# MP2: Simple File System

Kai-Chih Huang
UIN: 333000021
CSCE611: Operating System

## Assigned Tasks

Main: Completed.

## System Design

The goal of this machine problem is to design a memory frame manager. I used bitmap to store information of each frame. However, what is different from examples from SimpleFramePool is that I used 2 bits to represent three frame states, so that it can manage contiguous frames (detail described in following section).

Also, I used a static linked list to keep record of each pool that was created, so that when the release function is called, we can figure out which pool is the releasing frame belongs to.

When a ContFrampool object is created, it basically used a create a bitmap to keep track of the usage of frames in the pool. Every time the frame is requested, read, or released, we go to the bitmap then check status or change status from it.

## Code Description

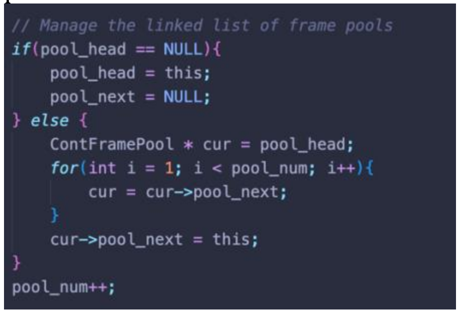
I changed cont_frame_pool.C and cont_frame_pool.H for this machine problem. To compile code, simply run following command lines under MP2_Sources directory:

```
$ make clean    // clean the old compile files before we compile

$ make          // compile files

$ ./copykernel.sh (if permission denied, try chmod u+x ./copykernel.sh)    //copy kernel

$ bochs -f bochsrc.bxrc     // run bochs
```

Since cont_frame_pool.H is the header file of cont_frame_pool.C, I will walk through the functions/methods defined in cont_frame_pool.C. My implementation is to modify codes of SimpleFramePool, so I will focus on explaining the difference I made from it.

**ContFramePool(unsigned long _base_frame_no, unsigned long _n_frames, unsigned long _info_frame_no)**
Initializes the data structures needed for the management of this frame pool. The difference from SimpleFramePool is that I added pool_head, pool_next and pool_num to manage multiple pools (e.g. kernel and process pool). Pool_head is a static pointer that always points at the first pool, each pool has pool_next pointer to point to next pool that is created, and pool_num is a static integer variable that records how many

pools were created.



```
// Manage the linked list of frame pools
if(pool_head == NULL){
    pool_head = this;
    pool_next = NULL;
} else {
    ContFramePool * cur = pool_head;
    for(int i = 1; i < pool_num; i++){
        cur = cur->pool_next;
    }
    cur->pool_next = this;
}
pool_num++;
```

## set_state(unsigned long _frame_no)

Set the state of given frame. What is different from the SImpleFramePool is that we use three states: Used, Free, HoS(head of sequence). Since there is three states, we used 2 bits to store the states of each frame: State 0 is Free, state 1 is Used, and state 2 is HoS. To set the state, first we set the desired bits to 00, then XOR with a mask of each state to set to correct value.

## get_state(unsigned long _frame_no)

Get the state of given frame. We set mask to 3 (binary 11) then left shift to desired position, then use bitwise and to extract the states of the frame.

## get_frames(unsigned int _n_frames)

Allocates a number of contiguous frames from the frame pool. My algorithm is simply start from the beginning of the frame pool and walk through each frames, if the frame is free, then we start count contiguous free frames, if total count >= required frames, then set those frames to used then return

## mark_inaccessible(unsigned long _base_frame_no, unsigned long _n_frames)

Basically the same as implemented in SimpleFramePool, I just change setting the state of the first frame to HoS

## release_frames(unsigned long _first_frame_no)

Since we don't know which pool does the given frame belongs to, we first need to find the correct pool. Since we have record of pointers to each pool, we simply walk through each pool, and check if the given frame number falls in the range of the current pool. If so , we call that pool's private function **release_frame_in_pool** to release.

## release_frame_in_pool(unsigned long _first_frame_no)

We already know this given frame belongs to the ContFrampool object that calls this function. First we check if given frame state is HoS. If not, the frame is not the start of a chunk of memory, so we raise an error. Then, we simply start from HoS to mark the state all the contiguous frames to Free until the state of the next frame is Free or we reached the end of the pool.

## needed_info_frames(unsigned long _n_frames)

Returns the number of frames needed to manage a frame pool of size _n_frames. What is different from the SimpleFramePool is that we use 2 bits to store status of 1 frame. Therefore, given 4KB = 4096*8 bits frames size in x86, we can store states of (4096 * 8) / 2 = 16384 frames in 1 frame. Then we calculate the ceiling of _n_frames / 16384 to get number of frames needed.

## Testing

Aside from using the provided test_memory function, I run tests on following scenarios when implemented the functions.
When allocating memory (get_frames())

1. _n_frames is greater than current free frames
2. Can't find contiguous free _n_frames amount of frames

When releasing memory

3. the given frame is not the head of a sequence