Project 5: Memory Management[1]

This project deals with memory management where you will design a contiguous memory allocator library.

Marking

This project will be graded based on correctness (90%) and correct memory management (10%). For correctness, you can assume that there will be more than 45 test cases for this project, and every test failure will cause one point deduction. For correct memory management, you will lose 5% for memory leak and 5% for memory error (such as incorrect memory access). The correct management will be checked by valgrind.

Overview

In this project, you will implement a simple contiguous allocator that supports different allocation techniques that were discussed in class. The allocator will be used as a library that exposes interface to allocate, deallocate and manage memory.

Design

1. Initializing contiguous memory allocation:
   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 _aalgorithm);
   ```

   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()`. In the above API, `allocation_algorithm` 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. The allocation and deallocation requests will be similar to `malloc` and `free` calls in C, except that they are called `kalloc` and `kfree` as shown below:

   ```
   void* kalloc(int _size);
   void kfree(void* _ptr);
   ```

   As expected, `kalloc` returns a pointer to the allocated block of size `_size` and `kfree` takes away the ownership of the block pointed by `_ptr`. If allocation cannot be satisfied, `kalloc` returns `NULL`. Hence, the calling program can now look like:

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

   Your allocator must maintain meta-data (pointer to the block and size of block) about the allocated and free blocks so that they can be used for faster allocation and compaction (as discussed in point 3). This meta-data must be held in form of `linked lists` (separate lists for allocated blocks and free blocks). When a block gets allocated (using `kalloc`), its meta-data must be inserted to the list of allocated blocks. Similarly when a block gets freed (using `kfree`), its meta-data must be inserted to the list of free blocks. 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 `kfree` is called.

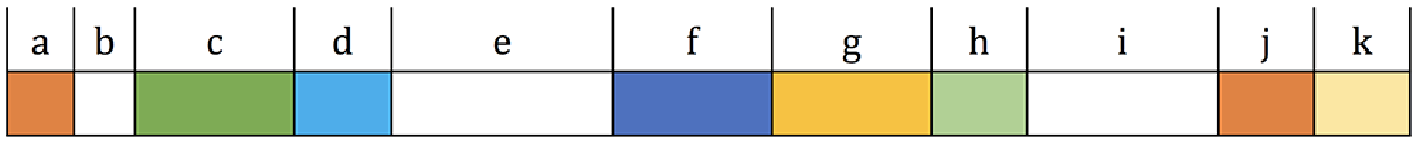

Figure 1: Each block is labeled with its size. White indicates free block while allocated blocks are colored.

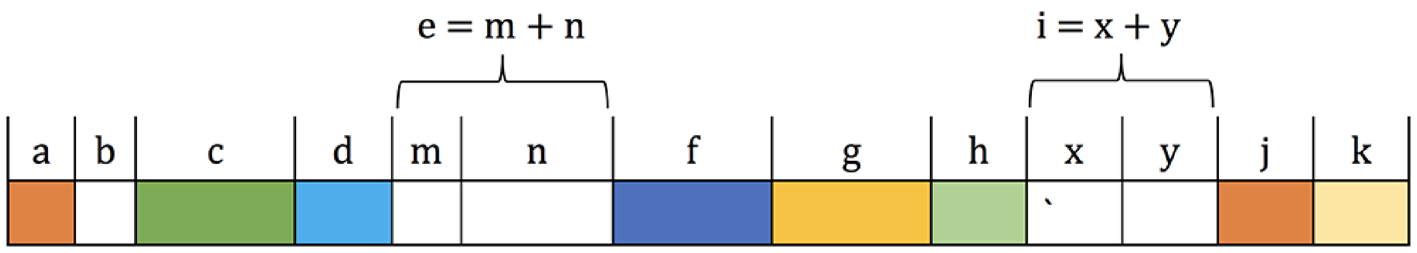

Figure 2: Example with contiguous free blocks. This should never occur as contiguous free blocks should be merged immediately (as shown in Figure 1).

3. 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 meta-data 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 a sorted manner and copying the contents of allocated blocks on free blocks from left to right.

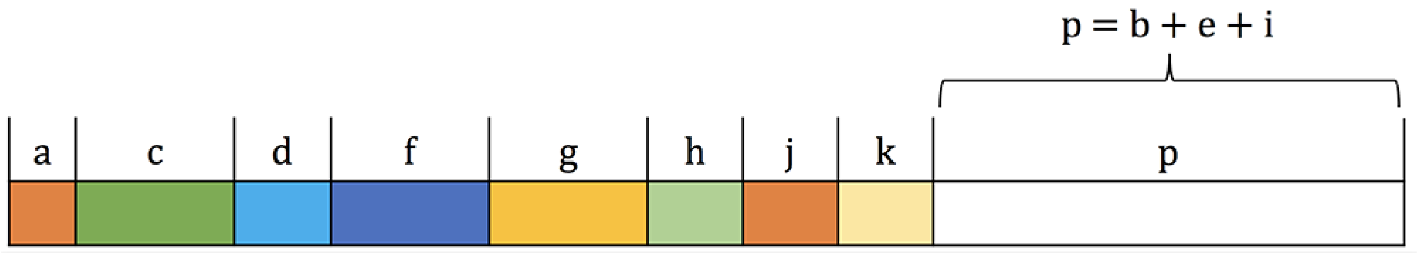

Figure 3: Result of compaction on memory chunk shown in Figure 1

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
}
```

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

```
void print_statistics();
int availableMemory();
```

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

```
Allocated size = <X>
Allocated chunks = <X>
Free size = <X>
Free chunks = <X>
Largest free chunk size = <X>
Smallest free chunk size = <X>
```

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

5. 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.

Notes

You **can** use your own linked list solution or the provided solution set from project 1.

Submission

Submit an archive `kallocation.tar.gz` of your code (including `main.c`) and make file (Sample Codes) on CourSys. We will build your code using your Makefile, and then run it using the command: `./kallocation`. You may use more than one .c/.h file in your solution, and your Makefile must correctly build your project. Please remember that all submissions will automatically be compared for unexplainable similarities.

---

1. Created by Keval Vora. top