Deliverables

You should deliver a **compressed file** containing all the files originally handed in the skeleton code, completed according to the specifications below.

Pair Programming

You are required to work with your assigned partner on this assignment. You must observe the pair programming guidelines outlined in the course syllabus — failure to do so will be considered a violation of the Davidson Honor Code. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

1 Introduction

In this project, you will implement a simulator for the main routines for memory management in an operating system. The simulator has the following features:

1. Performs frame allocation and management on a memory chunk of 4MB.
2. Emulates the x86-64 multi-level page table infrastructure.
3. Implements kernel `malloc()`, `free()`, and `realloc()`.

You cannot use `malloc()`, `free()`, or `realloc()` in any part of this project. You also cannot use the doubly linked list provided as part of the skeleton code. This list is used only in the testing programs.

2 Setup

You can implement this homework in any UNIX-like system (including FreeBSD, OpenBSD, Linux, macOS). The easiest programming/debugging setup is using VSCode, but you can use any other preferred method.

3 (25 pts) Frame Allocation

Suggested finish by: Oct 24. Tell me in office hours when you are done.

In this part, you will implement a simple frame allocator in your memory management simulator. The file `frames.c` implements all functions described in the corresponding `frames.h` header file.

`frame_init()`. Creates a bitmap that represents which frames of the memory chunk `memory` are allocated or available. A chunk of 4MB of memory has 1024 frames of 4096B (4KB) each. Your bitmap must have therefore 1024 entries (one per page), with each byte taking care of 8 entries. Hence, your bitmap should be of size 128B. The function also initializes the variables `frames_available` and `frames_allocated`.

`allocate_frame()`. Finds one empty frame and returns the *number* of the frame to the caller. Marks the corresponding entry of the bitmap with 1. Increases `frames_allocated` , decreases `frames_available`.

`deallocate_frame()`. Performs the opposite operation than `allocate_frame()`. Decreases `frames_allocated`, increases `frames_available`.

Look in the header file for this module (`header.h`), and you will have a documentation on what you should return for all the functions above if you cannot allocate or deallocate a frame.

Design a test case for the frame allocator before you move on. Deliver the test case in a file called `test_frame.c`. I'll evaluate how good of a test it is as part of the grading.

In a production operating system, you would also keep track of an additional stack of free frames, and a pointer to the first frame that has never been allocated. This way, finding empty memory frames would not be an $O(n)$ operation but instead $O(1)$. **You are not required to implement this here, but you are required to have a clear picture of how this simple modification brings the time from $O(n)$ to $O(1)$.**

4 (35 pts) Translation Table

Suggested finish by: Oct 31. Tell me in office hours when you are done.

The translation in x86-64 happens using a 4-level *tree of tables*. Each node of the tree is a table that contains 512 8B entries (therefore of size 4KB). Each of the 8B in each entry is divided as follows:

- Bits 0 to 8: flags
- Bits 9 to 11: unused
- Bits 12 to 63: address to the child node (i.e. child table) if an internal node; or physical address being mapped to if a leaf node.

So, in this tree, the root points to a 512-entry table (a "node"), and each entry points to another child 512-entry table (another "node"). Lets number the levels from 4 (top) to 1 (leaves). To translate an address, you look at its first 48 bits, and, within this 48-bit sequence, the first 9 bits are used to index an entry in the level 4 table (the root node), pointing to a particular level 3 table. The next 9 bits are used as an index the level 3 table above, pointing to a particular level 2 table, and so on as described in Fig. 1.

In essence, the first **36 bits** of the virtual address (**which is the page number**) are used to **index** a last-level table. The last 12 bits of the virtual address represent the **offset** within a page (or within its translated frame).

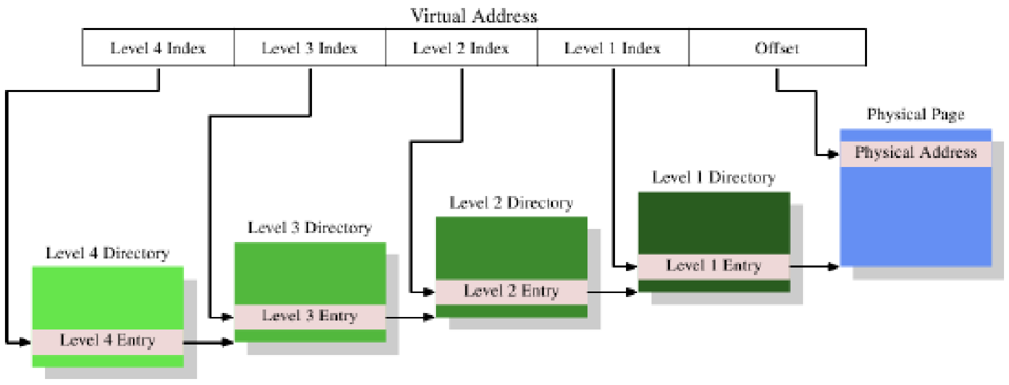

Figure 1: x86-64 indexing structure. Credit: LVM.net.

This all happens in *hardware*, but we are going to simulate this process in software.

4.1 Our Own Implementation

I give you a struct implementing each entry in the table. It's called (guess) `entry`, and it is defined in `translation.h`. To allocate 512 entries, you cannot use `malloc()`, since you are implementing it! You should:

- Allocate one frame, get the frame number. Note that one frame is 4K, conveniently the size of 512 8B entries.
- Calculate the address of the frame, and make a pointer to it:

```
frame_number = allocate_frame(1)
frame_address = FRAME_ADDRESS(frame_number)
entry *table = (entry *) frame_address
// Now use table[0] to table[511]
```

The nodes in the table should be constructed on demand. So your root pointer should be initialized to NULL, and upon any mapping attempt using `vm_map()` function, the `page` parameter (**which is already the 36-bit page number**) should be used to construct only the necessary path down the tree from root. The `frame` parameter **is also already the 36-bit frame number**. In the leaf node, the bits 12 to 63 of the table entry should finally point to the **frame address, not frame number**.

You should use the field `flags` within each entry to indicate if that entry contains a valid address or not. In other words, you will need at least a "used" flag for each entry. Make sure to initialize all the "used" flags of a freshly created node table with false. You are free to define the bit structure of your `flags` field – in other words, you define which bit is set to indicate an used entry.

Here are the functions in the module:

`vm_map(p, f, number, flags)`. Traverses down the translation tree as described above, and maps a page number p to the address of frame number f. If a path does not exist, it will be created on demand by allocating frames using `allocate_frame()`. The tree elements are linked using the addresses of the frames:

- In our simulator, the address of frame i is `memory + (i * 4096)`.
- In a real kernel, the address of frame i is `(i * 4096)`.

The last-level entry also points to the address of the frame f it maps the page p to.

Note. You will need to collect the first group 9 bits, the second group of 9 bits, etc, using bit masking. The rightmost bit is at position 0. You should index the root table with bits at positions 27-35, the next table with bits at positions 18-26, the next table with bits at positions 9-17, and the last-level table with bits at positions 0-8 (the rightmost bits).

Please look at the documentation of the function in `translation.h` and `translation.c` for further details.

`vm_unmap()`. Traverses down the translation tree as described above (Fig. 1), and unmaps a page number p. You should not destroy paths in the translation tree even if all their entries become unused, but you should make entries unused by setting the appropriate bit in the `flags` field **of the last-level entry only**.

Please look at the documentation of the function in `translation.h` and `translation.c` for further details.

`vm_locate()`. Traverses down the translation tree and finds the first page (starting from zero) that does *not* have an address associated with it, and that has at least "`number`" pages ahead that are also unmapped. You will have to use the "used" bit you defined (as described above) to differentiate between used and free entries.

Please look at the documentation of the function in `translation.h` and `translation.c` for further details.

`vm_translate()`. Traverses down the translation tree as described above (Fig. 1), and maps a page number p to a frame number f.

Please look at the documentation of the function in `translation.h` and `translation.c` for further details.

Design a test case for the translation table module before you move on. Deliver the test case in a file called `test_translation.c`. I suggest a simple test where you map each of the 1024 frame *numbers* in `memory` to their physical addresses. Translating an address (in binary) 1000000000001 (frame 1, byte 1) should give you the address of the second byte in the second frame.

5 (45 pts) Kernel Memory Allocation

`palloc()`. Allocates a 4K fram, maps a page number to the frame address, and returns the page (virtual) address that resolves to that allocated area. This function should:

1. Allocate `number` frames of memory

```
frame_number = allocate_frame(...)
```

2. Find the first page number (virtual) that is not mapped to a frame (physical), and `number` of them are consecutive

```
page_number = vm_locate(...)
```

3. **Ignore** the result of the previous call (for the sake of simulation, see note below), and map the frame number to its own address:

```
page_number = frame_number
vm_map(page_number, frame_number, number, /* extra flags */)
```

4. Return the address of the first byte of the allocated page.

```
return PAGE_ADDRESS(page_number);
```

   The macro `PAGE_ADDRESS` should be defined in `translation.h` and works similarly as `FRAME_ADDRESS` in `frame.h`.

   Additional deliverable: Leave a comment with about 4 lines on top of the macro explaining (i) why the page address is calculated by multiplying by 4096; (ii) why in our case we sum the base memory address to it; and (iii) why in a real kernel we would not sum the base memory address to it.

You are simulating the hardware automatic address translation in software. The `palloc()` function would return the address of the first virtual (mapped) address, based on the page number of step (2), but your application would not understand this since it only understands its own application addressing. Therefore, you are **simulating** that there is another layer of automatic virtual addressing by always mapping page number X to the address of a frame number X, regardless of the result of step (2). Inside a real kernel, you wouldn't perform the overwrite of the page number in step (3), but the hardware would be performing the automatic translation between `page_number` to `frame_number`.

Make sure, that in the end of this function:

```
vm_translate(page) is GET_ADDRESS(frame)
```

Please look at the documentation of the function in `kmalloc.h` and `kmalloc.c` for further details. **Implement this function before implementing `malloc()`, `realloc()`, or `free()`.**

`pfree()`. Frees chunks of 4K of memory previously allocated with `palloc()`. The `address` parameter is a page (virtual) address. This function should:

1. Calculate the page number (virtual) from the page address (virtual) – i.e., which of the 1024 chunks of `memory` does the address belong to.

   The macro `PAGE_NUMBER` should be defined in `translation.h` and works similarly as `FRAME_NUMBE` in `frame.h`.

   Additional deliverable: Leave a comment with about 4 lines on top of the macro explaining (i) why the page address is calculated by dividing by 4096; (ii) why in our case we subtract the base memory address to it; and (iii) why in a real kernel we would not subtract the base memory address to it.

2. Translate the page (virtual) number to a frame (physical) address using `vm_translate ()`.

3. Unmap the page (virtual) number from the frame (physical) address.

4. Deallocate the frame that has just been unmapped.

5. Repeat the procedure for the number of times provided in the parameter `number`.

Please look at the documentation of the function in `kmalloc.h` and `kmalloc.c` for further details. **Implement this function before implementing `malloc()`, `realloc ()`, or `free()`.**

At this point, you are able to allocate and deallocate exact chunks of 4K. The following section describes the mechanism that you are going to use to allocate and deallocate arbitrary sizes. The functions that are going to be implemented are the ones below:

`kmalloc()`. Provides the same functionality as `malloc()`, but implements the heap management manually as described below.

All the documentation for this function has been provided in `kmalloc.h` and `kmalloc.c`. Please refer to those files for further details.

`krealloc()`. Provides the same functionality as `realloc()`, but implements the heap management manually as described below.

All the documentation for this function has been provided in `kmalloc.h` and `kmalloc.c`. Please refer to those files for further details.

`kfree()` .Provides the same functionality as `free()`, but implements the heap management manually as described below.

All the documentation for this function has been provided in `kmalloc.h` and `kmalloc.c`. Please refer to those files for further details.

5.1 Heap Management

To manage the space available in the heap, you are going to implement a *free list* as indicated in the figure below. Each node of the list contains the starting address of any free chunk of space, and the length of that chunk. Please refer to Fig. 2.

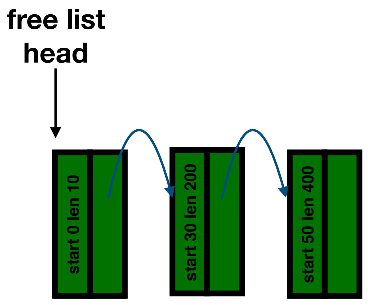

Figure 2: Representing (abstractly) a list of free memory regions.

In the list above, if the user wants to allocate 5 bytes, you source the storage from one of the free chunks. One possible outcome is depicted on Fig. 3.

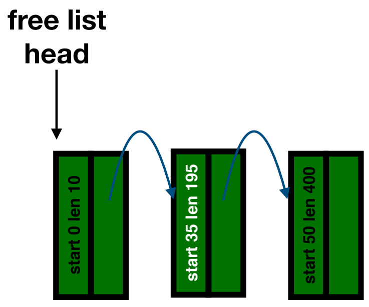

Figure 3: Free list after allocating 5 bytes. We could theoretically allocate these bytes from any of the regions in Fig 2, in the beginning or the end of the region.

You are not allowed to dynamically allocate nodes in the list (i.e. use `malloc()` and `free()`, because you are just implementing those functions. **You can however use `palloc()` and `pfree()`, because you just implemented them in terms of lower-level virtual memory operations.**

5.1.1 Bookkeeping on Allocated Regions

When users call `kfree()`, your implementation has to figure out two things:

1. How much space needs to be freed, and returned to the free list as a new free node;
2. If the address that the user provided to the `kfree()` call has been previously allocated with `kmalloc()`.

The solution to this problem is that whenever the asks to allocate 100B, the system actually allocates `100 + 2*sizeof(uint64_t)` bytes. The extra two bytes in the front of the allocated region contain (i) its size; (ii) a magic number (say `0xc0dec0de`); and (iii) the storage. Look at any red area of Fig. 4. The `kmalloc()` call returns then a pointer to the storage, by adding `2*sizeof(uint64_t)` to the base address of the red area. The `kfree()` call gets the address from the user, subtracts `2*sizeof(uint64_t)` bytes from it, and figures out both the size of the storage section (call it s) and checks if magic number matches the predefined one. If the magic numbers match, the user called `kfree()` in a valid region, and the system returns `100 + 2*sizeof(uint64_t)` bytes back to the free list.

Note that C performs *pointer arithmetic*. So, if your address is already a pointer to a `uint64_t`, summing **one** will move the address forward as many bytes as needed to store a `uint64_t`. Specifically:

```
// How to give the user 100B
uint64_t *address = //find 100 + 2*sizeof(uint64_t) bytes available
*( address ) = 100
*( address +  1 ) = 0xc0dec0de
```

So it's a good idea to get whichever address that you get after finding the empty space, casting it to a `uint64_t` and performing the operations above. You could also use `char *` pointers. The logic is a bit more complicated but it performs the same task:

```
// How to give the user 100B
char *address = //find 100 + 2*sizeof(uint64_t) bytes available
*( (uint64_t *) (address) ) = 100
*( (uint64_t *) (address + sizeof(uint64_t) ) = 0xc0dec0de
```

5.1.2 Packing the Free List on `palloc()`'ed Areas

Since you don't have `malloc()` or `free()` available to implement the free list, you are going to *pack* the free list in areas allocated with `palloc()`, and *manage free space together with allocated space.*

An area of memory looks like Fig. 4. The green areas represent available space (represented by *free nodes*), and the red areas represent allocated space (represented by *used nodes*). You shoud keep a single pointer to the beginning of the free list.

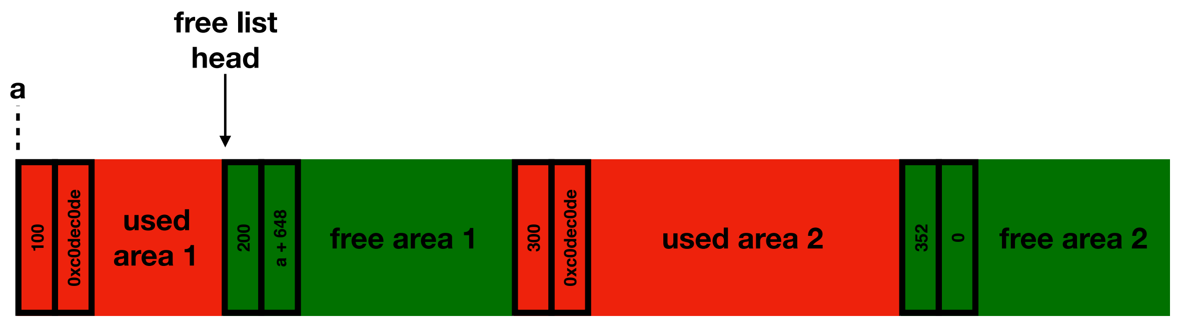

Figure 4: Embedding the free list within the areas allocated to the application. We are assuming only for the sake of simplification that the frame has 1000 bytes and is located at memory address a.

In your implementation, the initial free list should contain only a single node indicating that a chunk of `4K - 2*sizeof(uint64_t)` is free. Make sure to allocate that chunk using `palloc()` in the beginning of your program.

Here are some requirements to have in mind when you implement your free list within areas allocated with `palloc()`.

- The first 64-bit integer of the header in a free node (a green area above) represents the size of the available region, and the second 64-bit integer represents the address to the next free node.
- The first 64-bit integer of the header in a used node (a red area above) represents the size of the allocated region, and the second 64-bit integer represents the magic number.
- If the user requests memory than cannot be fitted in the current page frame, a new page frame is created, and the free list is extended. See Fig. 5.
- If the user calls `kfree()`, you have to *coalesce* free the free if two consecutive free areas are formed. See Fig. 6.

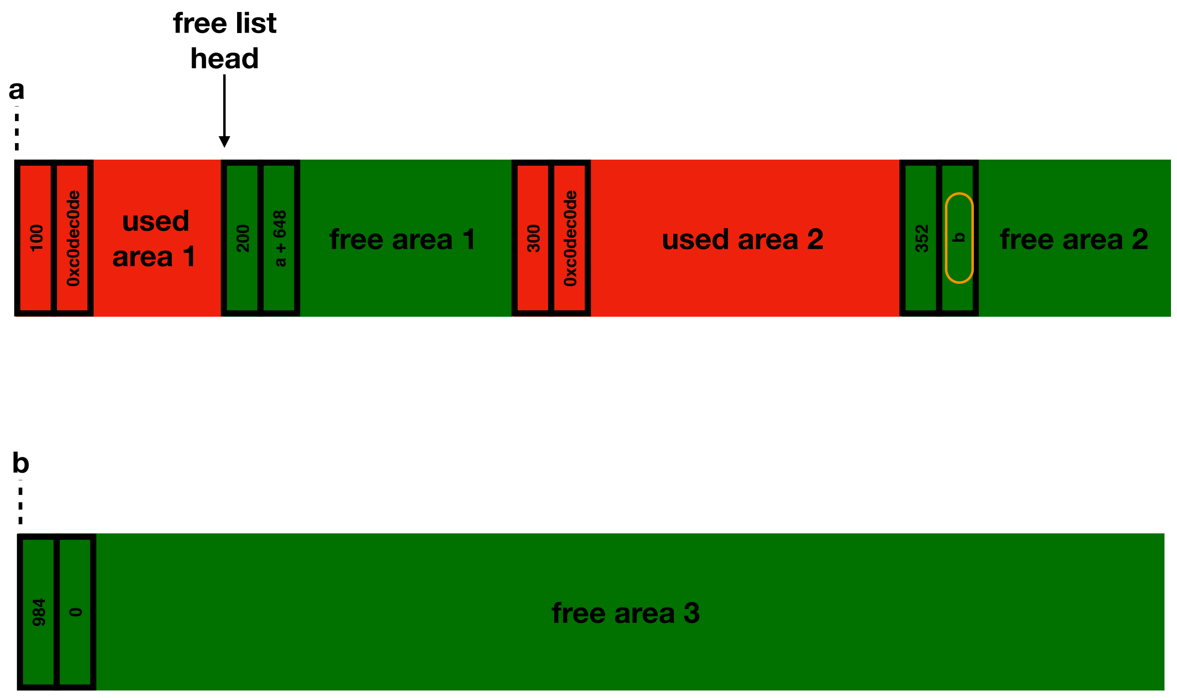

Figure 5: When a new frame needs to be allocated with `palloc()` (say, because we needed to allocated more memory than what was available in the free list), we extend the free list's last element to point to a whole new page containing a single free node.

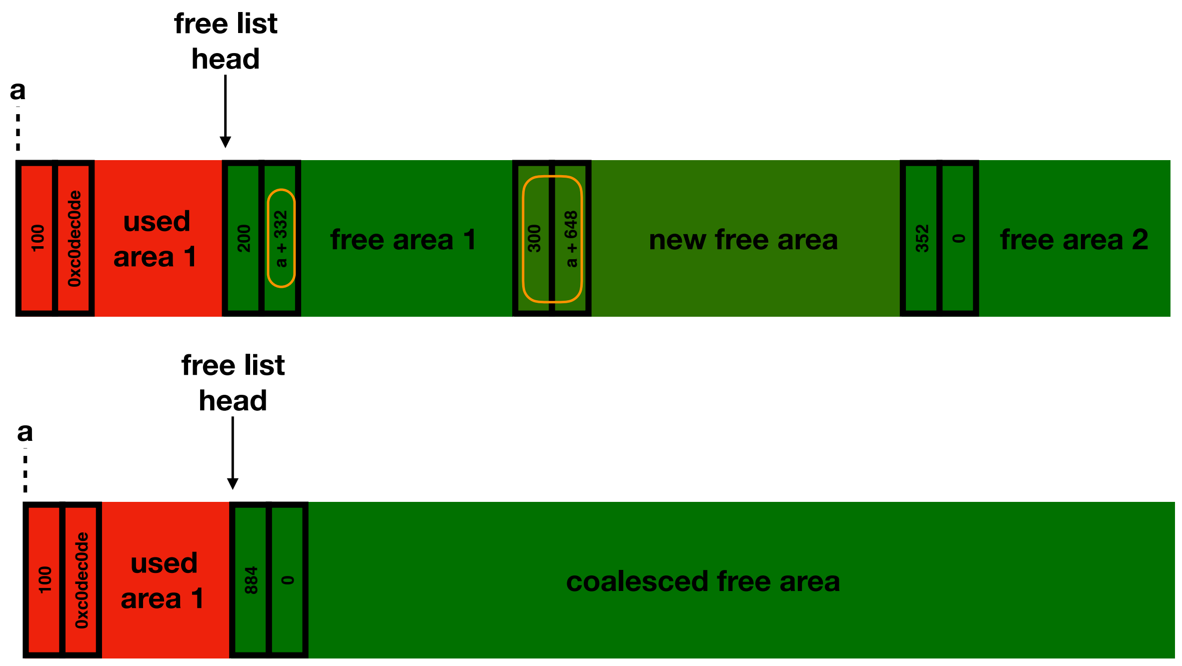

Figure 6: Coalescing free regions inside a single frame allocated with `palloc()`.

- When the user calls `kmalloc()`, you have to use the *first fit* strategy to find an empty free node in the list: navigate the free node list sequentially and find the *first* entry that allows you to allocate a chunk of memory. This should create external fragmentation on the heap, but we don't perform memory compaction because that would change the addresses of memory chunks in the application!

Note: To deal with external fragmentation, we have alternative schemes to allocate used nodes within the free node list. One scheme is called *worst fit*, which minimizes the number of very small chunks by allocating the used node within the largest free node; and *best fit*, which reduces the size of the very small chunks, by allocating the used node within the free node that is closest in size to the allocated chunk.

6 Style Deductions

I may deduct up to 15% of every grade item to account for bad style, which includes:

1. Poor indentation;
2. Cryptic variable names;
3. Poor error treatment;
4. Naming style inconsistencies (adopt one style with your partner and stick to it);
5. Overly-complicated code.

Good luck,
- Hammurabi