

# MP2: Frame Manager

Caleb Frye

UIN: 631004185

CSCE 410: Operating Systems

## Assigned Tasks

**Main:** Completed, completely implemented Frame Manager including using two bits to store frame state and testing with two pools.

## System Design

**Files modified:** `cont_frame_pool.C`, `cont_frame_pool.H`, `kernel.C`

The goal of Machine Problem 2 was to design a *frame manager* for an existing kernel. This frame manager handles allocating frames, which are sections of physical memory that get mapped to sections (pages) in virtual memory. Different parts of the operating system use different parts of memory, so the frame manager will allocate frames in frame *pools*, which are contiguous sections of frames. Each frame pool handles getting and setting of the frames that it manages. Each pool stores state information about its frames, i.e. whether they are Free, Used, or Head-of-Sequence (HoS). The first frame in an allocated sequence within a pool is marked HoS, and subsequent frames in the sequence are marked as Used. I chose to use a bitmap to store this information, which allows the state of each frame to be stored using only two bits. This means that a single frame can store state information for 16,384 frames (4KB frame size \* 4 frames per byte).

## Code Description

The frame manager is implemented with the class **ContFramePool**. This class supports allocation and release of the frames managed by each frame pool. Each ContFramePool object can be thought of as an individual frame pool, and can only manage the frames owned by that specific ContFramePool object. A frame pool needs to store management information about itself somewhere, and this can be in one of its own frames, or a frame in another frame pool.

I modified `cont_frame_pool.C`, `cont_frame_pool.H`, and `kernel.C` for this machine problem. I changed `kernel.C` by uncommenting the process frame pool code which was commented out. I did this to test my frame manager with two pools for future machine problems, and my implementation handled it with expected behavior. I also added a call in `kernel.C` to the `print_pool_info()` function which I wrote to show information about the two frame pools, and I manually marked some frames in the process pool as Used and then Free to further test functionality. You can see this in the following image:

```

/* -- TEST MEMORY ALLOCATOR */

test_memory(&kernel_mem_pool, N_TEST_ALLOCATIONS);

test_memory(&process_mem_pool, N_TEST_ALLOCATIONS);

//we can see in the output that the frames I allocate here
//are properly released, because the process pool only has
//the 256 frames from the hole still Used after the program finishes
process_mem_pool.mark_inaccessible(7239, 100);
process_mem_pool.release_frames(7239);

/* --- Add code here to test the frame pool implementation. */
ContFramePool::print_pool_info();
/* -- NOW LOOP FOREVER */
Console::puts("Testing is DONE. We will do nothing forever\n");
Console::puts("Feel free to turn off the machine now.\n");

```

Figure 1: Manually Using/Freeing frames and calling print\_pool\_info() in kernel.C

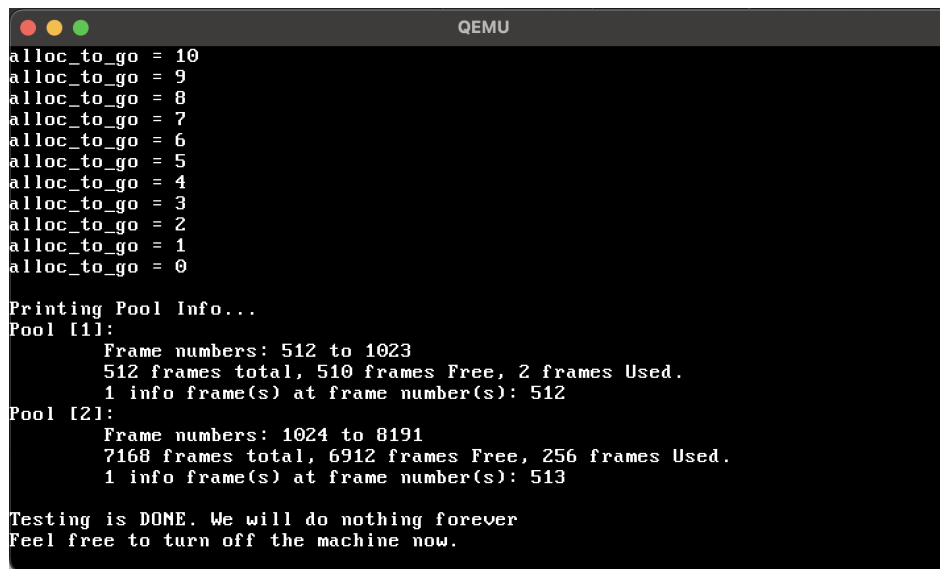
In order to compile the code run:

```

make
make run

```

This will start the operating system and create two frame pools: a kernel frame pool and a process frame pool. The system will then allocate and deallocate frames in each of the two frame pools in order to test for proper functionality, and will print information about both pools once this process is completed. You should see the following output:



```

QEMU
alloc_to_go = 10
alloc_to_go = 9
alloc_to_go = 8
alloc_to_go = 7
alloc_to_go = 6
alloc_to_go = 5
alloc_to_go = 4
alloc_to_go = 3
alloc_to_go = 2
alloc_to_go = 1
alloc_to_go = 0

Printing Pool Info...
Pool [1]:
    Frame numbers: 512 to 1023
    512 frames total, 510 frames Free, 2 frames Used.
    1 info frame(s) at frame number(s): 512
Pool [2]:
    Frame numbers: 1024 to 8191
    7168 frames total, 6912 frames Free, 256 frames Used.
    1 info frame(s) at frame number(s): 513

Testing is DONE. We will do nothing forever
Feel free to turn off the machine now.

```

Figure 2: Program Output

We can see the information about each frame pool printed in the console. Pool [1] is the kernel frame pool, and pool [2] is the process frame pool. The kernel frame pool has 2 frames used, one info frame for itself (frame 512) and one for the process frame pool (frame 513). The process frame pool has 256 frames marked as Used because of the memory hole which we define in kernel.C.

All of the following functions are defined in `cont_frame_pool.C`:

**ContFramePool::ContFramePool():** Constructs the ContFramePool object. Sets the base frame, number of frames, and info frame number of the frame pool based on the constructor parameters. If `_info_frame_no = 0`, the ContFramePool will store state information internally (in one of its own frames). Otherwise, frame state information is stored externally in frame number `_info_frame_no`. The constructor will initially set all frames to Free, then set the first frame to Used for the info frame, if that option is chosen. It also initializes the static linked list of frame pools if it is uninitialized, otherwise adds the new frame pool to the list.

**ContFramePool::get\_frames():** This method takes the number of requested frames as its argument, and returns the frame number of the first frame in the sequence, if the number of frames was able to be contiguously allocated. It loops through the frames of the given frame pool until it finds a sequence of free frames with length equal to the input value. It marks the first frame in this sequence as HoS (Head of Sequence), and returns the frame number of the new HoS frame. It makes use of the `mark_inaccessible()` function to mark the new sequence of frames as Used.

**ContFramePool::mark\_inaccessible():** This method is similar to `get_frames()`, except it does not search for a contiguous section of Free frames. Instead, it takes in a base frame number and a number of frames, and will always set that number of frames as Used, starting from the base frame. It checks to make sure that the frames the user wants to mark are actually owned by the given frame pool before marking them. Once this check is done, it uses the `set_state()` function to set the sequence of frames as Used.

**ContFramePool::release\_frames():** This method takes in a first frame number, and will release the contiguous section of Used frames starting from that first frame number. This method is static because it needs to find the frame pool which owns the frame with the given frame number. It does this by looping through the linked list of existing frame pool objects and checking the range of each one by doing calculations with the base frame and frame number of each frame pool. Once it finds the correct frame pool, it uses the `set_state()` function to set frames to Free, until it finds a frame marked as Used or HoS.

**ContFramePool::needed\_info\_frames():** Simple function which takes in a number of frames, and returns the number of frames needed to store management information for that number of frames. It takes into account that 2 bits are required to store state information for one frame. It does  $(\text{number of frames desired}) \div (\text{number of frame states stored per frame})$  (this is about 16k, see System Design section). It also handles remainders, adding an additional frame for them. The number of frames stored per frame is  $4 * 4\text{KB}$ , because we have 4KB to use and we can store 4 frame states per byte.

**ContFramePool::get\_state():** Helper function which abstracts away the bitmap, and makes it much easier to manage frame states. Takes in a frame number, and sets that frame number's state in the bitmap. Uses division to find the byte index of a particular frame, and uses bit manipulation to access each group of 2 bits inside a byte of the bitmap. The following bits map to each state:

- 00 - Free
- 01 - Used
- 10 - Head of Sequence (HoS)

**ContFramePool::set\_state():** Helper function which works very similarly to `get_state()`, but instead of setting the state, it takes in a frame number and returns the state of that frame. It also uses bit manipulation to access each group of 2 bits inside a byte of the bitmap.

**ContFramePool::print\_pool\_info():** Static function which prints (using `Console::puts()` and `Console::puti()`) the following information about every frame pool object that currently exists. I used this to ensure proper functionality of my system. See Figure 2 for what the output looks like :

- Frame numbers: range of frame numbers managed by this pool
- Number of total, Free, and Used frames in this pool
- Number and frame number of info frame(s) for this pool

## Testing

I made use of the provided process frame pool code to test my system, which was commented out in the starter code. I also used manual allocation/deallocation in `kernel.C` to further test my system. Combined with this, I believe the provided `test_memory()` function does a sufficient job of testing the system. It allocates frames, checks that the allocated memory contains the correct values, then deallocates the memory. This function combined with my `print_pool_info()` function shows how the system correctly allocates and then deallocates all of the memory it uses, since we can see in Figure 2 that the only two frames marked as Used after the program finishes are the two info frames, one for each frame pool. We can also see how the process frame pool's info frame is correctly stored externally, in the kernel frame pool. Since we can assume a user will use the functions in the way they are intended, there is no need to do any more exhaustive testing. For safety I have also added `assert(false)` statements in `get_frames()`, `mark_allocated()`, and `release_frames()` that will terminate the program if any of the following happens:

- the user requests more frames than there are available in the frame pool
- the user request an amount of frames which there is not a large enough contiguous section to allocate
- the user calls `get_frames()` or `mark_inaccessible()` with a frame number not managed by the `ContFramePool` object they called the function on
- the user calls `get_frames()` or `mark_inaccessible()` on a valid frame, but request a number of frames that would go beyond the scope of the given frame pool
- the user attempts to `mark_inaccessible()` on a base frame which is already set to Used or HoS
- the user attempts to `release_frames()` that are not managed by any frame pool