MP2: Frame Manager

Nikhil Nunna
UIN: 525008698
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The goal of this program was to create a Frame Manager. A Frame manager makes it possible for the OS to manage memory allocation for the kernel and user programs. It changes the memory view from individual addresses to contiguous chunks of memory, each chunk consisting of Frames that are all the same size. This makes it easy to allocate and de-allocate physical memory.

The Frame Manager consists of a few simple components. First, each frame is the same size and has a state that can be free, used, or head of sequence. A free Frame is open to be allocated, a used frame belongs to a process or the kernel, and a head of sequence frame is the beginning of a contiguous chunk of frames that belong to a single process. We represent the state of these frames in a bitmap which uses 2 bits per frame with the corresponding binary numbers 00, 01, and 10 to represent the states. The pointer to the Bit Map is stored in the stack but the bit map itself is stored in the start of a Frame pool in an information Frame. A Frame pool is just a grouping in the Frame Manager it is a way to divide the frames allocated to the kernel and the frames allocated to the user processes although you could create more divisions if you want. There are 2 Frame pools in our implementation one is for the kernel and the other is for the user processes. Using these tools were able to subdivide memory into a space for user processes and kernel processes, manage the state of memory, allocate and de-allocate memory by setting the frame state, and assign a contiguous set of frames to processes by setting a frame as head of sequence with a chain of used frames to follow.

Code Description

The code compiles and runs normally. All the grader needs to do is run 'make', './copykernel.sh' and then 'bochs -f bochsrc.bxrc'. As an alternative to this, I've created test.sh which can be run using './test.sh'. The test.sh shell script just runs make, copykernel and the bochs command for you.

test.sh: code block : This shell file compiles and runs the project

```
#!/bin/bash

make
./copykernel.sh
bochs -f bochsrc.bxrc
```

cont_frame_pool.c: get_state : In this function, the frame number is passed in and the frame state of the frame number is returned, this is done by translating the frame number to the corresponding binary representation in the bitmap and checking the state of the 2 corresponding bits. States are either 0 for free, 1 for used, and 2 for Head of sequence.

```
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no)
{
    // prep for bit shifting
    unsigned int bitmap_index = _frame_no /4;
    unsigned char mask = 0x3 << ((_frame_no % 4) * 2);

    //Get Frame state in binary to switch case through
    unsigned char frame_state = (bitmap[bitmap_index] & mask) >> ((_frame_no % 4) * 2);
    if (frame_state == 0x0)
    {
        return FrameState::Free;
    }
    else if (frame_state == 0x1)
    {
        return FrameState::Used;
    }
    else if (frame_state == 0x2)
    {
        return FrameState::HoS;
    }
}
```

cont_frame_pool.c: set_state : In this function, the frame number is passed in along with the state it needs to be set to, and the frame state is updated, this is done by translating the frame number to the corresponding binary representation in the bitmap and updating the state of the 2 corresponding bits. States are either 0 for free, 1 for used, and 2 for Head of sequence.

```
void ContFramePool::set_state(unsigned long _frame_no, FrameState _state)
{
    // prep for bit shifting
    unsigned int bitmap_index = _frame_no /4;
    unsigned char shift = ((_frame_no % 4) * 2);
    unsigned char mask = 0;

    // Switch through on requested state to set state
    switch(_state) {
        case FrameState::Free:
            mask = 0x3 << shift;
```

```
            bitmap[bitmap_index] &= ~mask;
            break;
        case FrameState::Used:
            mask = 0x1 << shift;
            bitmap[bitmap_index] |= mask;
            break;
        case FrameState::HoS:
            mask = 0x2 << shift;
            bitmap[bitmap_index] |= mask;
            break;

    }

}
```

cont_frame_pool.c: Constructor : In this function, the base frame number, the number of total frames, and the info frame number are passed in the constructor then adds the frame pool to the bitmap. Set the private variables, it then creates the bitmap and sets the various states of the frames.

```
ContFramePool(unsigned long _base_frame_no,
                              unsigned long _n_frames,
                              unsigned long _info_frame_no)
{
    // Add Frame pool to Doubly Linked List
    if (Head == nullptr)
    {
        Head = this;
    }
    if (Tail == nullptr)
    {
        Tail = this;
    }
    if (Tail != this)
    {
        Tail -> next = this;
        prev = Tail;
        Tail = this;
    }
    // Set Frame pool private vars
    base_frame_no = _base_frame_no;
    nframes = _n_frames;
    nFreeFrames = _n_frames;
    info_frame_no = _info_frame_no;

    // If _info_frame_no is zero then we keep management info in the first
    //frame, else we use the provided frame to keep management info
    // allocate bitmap
    if(info_frame_no == 0) {
        bitmap = (unsigned char *) (base_frame_no * FRAME_SIZE);
    } else {
        bitmap = (unsigned char *) (info_frame_no * FRAME_SIZE);
    }

    // Everything ok. Proceed to mark all frame as free.
    for(int fno = base_frame_no; fno < base_frame_no + nframes; fno++) {
        set_state(fno, FrameState::Free);
    }
```

```
    // Mark the first frame as being used if it is being used
    set_state(_info_frame_no, FrameState::HoS);
    if (info_frame_no >= base_frame_no)
    {
        nFreeFrames--;
    }

    // Mark subsequent frames as being used for information frames
    unsigned long info_frames = needed_info_frames(_n_frames);
    for (unsigned long i = _info_frame_no + 1; i < _info_frame_no + info_frames; i++)
    {
        set_state(i, FrameState::Used);
        if (i >= base_frame_no)
        {
            nFreeFrames--;
        }
    }


    Console::puts("Frame Pool initialized\n");
}
```

cont_frame_pool.c: get_frames : In this function, the number of requested frames is passed in. The function then searches for a contiguous block of memory large enough to fulfill the request. If the request cannot be fulfilled then it returns -1, which will generate an error. If it can fulfill the request then it returns the first frame in the sequence and sets the frame state of all the frames in the contiguous block.

```
unsigned long ContFramePool::get_frames(unsigned int _n_frames)
{
    // Implement First Fit
    // Check there are enough free frames
    if (nFreeFrames < _n_frames)
    {
        return 0;
    }
    // set vars to find start frame
    unsigned long counter = 0;
    unsigned long start_pos = base_frame_no;
    bool found = false;
    // Loop through frame pool to find a contigious block of _n_frames
    for (unsigned long i = base_frame_no; i < base_frame_no + nframes; i++)
    {
        if(get_state(i) == FrameState::Free)
        {
            counter++;
            // if found break and go to allocation
            if (counter == _n_frames)
            {
                found = true;
                break;
            }

        }
        // if not found then reset counter and move start to next frame
        else
```

```
            {
                counter = 0;
                start_pos = i + 1;
            }

        }
        // allocate the frames as HOS and used
        if(found)
        {
            // Allocated Hos
            set_state(start_pos, FrameState::HoS);
            nFreeFrames--;
            for (unsigned long i = start_pos + 1; i < start_pos + _n_frames; i++)
            {
                // Alocated subsequent frames
                set_state(i, FrameState::Used);
                nFreeFrames--;
            }
            return start_pos;
        }

        return -1;
}
```

cont_frame_pool.c: mark_inaccessible : In this function, the base frame and the size of frames for a block memory that needs to be made inaccessible are passed in. The function updates the state of those frames to be hos and used so that their made inaccessible.

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames)
{
    // Mark first frame as Hos
    set_state(_base_frame_no, FrameState::HoS);
    nFreeFrames--;
    for(int fno = _base_frame_no + 1; fno < _base_frame_no + _n_frames; fno++)
    {
        // Mark subsequent frames as used
        set_state(fno, FrameState::Used);
        nFreeFrames--;
    }
}
```

cont_frame_pool.c: mark_inaccessible : In this function, a frame number that is supposed to be head of sequence (Hos) is passed in, and the function then searches the doubly linked list of frame pools to check which frame pool the frame belongs to. Once the frame pool it belongs to is found it then removes the chunk of contiguous memory associated with it until it finds the next chunk of memory by running into a frame that is hos or by running into unallocated memory by running into a free frame. In the event the frame passed in is not Hos it will just exit the function.

```
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    if (Tail != nullptr)
    {
        ContFramePool * pool = Tail;
        bool not_found = true;
        // Find the Frame pool this frame belongs too
```

```
        while (not_found)
        {
            // Find largest Base Frame less than _first_Frame and
            ensure first frame is less than the last frame in pool
            if ((pool->base_frame_no <= _first_frame_no) &&
            (pool->base_frame_no + pool->nframes > _first_frame_no))
            {

                not_found = false;
            }
            else
            {
                // iterate to the previous frame pool
                if (pool != Head)
                {
                    pool = pool->prev;
                }
                else
                {
                    return;
                }

            }

        }
        // Ensure frame is start of the contigous block
        if(pool->get_state(_first_frame_no) == FrameState::HoS)
        {
            // Set vars to release frames
            unsigned long curr_frame_no =_first_frame_no;
            // Release Hos
            pool->set_state(curr_frame_no, FrameState::Free);
            pool->nFreeFrames++;
            curr_frame_no++;
            bool alloc = true;
            while (alloc)
            {
                // Release Frames till next block our empty frames
                if((pool->get_state(curr_frame_no) != FrameState::HoS) &&
                (pool->get_state(curr_frame_no) != FrameState::Free))
                {
                    pool->set_state(curr_frame_no,FrameState::Free);
                    curr_frame_no++;
                    pool->nFreeFrames++;
                }
                // end loop
                else
                {
                    alloc = false;
                }
            }
        }

    }
}
```

cont_frame_pool.c: needed_info_frames : In this function, the total number of frames in the frame pool are passed in which are then divided by the total frame size. This number is again divided by 4 to find the total number of frames needed to address all the frames in the frame pool. The modulus of the first calculation is also added in the event that a partial frame might be needed.

```
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
{
    // return frames by FRAME_SIZE divide by 4 plus one if remainder
    // Divide by 4 because 2 bits per frame to address it
    return (((_n_frames / FRAME_SIZE) / 4) + (_n_frames % FRAME_SIZE > 0 ? 1 : 0));
}
```

cont_frame_pool.c: free_frames : This function that I added for testing purposes returns the number of free frames in a frame pool.

```
unsigned long ContFramePool::free_frames()
{
    // Return the number of Free Frames
    return this->nFreeFrames;
}
```

kernel.c: test_max_alloc : This function that I added for testing purposes allocates every single frame in the process frame pool.

```
void test_max_alloc(ContFramePool * _pool)
{
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 6912 Frames "); Console::puts("\n");
    _pool->get_frames(4096);
    _pool->get_frames(2816);
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 0 Frames "); Console::puts("\n");
}
```

kernel.c: release_all_frames : This function that I added for testing purposes releases all the frames allocated by the test_max_alloc test case.

```
void release_all_frames(ContFramePool * _pool)
{
    _pool->release_frames(1024);
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 2816 Frames "); Console::puts("\n");
    _pool->release_frames(4096);
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 6192 Frames "); Console::puts("\n");
}
```

kernel.c: alloc_and_release_last_frame : This function tests to ensure that the last frame in the process frame pool can be allocated as Hos and deallocated.

```
void alloc_and_release_last_frame(ContFramePool * _pool)
{
    _pool->get_frames(4095);
    _pool->get_frames(2816);
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 1 Frame "); Console::puts("\n");
    _pool->get_frames(1);
```

```
    _pool->release_frames(1024);
    _pool->release_frames(4096);
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 6191 Frames "); Console::puts("\n");
    _pool->release_frames(8191);
    Console::puts("Free Frames: "); Console::puti(_pool->free_frames());
    Console::puts(" Expecting 6192 Frames "); Console::puts("\n");
}
```

Testing

I have added 4 functions strictly for testing, free_frames, test_max_alloc, release_all_frames, and alloc_and_release_last_frame. The first function allowed me to verify that the number of free frames was being updated correctly. The second function test_max_alloc allowed me to check that I was able to correctly allocate all the frames and ensure that mark_inaccessible function worked. It helped me find a bug where I was subtracting free frames when I was allocating my process pool info frame even though that frame is not part of the process pool. The next function release_all_frames then releases all the frames allocated by the previous function and it helped me catch a bug in my release_frames function where I was using the wrong comparator. I was using < in an if statement to check whether the first allocated frame was greater than the base frame when I should've been using <=. The last function I added for testing was alloc_and_release_last_frame, this covered the edge case and ensured all my comparators in the release_frames function worked as intended and didn't exclude any edge cases.

Overall I think I have very good coverage, I have coverage of all my comparators in all functions, all possible frame allocations, lastly I have covered the only edge case I can think of.