MP2: Frame Manager

By: Mehul Varma

Part 1 Introduction

In this assignment, we were tasked to implement a simple frame manager that manages the allocation of physical frames in the memory of the kernel we will be developing.

The frame manager takes responsibility for the allocation, release, and management of frames in general.

Our frame manager is implemented as a collection of frame pools (a collection of frames themselves). Each of these frame pools would manage a segment of the memory managed by the frame manager as a whole. Each frame pool is responsible for the various actions that can be performed with the frames in their control.

Given this structure, in this assignment, we will be implementing frame pool as a C++ class where each of the member functions would represent the various responsibilities of the frame pool.

***** In this assignment only the *cont_frame_pool.H* and *cont_frame_pool.C* files have been modified from the given *MP2_sources* starter code. *kernel.C* was modified to perform testing on the class implemented.

Part 2 Data Design

Other than some easy to use hash define values used in internal functions we use the following data variables and structures in the designed class.

Global data variables and structures-

In the *cont_frame_pool* class, only two variables have been used.

- `static ContFramePool* head = NULL;`
- `static ContFramePool* tail = NULL;`

They are the head and tail variables of pointer type of object *cont_frame_pool.* These variables store the address of the first created frame pool and the last created frame pool. Using these variables we implement the linked list implicitly.

Instance data variables and structures-

The following are the instance variables used in the *cont_frame_pool* class -

- `unsigned char * bitmap_work;`
- `unsigned char * bitmap_hos;`
- `unsigned long   base_frame_no;`
- `unsigned long   nframes;`
- `unsigned long   info_frame_no;`
- `unsigned long   _n_info_frames;`

The first two variables point to the two bitmaps of the instance being created. This is explained more in part 3.

The other arguments are the frame where the pool created starts from, the number of frames the pool handles, the frame where the management info is stored for the current pool, and the number of info frames used by the pool respectively.

Part 3 Architectural and component-level design

ContFramePool constructor-

The constructor first initializes the arguments to the instance variables.

In our implementation of the class, we develop two bitmaps. The first bitmap consists of the busy/free status of the frame and the second bitmap consists of the Head of Sequence status of the frame. The bitmaps are designed to be stored sequentially rather than nesting the two bits together in one single bitmap. Therefore the length of each bitmap is the length of frames managed by the pool.

Before creating the address for our bitmaps, we check if the constructor is called by the kernel pool or a process pool. Depending on that we know if the info frames are supposed to be stored internally or externally.

After we are done getting the address of our bitmaps, we mark all the frames free and make them all not head of sequence by using the helper functions *set_work_status* and *set_hos_status.* Once we are done with that we would want to mark the first bit to be busy and head of sequence if the info frames are stored internally as they occupy the first frame in the kernel pool.

Finally, we implement the implicit linked list we use in the implementation of this class. We only support the creation of the pool, therefore every time the constructor is called we simply insert the new instance object in the linked list.

Get Frames-

Our get frame implementation consists of a single algorithm whose job is to find the first sequence with the n number of frames free in the pool. If the function is unable to find such a sequence it just returns 0.

The algorithm we developed has a runtime of *O(n)*. As it only needs to iterate the bitmap once to find such a sequence even though this algorithm uses two nested loops. The outer loop searches for a singular free frame and the inside loop checks if the following n-1 frames are free or not. If they aren't free, the outer loop starts with the frame after the sequence which was just checked.

If the algorithm is successful in finding such a sequence of frames, it returns the actual frame number that can be used. Since this function is a non-static function, it is called upon a singular pool to find if n frames can be stored in it.

Release Frames-

This function is a static function, so it would need to iterate through the linked list of frame pools to find out which frame the requested frame is to be released from.

Therefore, the function first iterates through the linked list and uses the range of frames handled by each pool to figure out which pool the frame to be released belongs to.

If it doesn't find such a pool, it just returns. But if it does, checks to make sure that the requested frame to be released is a head of the sequence and is busy. Once these preliminary checks are done, the function marks all contiguous busy and non-head of sequence frames to be free.

Mark Inaccessible

This function is similar to get frames except it knows the number of frames it needs to mark to be busy.

Before changing their status to be busy, preliminary checks the inputs is done to check their range and their status to be free.

After the checks, the n number of frames are set to be busy.

Needed Info Frames

Since in our implementation, we are using two bits for every frame managed by a single frame pool, we just divide that number by the number of bits in each page to simply give us the number of info frames needed. The ceiling of this calculation is taken using bit logic. The following is the formula used in this function -

- *# of info frames required =* `(nFrames*2)/(FRAME_SIZE*8)`

Helper functions

The helper functions in this class are just masked bit manipulation functions used. The following is a short description of the functions used -

- **`Set_work_status:`** It sets the work bit in the work bitmap for a given frame
- **`Set_hos_status:`** It sets the Head of sequence bit in the work bitmap for a given frame
- **`Set_bit:`** It uses bit manipulation to set a bit at given index as 1 in an unsigned char
- **`Unset_bit:`** It uses bit manipulation to set a bit at given index as 0 in an unsigned char
- **`Get_bit:`** It uses bit manipulation to return a bit at given index in an unsigned char
- **`Print_array:`** It prints the unsigned char array bit by bit for helping us in the testing process.

Part 4 Testing

For this assignment we altered the *kernel.C* file to perform the following tests as the singular call to *test_memory()* function fell short at a few edge cases -

- Larger number of allocs in *test_memory()* with kernel pool
- Creation of process pool
- Get frames from process pool
- Release frames from process pool
- Larger number of allocs in *test_memory()* with kernel pool
- Testing *make_inaccessible()* on process pool and kernel pool and then trying get frames after that call
- Using two process pool, and testing its get frames and release frames
- Changing *test_memory()* allocate more frames at a time and repeat prior tests

- Stress test kernel and process frame pools with 50+ and 100+ allocs respectively and then checking if all frames released correctly