equal to the file size.

As an implementation note, it is valid to close the file descriptor immediately after you map the file into memory – the file will remain mapped into memory even after the file descriptor is closed.

These syscall families may be useful:

- **`open`**
- **`close`**
- **`ftruncate`**
- **`mmap`**
- **`munmap`**

Testing your implementation

The provided runner tests your implementation in a manner similar to the example above – two processes will concurrently modify a common file and the runner checks if the processes are able to read the data written by each other.

The following output should be printed:

```
$ ./main 4096
Child A received data successfully
Child B received data successfully
```

Exercise 1: Single allocation and pointer transfer (2%)

In exercise 0, you used a memory-mapped file to pass some data between processes. Data modified by one process is written to the disk and then accessed by the other process.

In many situations, we do not actually need data to persist on the disk. For many use cases, the ability to have multiple processes share a portion of their address spaces (and hence one process can see the writes from another process) is sufficient. It is then "wasteful" to use regular memory-mapped files because reads from and writes to the disk are expensive.

To facilitate this use case, many operating systems provide a facility known as *shared memory*, which allows sharing of a region of virtual address space without the overhead of writing to disk. On Linux, shared memory is usually implemented using a virtual filesystem mounted at **`/dev/shm`**, and POSIX specifies the **`shm_open`** call that creates a file in this filesystem. This filesystem appears as a regular directory, but is stored in memory and never written to disk, giving the shared memory access speeds that are comparable to usual memory operations.

Exercise 1 requires you to set up a shared memory region, initialize a heap in that region as necessary, and design a process-independent representation of pointers to your shared memory region, leading up to the next exercise where you have to implement a memory allocation algorithm in your shared memory region.

For ease of explanation, this exercise is split into two parts:

a. Constructing and managing shared memory

b. Representing pointers in a way that can be transferred between processes

We will test the two parts together during grading. This is because we cannot generally check if you are using the shared memory correctly unless we are able to send data from one process to another.

Exercise 1a: Construction and management of shared memory

Before we can work with the shared heap, some functions are needed to set up the shared memory region managed by your heap. Your task for this sub-exercise is to implement the functions for initialisation and destruction of the shared heap.

You will also need to design the **`shmheap_memory_handle`** type appropriately – this type should behave like a pointer to the shared memory region.

Functions that you need to implement:

`shmheap_memory_handle shmheap_create(const char *name, size_t len);`

This function creates a new shared heap with the given name and size, and maps the shared heap into process memory. **`name`** should be passed as-is to **`shm_open`**. It is guaranteed that there is no existing shared memory of the same name, and that **`len`** is an integer multiple of the system page size. This function will be called just once per shared heap, before any other functions that operate on the same heap. It returns a handle to the shared heap.

`shmheap_memory_handle shmheap_connect(const char *name);`

This function opens an existing shared heap with the given name, and maps the shared heap into process memory. It is guaranteed that **`name`** is a valid shared heap that was previously created with **`shmheap_create`**. This function will be called once for each process that wants to connect to the shared heap.

`void shmheap_disconnect(shmheap_memory_handle mem);`

This function is the logical opposite of **`shmheap_connect`**. It unmaps the shared heap. It is guaranteed that the handle refers to a valid and connected shared memory.

`void shmheap_destroy(const char *name, shmheap_memory_handle mem);`

This function is the logical opposite of **`shmheap_create`**. It unmaps the shared heap, and unlinks (i.e. deletes) the shared memory with the given name. It is guaranteed that the handle refers to a valid and connected shared heap, that no other process is connected to the heap. After this call, the shared memory should not remain in the system.

`void *shmheap_alloc(shmheap_memory_handle mem, size_t sz);`

This function allocates an object of the given size on the given shared heap, similar to **`malloc`** but in shared memory. It returns a pointer to the object that was allocated. It is guaranteed that **`sz`** is no larger than **`(len - 80)`**, where **`len`** is the argument of **`shmheap_create`**. This function will only be called when **`mem`** is a valid and connected shared heap.

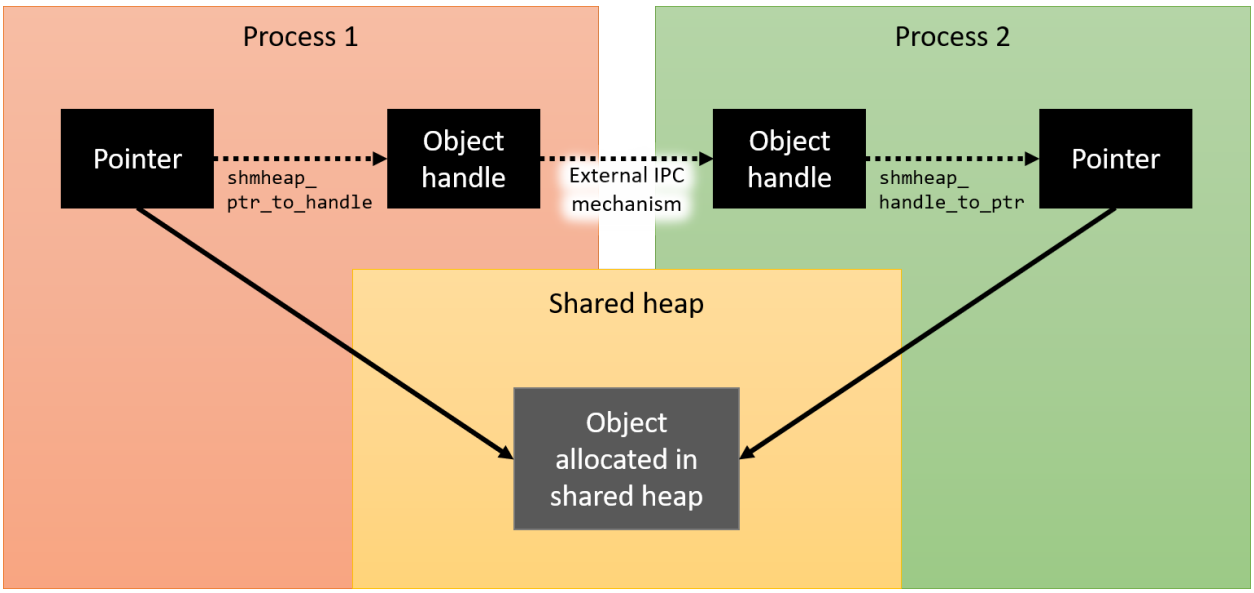
For exercise 1: We additionally guarantee that this function will be called exactly once per shared heap. As such, you do not need to implement any allocation algorithm (e.g. first fit) here. Furthermore, you do not need to worry about freeing this allocation explicitly – it suffices to let **`shmheap_disconnect`** or **`shmheap_destroy`** clean this up automatically.

These syscall families may be useful:

- **`shm_open`**
- **`shm_unlink`**
- **`ftruncate`**
- **`mmap`**
- **`munmap`**

Exercise 1b: Process-independent pointer representation

One of the important uses of an inter-process heap is for a process to read data that another process has written to. Processes accomplish this in the following manner:



Suppose process 1 has written some data in a shared object, and wants to let process 2 read it. Process 1 needs to send to process 2 a pointer to this shared object. It is sent via the following steps:

1. Process 1 converts the pointer to a sharable object handle via **`shmheap_ptr_to_handle`**.
2. Process 1 sends the handle over to process 2 through some external IPC mechanism (e.g. pipes). This IPC mechanism does a bitwise copy of the handle.
3. Process 2 converts the received handle into a pointer via **`shmheap_handle_to_ptr`**, and then reads the data from the shared object.

Your task for this part-exercise is to define the **`shmheap_object_handle`** type appropriately, and implement the functions **`shmheap_ptr_to_handle`** and **`shmheap_handle_to_ptr`**.

You do not need to implement the IPC mechanism – this would depend on the use case of your shared heap. Our runner uses pipes to send the object handles between processes.

Functions you need to implement:

`shmheap_object_handle shmheap_ptr_to_handle(shmheap_memory_handle mem, void *ptr);`

This function converts the pointer **`ptr`** (that points to an object in the shared heap given by **`mem`**) into a handle that may be transferred to other processes.

`void *shmheap_handle_to_ptr(shmheap_memory_handle mem, shmheap_object_handle hdl);`

This function converts the handle **`hdl`** (that was received from another process) into a pointer to an object in the shared heap given by **`mem`**.

Testing your implementation:

The runner for exercise 1 constructs a shared heap, allocates a single object, writes some data to it, sends the handle over to some number of reader processes, and checks that all readers see the correct data in the object.

The argument for the runner represents the number of reader processes.

The following is a possible output:

```
$ ./runner_ex1 10
Child [pid = 7376] received data successfully
Child [pid = 7377] received data successfully
Child [pid = 7378] received data successfully
Child [pid = 7379] received data successfully
Child [pid = 7380] received data successfully
Child [pid = 7381] received data successfully
Child [pid = 7382] received data successfully
Child [pid = 7384] received data successfully
Child [pid = 7385] received data successfully
Child [pid = 7383] received data successfully
```

Exercise 2: First fit memory allocation algorithm (3%)

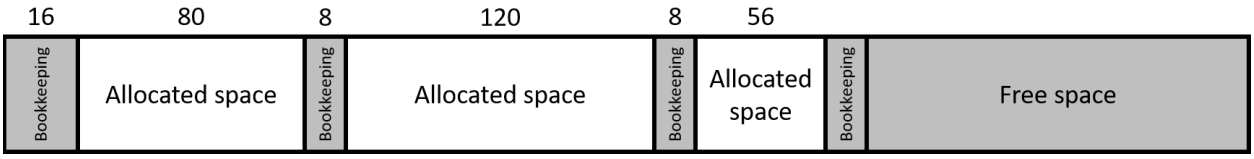
Much like a normal heap (malloc/free), users will want to allocate multiple objects on your shared heap. However, these allocations and deallocations may be requested by any process that is connected to the shared heap.

For simplicity, you will be implementing the **first-fit allocation** algorithm.

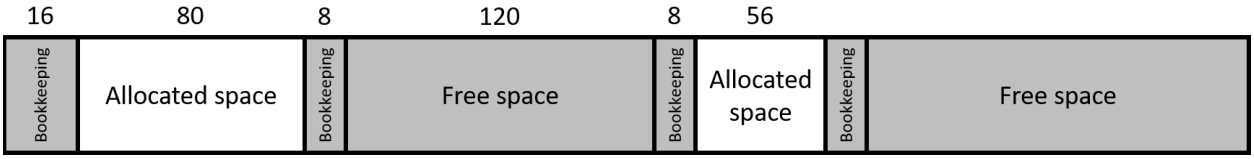
When a process requests for space for a new object, the first-fit algorithm allocates the free space closes to the start of the heap.

Consider the following example:

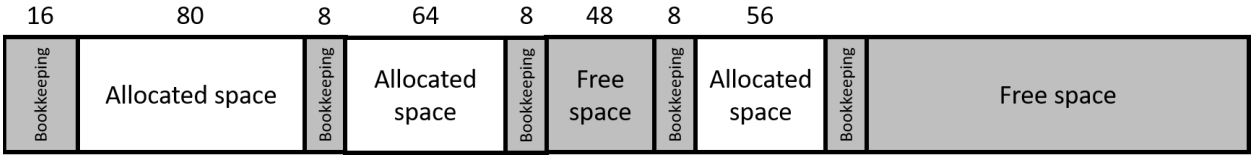
Suppose three objects are allocated (in order): 80 bytes, 120 bytes, and 56 bytes. The heap will then look like this (assuming that the initial bookkeeping space is 16 bytes, and subsequent bookkeeping spaces are 8 bytes each):



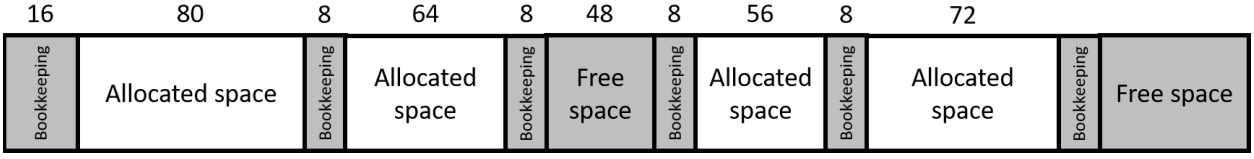
Then the 120-byte object is freed:



Then, a new object of 64 bytes is allocated:



Then, a new object of 72 bytes is allocated. Since the free space in the middle is not large enough, this object must be placed after the 56-byte object:



Then, the 56-byte object is freed, causing the two adjacent free spaces to be merged:

The bookkeeping spaces at the start of the heap and in between allocated objects are for you to store any information you need (e.g., whether a space is free or allocated). You are free to use the bookkeeping spaces as desired, subject to the following size constraints:

- The first bookkeeping space must be no larger than 80 bytes
- All subsequent bookkeeping spaces must have equal sizes, and be no larger than 16 bytes each

You will need to replace your implementation of **`shmheap_alloc`** from exercise 1 with an algorithm that does first-fit allocation, and also implement **`shmheap_free`** to free previously-allocated objects. Remember that after freeing an object, it should be merged with adjacent free space if available.

For exercise 2, we additionally guarantee that every function call to your interface is synchronised with all other function calls. This means that you do not need to implement any synchronisation mechanism for this exercise.

Memory alignment consideration

A pointer **`ptr`** is said to be **`n`**-byte aligned when **`ptr`** is a multiple of **`n`** bytes (where **`n`** is a power of 2). Alignment is important on most hardware architectures because certain hardware instructions may only work if their memory arguments are suitably aligned, and other instructions may be less efficient (i.e. take longer to complete) on unaligned memory.

For this and subsequent exercises, your shared heap is required to allocate objects at memory addresses that are **8-byte aligned** (this is equivalent to **`sizeof(size_t)`** in our testing environment).

If the caller requests for an allocation whose size is not a multiple of 8 bytes, you should round up the size to the nearest multiple of 8 bytes.

Functions you need to implement:

`void *shmheap_alloc(shmheap_memory_handle mem, size_t sz);`

This function allocates an object of the given size on the given shared heap, similar to **`malloc`** but in shared memory. It returns a pointer to the object that was allocated. It is guaranteed that **`sz`** is no larger than **`(len - 80)`**, where **`len`** is the argument of **`shmheap_create`**. This function will only be called when **`mem`** is a valid and connected shared heap. You may assume that you will not run out of memory in the shared memory region.

`void shmheap_free(shmheap_memory_handle mem, void *ptr);`

This function frees an object **`ptr`** that was previously allocated with **`shmheap_alloc`**. This function will only be called when **`mem`** is a valid and connected shared heap.

Testing your implementation

The runner for exercise 2 reads from standard input.

The first line of the input expects 3 integers, `P`, `S`, and `B`, which are the number of processes `P`, the size of the shared memory `S`, and the maximum number of objects that might be allocated `B`, respectively.

Each subsequent line is one of the following:

`0 p`	The p[th] process (0-based) calls **`shmheap_connect`**
`1 p`	The p[th] process (0-based) calls **`shmheap_disconnect`**
`2 p b`	The p[th] process (0-based) reads and prints the object associated with id `b`
`3 p b s`	The p[th] process (0-based) allocates an object of size `s` and associates it with id `b`
`4 p b`	The p[th] process (0-based) frees the object associated with id `b`

Table 1: Command format

Note: Object ids may be reused after they are freed.

The runner will create an instance of your shared heap by calling **`shmheap_create`** before executing any commands. These commands will then be executed in order. After all commands have been executed, the runner will call **`shmheap_destroy`** to clean up the shared heap.

Sample command and output

```
$ ./runner_ex2 < ex2_sample1.in
#0: Connected
#0: Allocated at offset 16: 0 1 2 3 4 5 6 7 8 9
#0: Read: 0 1 2 3 4 5 6 7 8 9
#1: Connected
#1: Read: 0 1 2 3 4 5 6 7 8 9
#1: Allocated at offset 104: 10 11 12 13 14 15
#0: Read: 10 11 12 13 14 15
#0: Read: 0 1 2 3 4 5 6 7 8 9
#0: Freed at offset: 16
#0: Allocated at offset 16: 16 17 18 19 20 21 22 23 24
#1: Read: 16 17 18 19 20 21 22 23 24
#2: Connected
#2: Read: 16 17 18 19 20 21 22 23 24
#2: Read: 10 11 12 13 14 15
#2: Allocated at offset 160: 25
#0: Read: 25
#2: Freed at offset: 160
#2: Freed at offset: 16
#2: Freed at offset: 104
```

Exercise 3: Synchronisation (1%)

In practice, multiple processes might call your functions in exercises 1 and 2 at the same time, because they may not be synchronised with one another. For example, two different processes (or threads of the same process) could call **`shmheap_malloc`** at the same time, or **`shmheap_free`** at the same time.

You will need to implement some synchronisation mechanism to ensure that your shared heap still functions correctly with unsynchronised calls to your library.

Testing your implementation

The runner for exercise 3 reads from standard input.

The first line of the input expects 3 integers, `P`, `S`, and `B`, which are the number of processes `P`, the size of the shared memory `S`, and the maximum number of objects that might be allocated `B`, respectively.

Each subsequent line contains a few commands, separated by "`|`", representing a set of commands that will be executed simultaneously.

Each command has a syntax equivalent to that of exercise 2 (shown in Table 1).

Sample command and output

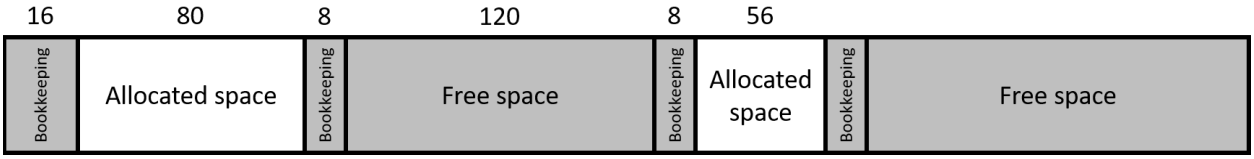
```
$ ./runner_ex3 < ex3_sample1.in
#2: Connected
#1: Connected
#3: Connected
#0: Connected
#4: Connected
#5: Connected
#2: Allocated at offset 56: 20 21 22 23 24 25 26 27 28 29
#3: Allocated at offset 144: 30 31 32 33 34 35 36 37 38 39
#4: Allocated at offset 232: 40 41 42 43 44 45 46 47 48 49
#0: Allocated at offset 320: 0 1 2 3 4 5 6 7 8 9
#5: Allocated at offset 408: 50 51 52 53 54 55 56 57 58 59
#1: Allocated at offset 496: 10 11 12 13 14 15 16 17 18 19
#0: Read: 10 11 12 13 14 15 16 17 18 19
#1: Read: 20 21 22 23 24 25 26 27 28 29
#3: Read: 40 41 42 43 44 45 46 47 48 49
#2: Read: 30 31 32 33 34 35 36 37 38 39
#5: Read: 0 1 2 3 4 5 6 7 8 9
#4: Read: 50 51 52 53 54 55 56 57 58 59
#0: Read: 30 31 32 33 34 35 36 37 38 39
#1: Read: 40 41 42 43 44 45 46 47 48 49
#2: Read: 50 51 52 53 54 55 56 57 58 59
#0: Read: 20 21 22 23 24 25 26 27 28 29
#1: Freed at offset: 144
```

```
#3: Allocated at offset 144: 60 61
#2: Freed at offset: 496
#0: Read: 40 41 42 43 44 45 46 47 48 49
#1: Freed at offset: 56
#1: Read: 60 61
#0: Read: 50 51 52 53 54 55 56 57 58 59
#0: Read: 0 1 2 3 4 5 6 7 8 9
#1: Freed at offset: 232
#2: Freed at offset: 408
#3: Freed at offset: 144
#0: Freed at offset: 320
#0: Disconnected
#2: Disconnected
#5: Disconnected
#3: Disconnected
#4: Disconnected
#1: Disconnected
```

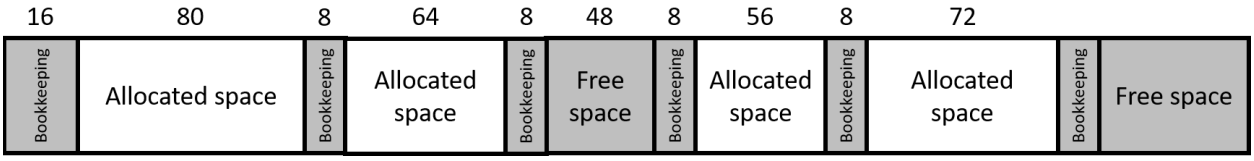
Non-determinism

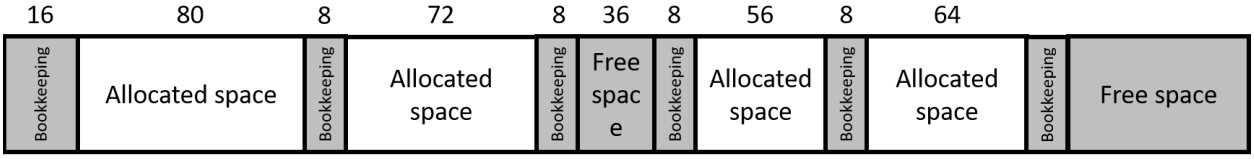
Note that due to simultaneous calls to your functions, the output may not be deterministic, so your output may not be exactly the same as our sample output.

For example, suppose that your shared heap is currently in the following state:



Then two processes simultaneously request for 64 bytes and 72 bytes of space, respectively. There are two acceptable states that your shared heap may end up in:





Either state is acceptable, so your output may differ from our sample output.

Exercise 4: Growing the heap (1% bonus)

Since the shared memory region has a size that is fixed upon initialisation, it is not easy to safely grow the shared memory while the shared heap is in use. Growing the shared memory may not be possible for all processes due to constraints on already-mapped virtual addresses, and yet moving the shared memory region to a different virtual address will break existing pointers to the shared memory.

For this bonus exercise, you task is to design and implement a way to grow the shared heap when it runs out of space. In other words, for this exercise **`shmheap_alloc`** may sometimes be called even when there is not enough space in the existing shared memory region, and you need to devise a way to add more space to your shared heap. Remember that all existing pointers to objects in your shared heap must remain valid.

You do not need to worry about shrinking the size of your shared memory region thereafter.

Testing your implementation:

The runner for exercise 4 is the same as that of exercise 3. It is up to you to design appropriate test cases for testing your ex4 implementation.

Submission and Grading

Zip the following files as E0123456.zip (use your NUSNET ID, **NOT** your student number). Use capital E as prefix.

Do **not** add any additional folder structure. Your zip file should have this structure:

- **`E0123456.zip`**
  - **`mmf.c`** (Ex 0)
  - **`mmf.h`** (Ex 0)
  - **`shmheap.c`** (Ex 1–4)
  - **`shmheap.h`** (Ex 1–4)

We will test each exercise independently, but we will deduct marks if we observe that functionalities work only partially when other exercises are tested.