

# CSCE611: Operating System

## MP2: Frame Manager

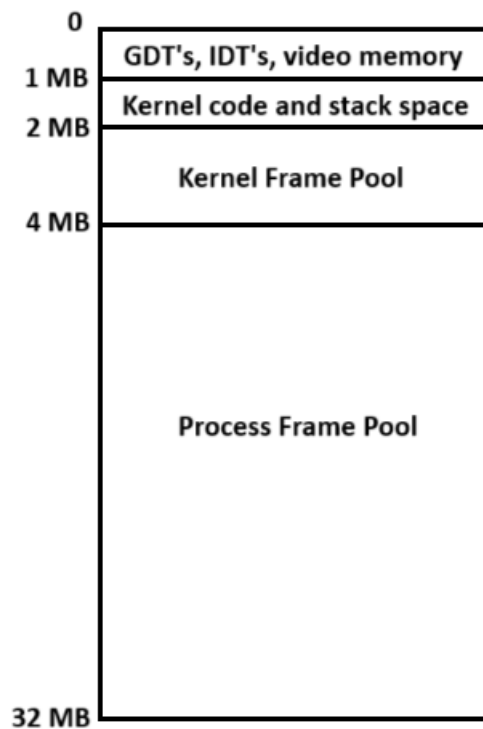
Name: Rahul Ravi Kadam  
UIN: 834005554

### System Design:

The goal of the machine problem is to implement a frame manager, which manages the allocation of frames (physical pages). The frame manager is responsible for the allocation and release of frames, i.e., it needs to keep track of which pages are being used and which ones are free. The memory layout of the machine is given below:

- Total memory in the machine = 32 MB
- Memory reserved for Kernel = 4 MB
- Process memory pool = 28 MB
- Frame size = 4 KB
- Inaccessible memory region = 15 MB – 16 MB

### Memory Layout Diagram:



Each frame pool separately manages a collection of frames that can be allocated and released. When the kernel needs frames, it retrieves them from the kernel frame pool, which is located between 2 MB and 4 MB. When a process needs frames, it retrieves them from the process frame pool, which is located beyond 4 MB.

To implement a frame pool, I used a bitmap describing the state of each frame in the frame pool. To efficiently track the state of each frame, I used 2 bits to represent one frame. The bit representation is given below:

Bit Representation	Frame State
00	Free
01	Used / Allocated
11	Head of Sequence (Allocated)

Kernel Pool Frames Location: 512 – 1023 → 512 frames in kernel pool

Process Pool Frames Location: 1024 - 8191 → 7168 frames in process pool

If we take 2 bits to store the state of each frame in the bitmap, then:

Kernel pool bitmap size will be 1024 bits or 128 bytes, which will fit in 1 frame (since frame size = 4 KB)

Process pool bitmap size will be 7168 bits or 896 bytes, which will also fit in 1 frame (since frame size = 4KB)

To manage multiple frame pools (kernel and process memory) which have their individual bitmaps to track the state of frames, I implemented a static linked list of frame pools. In this manner, we can traverse the linked list to access each frame pool and manage the frame pool bitmaps, get frames and release frames. The linked list approach was taken since the release frames functionality was static and is common across all the frame pools.

#### List of files modified:

1. cont\_frame\_pool.H
2. cont\_frame\_pool.C

#### Code Description

-> **cont\_frame\_pool.H : class ContFramePool** : In class ContFramePool, data structures for the frame pools were declared:

- a) unsigned char \* bitmap - Bitmap for Cont Frame Pool
- b) unsigned int nFreeFrames - Number of free frames
- c) unsigned long base\_frame\_no - Frame number at start of physical memory region
- d) unsigned long nframes - Number of frames in frame pool
- e) unsigned long info\_frame\_no - Frame number at start of management info in physical memory

Frame Pool Linked List pointers are also declared the in the class:

- a) static ContFramePool \* head - Frame Pool Linked List head pointer
- b) ContFramePool \* next - Frame Pool Linked List next pointer

-> **cont\_frame\_pool.H : release\_frames\_in\_pool()** : Since release\_frames() is a static function, I defined the prototype for a non-static internal version of release\_frames() which can be called to release the frames of the correct frame pool.

-> **cont\_frame\_pool.C : constructor ContFramePool()**: This method is the constructor for class ContFramePool. It initializes all the data structures needed for the management of the frame pool. It initializes all the class data members to the values passed in the arguments to the constructor. All the bits in the frame pool bitmap are initialized to 'Free' state. A linked list is created (if not created earlier) or a new frame pool is added to the linked list.

-> **cont\_frame\_pool.C : get\_state()** : This method is used to obtain the state of a frame by using the `_frame_no` argument. Assuming the bitmap was a set of rows and columns, we can obtain the row index and column index of the 2 bits that are used to represent the desired frame. By performing bitwise operations (right shift operation), we can extract the desired 2 bits from the bitmap and check the value to find out the state of the frame allocation. Example of bitmap row and column index calculation: To calculate the location of the 2 bits used to describe the frame state for frame number 10: Bitmap row index =  $(11 / 4) = 2$  (Integer Division)  $\rightarrow$  3rd row Bitmap column index =  $(11 \% 4) * 2 = 6 \rightarrow$  7th bit Hence, the desired frame state bits are the 7th and 8th bits of the 3rd bitmap element ( `bitmap[2]` ).

-> **cont\_frame\_pool.C : set\_state()** : This method is used to set the state of a particular frame. The state is set to either Free, Used or Head of Sequence (HoS) based on the `_state` argument passed to the method. Once again, we obtain the row index and column index of the bits for `_frame_no` and perform bitwise operations (AND, NOT, XOR operations) to set the desired state.

-> **cont\_frame\_pool.C : get\_frames()** : This method is used to allocate a number of contiguous frames from the frame pool. A check is performed to verify that enough free frames are available to be allocated for the `get_frames` request. Next, using a loop, we check if there exists a set of '`_n_frames`' contiguous frames available in the frame pool. If this set of frames are available, then using a loop, we set the state of the first frame to Head of Sequence (HoS) and the remaining '`_n_frames`' - 1 ) frame states to Used/Allocated. If successful, the frame number of the first frame is returned. If a failure occurs, the value 0 is returned. The search algorithm is optimized to run in  $O(n)$  time, i.e., the length of the frame pools.

-> **cont\_frame\_pool.C : mark\_inaccessible()** : This method is used to mark a contiguous sequence of physical frames as inaccessible. First, a check is performed to verify that the frames to be marked as inaccessible are within the range of the frame pool. To mark frames as inaccessible, we use a loop to iterate through the frame numbers and use the '`set_state`' method to set the state of the frame as Head of Sequence for the first frame and the remaining frame states as Used in the frame pool bitmap.

-> **cont\_frame\_pool.C : release\_frames()** : This method is used to release a previously allocated contiguous sequence of frames back to its frame pool. The frame sequence is identified by the number of the first frame. Using a loop, we iterate through the linked list and check whether the frame number exists within the range of frame numbers of the frame pool. If the frame number exists in the frame pool, we have identified the frame pool the frame belongs to. Next, we call the '`release_frames_in_pool`' method to release all the frames of the identified frame pool.

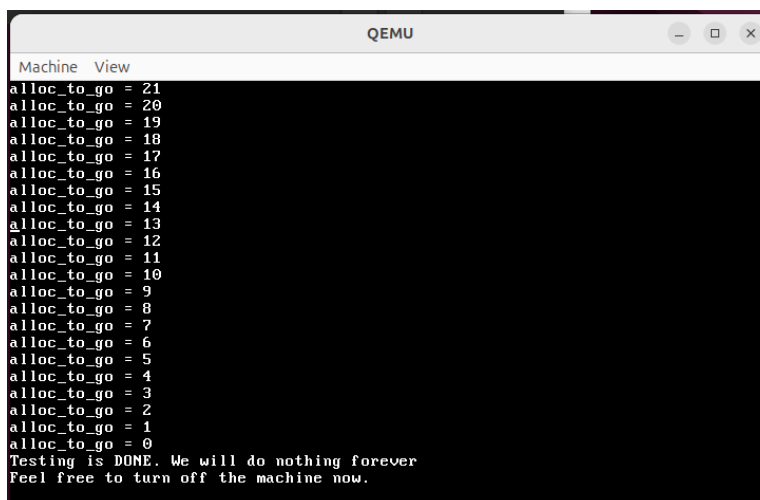
-> **cont\_frame\_pool.C : release\_frames\_in\_pool()** : This method is used to release all the frames of a particular frame pool. We first verify whether the bitmap state of the first frame is Head of Sequence. Next, using a loop, we iterate through the frame numbers and set the state of the frame to 'Free'. Also, the variable nFreeFrames is incremented.

-> **cont\_frame\_pool.C : needed\_info\_frames()** : This method returns the number of frames needed to manage a frame pool of size \_n\_frames. For frame size = 4096 bytes and a bitmap with two bits per frame, we can calculate the number of info frames needed to be:  $(\_n\_frames * 2) / 32k + ((\_n\_frames * 2) \% 32k > 0 ? 1 : 0)$

For example, for the process pool, we can perform the following calculation: Number of frames in 28 MB process memory pool = 7168 frames Considering 2 bits per frame, then Number of info frames needed =  $((7168 * 2) / 32000) + ((7168 * 2) \% 32000 > 0 ? 1 : 0) = 1$  Frame

### Testcase Outputs:

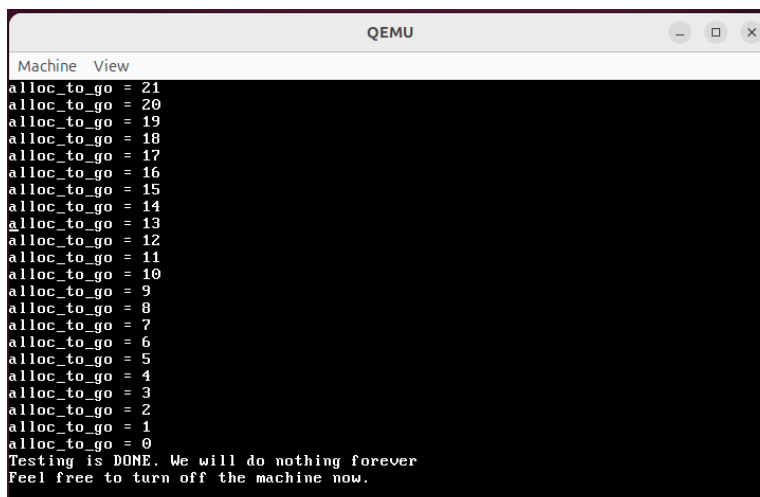
Testcase 1: test\_memory(&kernel\_mem\_pool, 32);

A screenshot of a QEMU terminal window. The window title is "QEMU". It has a menu bar with "Machine" and "View". The terminal output shows a list of "alloc\_to\_go" values from 21 down to 0, each on a new line. At the bottom, it says "Testing is DONE. We will do nothing forever" and "Feel free to turn off the machine now.".

```
Machine View
alloc_to_go = 21
alloc_to_go = 20
alloc_to_go = 19
alloc_to_go = 18
alloc_to_go = 17
alloc_to_go = 16
alloc_to_go = 15
alloc_to_go = 14
alloc_to_go = 13
alloc_to_go = 12
alloc_to_go = 11
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
Testing is DONE. We will do nothing forever
Feel free to turn off the machine now.
```

Testcase PASSED

Testcase 2: test\_memory(&process\_mem\_pool, 32); (Added in kernel.C)

A screenshot of a QEMU terminal window, identical to the one for Testcase 1. It shows the same sequence of "alloc\_to\_go" values from 21 down to 0, followed by the completion message.

```
Machine View
alloc_to_go = 21
alloc_to_go = 20
alloc_to_go = 19
alloc_to_go = 18
alloc_to_go = 17
alloc_to_go = 16
alloc_to_go = 15
alloc_to_go = 14
alloc_to_go = 13
alloc_to_go = 12
alloc_to_go = 11
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
Testing is DONE. We will do nothing forever
Feel free to turn off the machine now.
```

Testcase PASSED