CMPT 300 Assignment 5: Memory Management[1]

This assignment can be done individually or in a group (up to 3 people, please form your group in CourSys).

Total points: 100 mandatory points (no bonus).

All code **must** run on a Linux machine. We will grade your code on a Linux machine. You should create a directory for this assignment, such as *~/cmpt300/a5/* and put all files related to this assignment in it.

1. Overview

You will implement a multi-threaded memory allocator that uses some memory management techniques that were discussed in class. You solution should be based on the provided sample code.

2. Design

2.1 Initialization

The allocator needs to know the total size of memory assumed in this project and the memory allocation algorithm to be used. This information will be supplied by the following API:

```
void initialize_allocator(int size, enum allocation_algorithm);
```

In the above function, `size` indicates the contiguous memory chunk size that is assumed for the rest of the program. Any requests for allocation and deallocation requests (see point 2) will be served from this contiguous chunk. You must allocate the memory chunk using `malloc` and have the memory chunk pre-faulted and its content initialized to 0; you can do this using `memset`. The `allocation_algorithm` parameter is an enum (as shown below) which will determine the algorithm used for allocation in the rest of the program:

```
enum allocation_algorithm { FIRST_FIT, BEST_FIT, WORST_FIT };
```

FIRST_FIT satisfies the allocation request from the first available memory block (from left) that is at least as large as the requested size. BEST_FIT satisfies the allocation request from the available memory block that at least as large as the requested size and that results in the smallest remainder fragment. WORST_FIT satisfies the allocation request from the available memory block that at least as large as the requested size and that results in the largest remainder fragment.

2.2 Allocation/deallocation interfaces

The allocation and deallocation requests will be similar to `malloc` and `free` calls in C, except that they are called `allocate` and `deallocate` with the following signatures:

```
void *allocate(int size);
void deallocate(void *ptr);
```

As expected, `allocate` returns a pointer to the allocated block of size `size` and `deallocate` takes a pointer to a chunk of memory as the sole parameter and return it back to the allocator. If allocation cannot be satisfied, `allocate` returns `NULL`. Hence, the calling program should look like:

```
  int* p = (int*)allocate(sizeof(int));
  if (p != NULL) {
    // do_some_work(p);
    deallocate(p);
  }
```

2.3 Metadata Management

You should maintain the size of allocated memory block within the block itself, as described here (slide 11). The "header" should only contain a single 8-byte word that denotes the size of the actual allocation (i.e., 8-byte + requested allocation size). For example, if the request asks for 16 bytes of memory, you should actually allocate 8 + 16 bytes, and use the first 8-byte to store the size of the total allocation (24 bytes) and return a pointer to the user-visible 16-byte.

To manage free/allocated space, you should maintain two separate singly linked lists, one for allocated blocks, and the other for free blocks. When a block gets allocated (using `allocate`), its metadata (i.e., a pointer to the allocation) must be inserted to the list of allocated blocks. Similarly when a block gets freed (using deallocate), its metadata must be inserted to the list of free blocks. Note that each linked list node should only contain a pointer to the allocated memory block and a pointer to the next node in the list, because size of the allocation is already recorded with the memory block itself (previous point). The free list must never maintain contiguous free blocks (as shown in Figure 1), i.e., if two blocks, one of size m and other of size n, are consecutive in the memory chunk, they must become a combined block of size m + n (as shown in Figure 2). This combination must happen when `deallocate` is called.

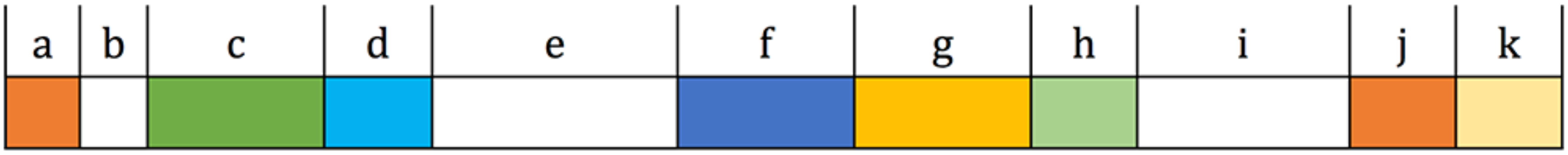

Figure 1: Each block is labeled with its size. White indicates free block while allocated blocks are colored.

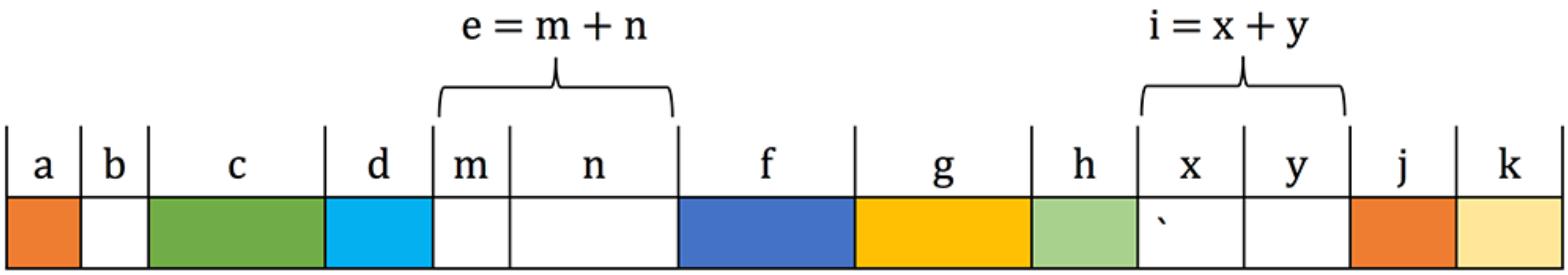

Figure 2: Example with contiguous free blocks. This should never occur as contiguous free blocks should be merged immediately (as shown in Figure 1).

2.4 Compaction

Since contiguous allocation results in fragmentation, the allocator must support a compaction API as shown below:

```
int compact_allocation(void** _before, void** _after);
```

Compaction will be performed by grouping the allocated memory blocks in the beginning of the memory chunk and combining the free memory at the end of the memory chunk (as shown in Figure 3). This process will require you to manipulate the allocated and free metadata lists. The process of compaction must be in-place, which means that you must not declare extra memory chunk to perform compaction. This can be done by going through the allocated list in sorted address order (relative to the base memory address, allocated during initialization), and copy contents of allocated blocks gradually (piece by piece) to the free space on its "left".

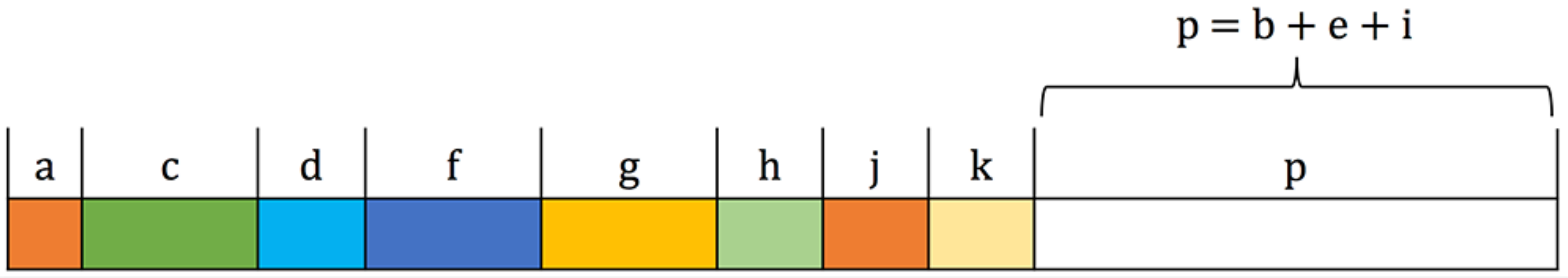

Figure 3: Result of compaction on memory chunk shown in Figure 1

As compaction relocates data, all the pointer addresses in the driver program must also be updated. Hence, the API accepts `_before` and `_after` arrays of `void*` pointers (hence they are `void**`). The relocation logic will insert the previous address and new address of each relocated block in `_before` and `_after`. You can assume that `_before` and `_after` arrays supplied by the driver program are large enough (i.e., you don't need to worry about allocation of these two arrays). The return value is an integer which is the total number of pointers inserted in the `_before`/`_after` array. This way, the calling program can perform pointer adjustment like this:

```
void* before[100];
void* after[100]; // in this example, total pointers is less than 100
int count = compact_allocation(before, after);
for (int i = 0; i < count; ++i) {
    // Update pointers
}
```

2.5 Statistics

Information about the current state of memory can be found by the following API:

```
void print_statistics();
int available_memory();
```

`print_statistics()` prints the detailed statistics as shown below (is a number):

```
Allocated size =
Allocated chunks =
Free size =
Free chunks =
Largest free chunk size =
Smallest free chunk size =
```

`available_memory()` returns the available memory size (same as Free size in print_statistics())

2.6 Multi-threading support

Your allocator must support multi-threading, i.e., it must correct allocate, deallocate, and do compaction even with mutliple concurrent threads.

You may follow the simple design that uses a global `pthread_mutex` to protect the entire allocator. That is, all the previous mentioned functions (except initialization), must first acquire the mutex before continuing to do its work, and must release the mutex before returning. You are also free to devise your own synchronization mechanism; as long as your allocator can support multi-threading, you will get the marks in this part.

2.7 Uninitialization

In order to avoid memory leaks after using your contiguous allocator, you need to implement a function that will release any dynamically allocated memory in your contiguous allocator.

```
void destroy_allocator();
```

You can assume that the `destroy_allocator()` will always be the last function call of main function in the test cases. And similar to previous projects, valgrind will be used to detect memory leaks and memory errors.

Hint: You can use your own linked list solution or the provided solution set from Assignment 1.

4. Submission

You need to submit a compressed archive named `a5.tar.gz` to CourSys that includes your modified version of the sample code (including main.c, myalloc.c, myalloc.h, and Makefile). We will build your code using your Makefile, and then run it using the command: ./myalloc. You may add more .c/.h file in your solution, and your Makefile must correctly build your solution. Please remember that all submissions will automatically be compared for unexplainable similarities.

If you did this assignment in a group, include also a simple readme text file in your package that details each group member's contributions; each of you should contribute to a reasonable share of the assignment, and every group member will receive the same marks. Please remember that all submissions will automatically be compared for unexplainable similarities.

5. Grading Policies

Make sure you are familiar with the course policies. Especially, we do not accept late submissions, so please submit on time by the deadline. You can get partial marks for coding style if your code does not compile.

---

1. Created/updated by Brian Fraser, Mohamed Hefeeda, Keval Vora and Tianzheng Wang. top