# MP2: Frame Manager

Po-Chieh Wang
UIN: 933004975
CSCE 611: Operating System

## Assigned Tasks

Main: Completed

## System Design

The goal of this machine problem is to build a frame manager. So when a program requests some frames, the frame manager can find out which frames could be assigned and mark these frames as used. First, I used a linked list to connect pools. Every time when a pool is created, it will automatically connect to the second last one. When a process requests some frames, the frame manager will check its map and find enough continuous frames and assign them, then it will mark the head of the frame as HoS  and the rest of these frames as used. Regarding the map, two bits were used to store the state of the frame. HoS is 0x10, Free is 0x01 and Used is 0x00. When the frames need to be released, the release function will first traverse the linked list and find the frame manager which is managing these frames. Then this frame manager will release these frames and mark these states as free.

## Code Description

To finish this machine problems, I have changed cont_frame_pool.H and cont_frame_pool.c.

**Cont_frame_pool: ContFramePool:** I used a linked list to connect all the pools, there is a static pointer head points to the firs pool, a pointer next points to the new pool and a static int num_pools which helps us to record how many pools are created.
When this class is initialized, it will first assign the parameters and set every frame to Free. When the _info_frame_no is 0, it will automatically assign the first frame to store bitmap and mark this frame as HoS.

```
class ContFramePool {

private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */
    static int num_pools;
    static ContFramePool * head;
    ContFramePool * next;
    unsigned char * bitmap;
    unsigned int nFreeFrames;
    unsigned long base_frame_no;
    unsigned long nframes;
    unsigned long info_frame_no;
```

```
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _n_frames,
                             unsigned long _info_frame_no)
{
    // TODO: IMPLEMENTATION NEEEDED!
    base_frame_no = _base_frame_no;
    nframes = _n_frames;
    nFreeFrames = _n_frames;
    info_frame_no = _info_frame_no;
    if(info_frame_no == 0) {
        bitmap = (unsigned char *) (base_frame_no * FRAME_SIZE);
    } else {
        bitmap = (unsigned char *) (info_frame_no * FRAME_SIZE);
    }

    for(int fno = 0; fno < nframes; fno++) {
        set_state(fno, FrameState::Free);
    }

    if(_info_frame_no == 0) {
        set_state(0, FrameState::HoS);
        nFreeFrames--;
    }

    if(head == NULL){
        head = this;
        next = NULL;
    }else{
        ContFramePool * cur = head;
        for(int i = 1; i < num_pools; i++){
            cur = cur->next;
        }
        cur->next = this;
    }
    num_pools++;

    // assert(false);
}
```

**Cont_frame_pool: get_state:** This method returns the state of the frame, it will first push the target frame to the right two bits and use “and” operation with 0x11. 0x00 means the state is used, 0x01 means the state is free and 0x10 means the state is HoS.

```
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no){
    unsigned int bitmap_index = _frame_no / 4;
    unsigned char ans = bitmap[bitmap_index] >> ((_frame_no % 4) * 2);
    unsigned char mask = 0x3;

    switch((ans & mask)){
        case 0x0:
        return FrameState::Used;
        case 0x1:
        return FrameState::Free;
        case 0x2:
        return FrameState::HoS;
        default:
        return FrameState::Used;
    }
}
```

**Cont_frame_pool: set_state:** This method sets the state of the frame. First, it will change the state to 0x11 by using “or” operation with 0x11. Then it will do “xor” operation according to the input parameter _state.

```
void ContFramePool::set_state(unsigned long _frame_no, FrameState _state){
    unsigned int bitmap_index = _frame_no / 4;
    unsigned char mask1 = 0x3 << ((_frame_no % 4) * 2);
    unsigned char mask2;
    bitmap[bitmap_index] |= mask1;

    switch(_state) {
        case FrameState::Used:
        mask2 = 0x3 << ((_frame_no % 4) * 2);
        break;
        case FrameState::Free:
        mask2 = 0x2 << ((_frame_no % 4) * 2);
        break;
        case FrameState::HoS:
        mask2 = 0x1 << ((_frame_no % 4) * 2);
    }
    bitmap[bitmap_index] ^= mask2;

}
```

**Cont_frame_pool: release_frame_in_pool:** A private method that changes the state from HoS and used to free, it will change the state to free until the state is free originally. This method will only be called by release_frames.

```
void ContFramePool::release_frame_in_pool(unsigned long _first_frame_no){
    if(get_state(_first_frame_no-base_frame_no)!=FrameState::HoS){
        return;
    }
    set_state(_first_frame_no-base_frame_no,FrameState::Free);
    nFreeFrames++;
    int fno = _first_frame_no-base_frame_no+1;
    while(get_state(fno)==FrameState::Used){
        set_state(fno,FrameState::Free);
        nFreeFrames++;
    }
}
```

**Cont_frame_pool:get_frames:** A method that returns the base frame of the assigned frames or 0. 0 means there are not enough contiguous frames. This method will travers the bitmap and find out if there is enough contiguous Free state. If it finds enough frames, it will mark the first one as HoS and the rest as Used.

```
unsigned long ContFramePool::get_frames(unsigned int _n_frames)
{
    // TODO: IMPLEMENTATION NEEEDED!

    if(nFreeFrames < _n_frames){
        return 0;
    }

    unsigned int frame_no = 0;
    unsigned int count = 0;
    unsigned int is_exist = 0;


    while(frame_no <= nframes && is_exist == 0){
        if(get_state(frame_no) != FrameState::Free){
            frame_no++;
        } else{
            if(get_state(frame_no+count) == FrameState::Free && count < _n_frames){
                count++;
            } else if (count == _n_frames){
                is_exist = 1;
            } else{
                frame_no += count;
                frame_no ++;
                count = 0;
            }
        }
    }

    if(is_exist == 1){
        set_state(frame_no, FrameState::HoS);
        nFreeFrames--;
        for(int i = 1; i < _n_frames; i++){
            set_state(frame_no+i, FrameState::Used);
            nFreeFrames--;
        }
        return (frame_no + base_frame_no);
    } else{
        return 0;
    }
    // Console::puts("ContframePool::get_frames not implemented!\n");
    // assert(false);
}
```

**Cont_frame_pool: mark_inaccessable:** This method basically does the same thing as get_frames, but it will not check if there are enough free frames. It will just change the first one as HoS, the rest as Used.

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames)
{
    // TODO: IMPLEMENTATION NEEEDED!
    set_state(_base_frame_no - base_frame_no, FrameState::HoS);
    nFreeFrames--;
    for(int i = 1; i < _n_frames; i++){
        set_state(_base_frame_no - base_frame_no + i, FrameState::Used);
        nFreeFrames--;
    }
    // Console::puts("ContframePool::mark_inaccessible not implemented!\n");
    // assert(false);
}
```

**Cont_frame_pool: release_frames:** This static method will traverse the linked list to find which frame manager owns these frames. It will then call release_frame_in_pool to release frames and mark these frames as Free.

```
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    // TODO: IMPLEMENTATION NEEEDED!

    ContFramePool * release_pool = head;
    while(_first_frame_no > (release_pool->base_frame_no + release_pool->nframes) || _first_frame_no < release_pool->base_frame_no){
        release_pool = release_pool->next;
    }

    release_pool->release_frame_in_pool(_first_frame_no);

    // Console::puts("ContframePool::release_frames not implemented!\n");
    // assert(false);
}
```

**Cont_frame_pool: needed_info_frames:** This method returns how many frames are needed to store the information (1 frame can store information of 16KB frames).

```
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
{
    // TODO: IMPLEMENTATION NEEEDED!
    return _n_frames / 16384 + (_n_frames % 16384 > 0 ? 1 : 0);
    // Console::puts("ContframePool::need_info_frames not implemented!\n");
    // assert(false);
}
```

**Cont_frame_pool: get_info_of_linked_list:** This method prints out the number of frames of the pool. This method is used to test if the linked list works correctly or not.

```
void ContFramePool::get_info_of_linked_list()
{
    Console::puts("The frames of linked list \n");
    ContFramePool * cur = head;
    for(int i = 0; i < num_pools; i++){
        Console::puts("The number of frames in ");
        Console::puti(i+1);
        Console::puts(" pool: ");
        Console::puti(cur->nframes);
        Console::puts("\n");
        cur = cur -> next;
    }
}
```

## Testing

In addition to the original test functions, I also printed out the frame number of the pools in the linked list to make sure the pointers are correct. I think the coverage of testing is enough for an intelligent user to operate.