CSCE - 611 (Operating Systems)
MACHINE PROBLEM -2

Nitin Chakravarthy Gummidela
(427006656)

In this machine problem a Frame Pool manager was supposed to be created which would allocate a contiguous block of frames, release those frames and also mark specific parts of the memory as inaccessible. A total memory of 32 MB was assumed. The first 4MB was used as the kernel memory where as the next 28 MB was used as the process memory space. Further the 1st 1MB contained global variables and the kernel code and stack space is present in the next 1MB. Therefore the available memory space for the kernel started from 2MB and ended at 4MB. Also there is a part of memory between 15MB and 16MB in the process space was supposed to be marked inaccessible.

The size of each frame was taken as 4KB. So the number of frames in 32MB is 8K frames. Since we are implementing a frame pool manager which allocates contiguous blocks of memory and also releases it, we need to keep track of the Head of the frame pools. For this we will use one extra bit to represent if the frame is the head of the pool or not. Therefore, for each frame we would be using 2 bits. We will use the combinations of these bits to represent if the frame is free, allocated and head of pool, allocated but not head and inaccessible. So using 2 bits per frame we would need 16K bits to represent the whole 32MB memory. This is less than 4KB therefore the bitmap would fit inside one frame.

We will have two FramePoolManager instances one to handle kernel frame pool memory and the other to handle process frame pool memory. This instantiation is done in the kernel.C file. Once the process FramePoolManager is instantiated, it's *mark_inaccessible()* method is used to mark the 15-16MB memory block as inaccessible.

The following section below gives a brief understanding of how each method in the Contiguous Frame Pool Manager we implemented.

cont_frame_pool.H and cont_frame_pool.C are the header and implementation files for the manager.

cont_frame_pool.H file has public and private variable and method declarations (static and non static). **cont_frame_pool.C** has the implementations of the declared methods.

First, let's look into the variable declarations and the methods that were declared in the **cont_frame_pool.H** file to be used in the implementation.

```
private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */
    unsigned char * bitmap;        // We implement the simple frame pool with a bitmap
    unsigned int    nFreeFrames;   //
    unsigned long   base_frame_no; // Where does the frame pool start in phys mem?
    unsigned long   nframes;       // Size of the frame pool
    unsigned long   info_frame_no; // Where do we store the management information?
    unsigned long   n_info_frames; // This is where we store the number of frames needed for
the managememnt information



    static ContFramePool* cont_frame_pool_head;
    static ContFramePool* cont_frame_pool_list;
    ContFramePool* frame_pool_next; // pointer to the next frame pool

    void mark_inaccessible(unsigned long _frame_no);
```

These are the private variables declared. The **bitmap** defined as a **char*** will be used to store the information of the bits corresponding to the frames.
nFreeFrames represents the number of free frames.
base_frame_no is the location where the frame pool starts in the physical memory.
nFrames is the total number of frames in the memory corresponding to the frame pool.
info_frame_no is the location where the bitmap is stored. (Start of the bitmap)
n_info_frames is the number of frames required to store the bitmap.

Then we have 2 static variables of the type ContFramePool*. cont_frame_pool_list is a list of the ContFramePool objects and cont_frame_pool_head is the head of that list. Frame_pool_next is the pointer to the next ContFramePool object. We add the ContFramePool objects to the list in the constructor. These are useful in identifying the FramePools when the frames are to be released.
Then a private mark_inaccessible() method was declared to use as a helper function in the static mark_inacessible() method.

Now, let's look at the implementation of the ContFramePool which is present in
cont_frame_pool.C
Starting from the top, the code has the following method implementations.

1. ContFramePool - (constructor)
2. get_frames
3. mark_inaccesible - (static)
4. mark_inaccessible - (private)
5. Release_frames
6. needed_info_frames

Before looking at the method implementation. We need to consider a few conventions to represent various states of the frame, so that we can handle the states and modify them using a suitable bitwise operation. These are the conventions that are used in this code.

Free - 00
Allocated and Head - 10
Allocated not Head - 11
Not Accessible - 01

Now let's look into the methods.

ContFramePool - Constructor:
In this function we initialize various private variables and define the initial states of the frames. The constructor performs the following:

1. Checks if the number of frames is less than the size of a frame in bits.
2. Initializes the private variables declared in the header file using the arguments passed.
3. Sets the variable 'bitmap' to the starting location of the bitmap, using base_frame_no or info_frame_no.
4. Marks all the bits in the bitmap to represent that the frames are free.
5. If the _n_info_frames >0 marks _n_info_frames starting from _info_frame_no as inaccessible. Otherwise marks the first frame as inaccessible. (This frame is being used for the bitmap in the kernel pool.)
6. Inserts the instance in the cont_frame_pool_list

get_frames(unsigned int _n_frames):
This method is used to allocate a contiguous memory pool. It marks the pool of array as allocated and then returns the first frame of the contiguous pool.
The method has two parts. In the first part the bitmap is searched for a contiguous block of memory of the given size (_n_frames). If the block of memory is found the index of the start is stored. Second the block of memory is marked as allocated and the pointer to the start is returned.

In the first part, 2 nested loops were implemented. The outer loop operates on the bitmap where as the innerloop operates inside the bitmap to handle the 2 bits that each frame is represented with. So the outer loop runs between 0 and n_frames/4, where as the inner loop runs between 0 and 4. Inside the loop, we first try to find a bit combination that pertains to a free frame. When we find one we store it's location and then we start looking for the empty frames after that. If we find an empty frame, we continue to look until we find the number of needed frames. If we find an occupied frame before finding enough frames, we reset the local variables to their initial states and restart searching from the next frame. If we find enough frames we come out of the loops by using break; To check for free frames we use a mask(**0xC0 - 11000000)** which on performing logical AND with the corresponding bitmap would lead to **0** if it is empty. Then we right the mask by 2 bits and continue the process of searching for new free frames.

After both the loops execute, we check if all the frames were found. If not found we **return 0** according to the provided specifications or else we continue.

Now, In the second part, We mark the selected frames as allocated. For this we keep a counter which equals the number of frames to be marked. We keep updating the counter once a frame is marked in the bitmap. We stop when this value reaches zero and then we return the head of the sequence. As the first step in this art we mark the head of the sequence as **10 (convention used.).** We use the index in the bitmap **(byte_index)** and the index inside the char **(bit_index)** that we save in part 1. We use a mask (**0x80 - 10000000**) and right shit it 2*bit_index times to reach the position of the bits that correspond to the head of the sequence.
Then we mark the other bits till the counter runs out as allocated but not head(**11**). For this we use a mask(**0xC0 - 11000000**) and shift it by 2 bits every time a frame is marked.
Finally we subtract the number of frames allocated from the total number of free Frames.

mark_inaccessible(unsigned long _base_frame_no, unsigned long _n_frames):
This method is used to mark a block of memory as inaccessible. _base_frame_no indicated the start of the block and _n_frames indicate the number of frames to be marked inaccessible.
We use a helper function that marks one frame as inaccessible and run a loop that runs from **_base_frame_no** to **_base_frame_no + _n_frames.** The helper function is described below.

mark_inaccessible(unsigned long _frame_no):
This function marks the bits corresponding to a single frame (_frame_no) as inaccessible.
To mark this as inaccessible we filter the specific bits using a filter_mask **(0xC0 - 11000000)**. Performing a logical **OR** between the bitmap and the mask isolates the bits. Then **XOR** is performed on the result and another mask **(0x80 - 10000000).** Corresponding shits are performed on the mask before operating based on the frame_no. Then the frame is removed from the number of free frames.

release_frames(unsigned long _first_frame_no):
This function is used to release a pool of frames that are already allocated. SInce there can be multiple frame pools defined in the system, it needs to first identify the correct frame pool and release the frames in it.
So this method contains 2 parts too.
In the first part, cont_frame_pool_list is used to identify the correct frame pool by comparing the _first_frame_no to the base_frame_no of the pool in the list. If no pool is matched, an error is logged and we return.

In the second part the release the frames starting from _first_pool_no;
First we release the head use of two masks **(0X40 - 01000000) and (0xC0 - 11000000)**. First we check if the provided frame number is the head of a sequence of frames. If we continue, else we log an error. Then we perform necessary bitwise operations to mark the corresponding bits as **00.**
Then we use the mask(**0xC0 -11000000**) and change the bits corresponding to the frames to be released to make them free. This is done till the next head is found. We return if we find the next head.

needed_info_frames(unsigned long _n_frames):
This function returns the number of frames needed to manage a framepool of size _n_frames

Execution and Testing:
The methods are tested for kernel pool and the process pool in the Kernel.C file. Two ContFramePool objects one for the kernel and one for the process pool were created. The process pool has piece of memory from 15MB-16MB marked as inaccessible.

Each FramePoolManager was used to allocate and then release 32 blocks of memory.
The function test_memory in the kernel.C is a recursive function that calls the get_Frames method to allocate memory and then release_frames function to release the memory once all the allocation has taken place.