Machine Problem 2: Frame Manager

Introduction

The objective of this machine problem is to get you started on a demand-paging based virtual memory system for our kernel. You begin by implementing a **frame manager**, which manages the allocation of frames (physical pages). The frame manager is responsible for the allocation and the release of frames, i.e, it needs to keep track of which pages are being used and which ones are free. Different parts of the memory will be used by different parts of the operating system, and so the frame manager will manage a collection of **frame pools**[1], which support get and release operations for frames.

Assumptions about our Kernel

The memory layout in our kernel looks as follows:

- The total amount of memory in the machine is 32MB.
- The memory layout will be so that the first 4MB are reserved for the kernel (code and kernel data) and are shared by all processes:
  - The first 1MB contains all global data, memory that is mapped to devices, and other stuff.
  - The actual kernel code starts at address 0x100000, i.e. at 1MB.
- Memory within the first 4MB will be **direct-mapped** to physical memory. By "direct-mapped" we mean that a given logical address is mapped to the physical address with the same value. For example, logical address say 0x01000 will be mapped to physical address 0x01000. The address space beyond 4MB will be **freely mapped**; that is, every page in this address range will be mapped to whatever physical frame was available when the page was allocated.[2]

The Project: Frame Management

You will build a **frame manager**, which allows you to get and release frames to be used by the kernel or by user processes. The frames are organized in so-called **pools**, which are managed separately. The frame manager allows for the creation of **frame pool objects**. Frame pools are each assigned a region of the physical memory. Callers request one or more contiguous frames by calling the appropriate frame pool's function `get_frames`. (For C++ geeks who know about "placement new" this should sound familiar.)

In addition, the frame manager supports the release of frames, but does so in a strange fashion. We return to this below.

[1] Authors who prefer dry land refer to these as *Arenas*.

[2] All this mapping business is not important yet, and you will understand this better by the time you get to the subsequent MPs. For now, all you need to know is that this memory will be used by the programs running in the system, and your **frame manager** will be the component keeping track of which physical pages are used and which ones are free. Later, whoever does the mapping will have to rely on the frame manager to request and release frames when needed.

The kernel, in file `kernel.C`, will set up two pools, the **kernel frame pool** and the **process frame pool**: Whenever the kernel needs frames, it gets them from the **kernel frame pool**, which is located between 2MB and 4MB. Non-kernel data is then allocated from the **process frame pool**, which manages memory above 4MB.

The reason for this separation of kernel from process frame pool will become clearer in the later MPs. For now, just focus on implementing the frame manager.

A Note about the First 4MB

Don't get confused by the fact that the kernel frame pool does not extend across the entire initial 4MB, and ranges from 2MB to 4MB only. The address range from zero to 1MB contains GDT's, IDT's, video memory, and other stuff. The address range from 1MB to 2MB contains the kernel code and the stack space. So we leave the address range from zero to 2MB alone.

The Frame Pool

Available physical memory frames are managed in objects of class **ContFramePool**, which provides the interface below for a Contiguous Frame Pool Manager [3]:

```
class ContFramePool {
private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */
public:
    static const unsigned int FRAME_SIZE = Machine::PAGE_SIZE;

    ContFramePool(unsigned long _base_frame_no,
                  unsigned long _n_frames,
                  unsigned long _info_frame_no);
    /* Initializes the data structures needed for the management of this
       frame pool.
       _base_frame_no: Number of first frame managed by this frame pool.
       _n_frames: Size, in frames, of this frame pool.
       _info_frame_no: Number of the first frame that should be used to store
           the management information for the frame pool. (Used if management
           information is stored EXTERNALLY.)
       NOTE: If _info_frame_no is 0, then the management information is stored
           INTERNALLY, and the frame pool is free to choose any frames from the
           pool to store management information. */

    unsigned long get_frames(unsigned int _n_frames);
    /* Allocates a number of contiguous frames from the frame pool.
       _n_frames: Size of contiguous physical memory to allocate,
       in number of frames.
       If successful, returns the frame number of the first frame.
       If fails, returns 0. */

    void mark_inaccessible(unsigned long _base_frame_no,
                           unsigned long _n_frames);
    /* Marks a contiguous area of physical memory, i.e., a contiguous
       sequence of frames, as inaccessible.
       _base_frame_no: Number of first frame to mark as inaccessible.
```

[3] This text provides a very brief description of the functions. Check File `cont_frame_pool.H` for details on the interface and File `cont_frame_pool.C` for a detailed description of one implementation approach.

```
        _n_frames: Number of contiguous frames to mark as inaccessible. */

    static void release_frames(unsigned long _first_frame_no);
    /* Releases a previously allocated contiguous sequence of frames
       back to its frame pool.
       The frame sequence is identified by the number of the first frame. */

    static unsigned long needed_info_frames(unsigned long _n_frames);
    /* Returns the number of frames needed to manage a frame pool of size
        _n_frames.
       The number returned here depends on the implementation of the frame pool
            and on the frame size. */
};
```

A frame pool can be implemented using a variety of ways, such as a free-list of frames, or a bit-map describing the availability of frames. The bit-map approach is particularly interesting in this setting, given that the amount of physical memory (and therefore the number of frames) is very small; a bit map for 32MB would need 8k bits, which easily fit into a single page. In fact, a frame has 4KB, and it could therefore be used to store a bitmap for a 128MB frame pool (4096 * 8bit makes for 32K frames, at 4KB each). Your implementation will need to provide a function `needed_info_frames` that returns the number of additional frames needed to manage the frame pool if that information shall be stored externally. (Check File `cont_frame_pool.H` for details.)

Note: There are some sections in the physical memory that should not be touched: Many locations within the first 1MB are used by the system and are therefore not available for the user. We safely ignore these portions because the kernel pool starts at 2MB anyway. There are other portions of memory that may not be accessible. For example, there is a region between 15MB and 16MB that may not be available, depending on the configuration of your system. This region is in the middle of our process frame pool. To address this, our frame pools support the capability to declare portions of the pool to be off-limits to the user. Such portions are defined through the function `mark_inaccessible`. Once a portion of memory is marked inaccessible, the pool will not allocate frames that belong to the given portion.[4]

Where to store the Memory Management Data Structures

Given that we don't have a memory manager yet, we find ourselves in a bit of a dilemma when it comes to storing the data structures needed for the memory management (in this case primarily management information for frame pools). For now we simply store the data structures on the stack - by defining the frame pool objects as variables that are local to the `main()` function. You will notice that in file `kernel.C` the two frame pools are defined in this way.

The management information for the pools, for example the bitmap, is too big to be stored on the stack, however. So it needs to be stored in one (or more) of the frames that are managed by the pool. Typically, the implementation of the frame pool would know that the first one or more frames of the pool are reserved for management purposes. This approach works fine for the kernel frame pool. It will become clear when we add paging in the next MP that this does not work for the process frame pool. Instead, we have to store the management information for the process frame pool inside frames of the kernel frame pool. This is the reason why the frame-pool

[4] We are very heavy-handed when it comes to managing available memory, and we basically assume that all memory between 2MB and 32MB is available, except for the missing region described above. The memory subsystem of a real OS would carefully interact with the BIOS to collect an accurate memory map of the system.

constructor needs support to place the information frames EXTERNALLY to the pool. In such a case, the kernel would create the process frame pool and specify one or more frames in the kernel pool to be used as information frames when creating the process frame pool. (See File `kernel.C` for how this is used.)

You talk about a Frame Manager, but I see only Frame Pools!

Indeed, there is no class called `FrameManager`. The functionality of the frame manager is hidden behind the class `ContFramePool`. We have constructors for frame pools, and get and release operations to manage frames. Think of the class `ContFramePool` as being the frame manager, which allows the creation of frame pools. The pool objects then deal with the allocation of frames.

The frame pool class becomes important again when we have to **release** frames. A caller identifies the frames to be released by their frame number. This in turn can cause difficulties when calling a pool's `release_frames` function directly, because we cannot guarantee that the pool even owns the frame. (This gets even worse once we have multiple processes, which could have "overlapping" pools.) Therefore, we call the static, and therefore class-wide function `release_frames`, which we can think of belonging to the frame manager.

The question now is, of course: *How does the frame manager know who the owner of the given frame is?* One solution is to maintain a list of frame pools as a static member of the class `ContFramePool`. Whenever a new frame pool is constructed, the constructor code adds the new pool to this list. Whenever frames are released, the static function `release_frames` first determines that frame pool that owns the frame, and then calls that frame pool's `release_frames` function. (Note that in the interface listed above we don't list the frame pool's `release_frames` function, but only the static function. The former function would be a private function, and therefore not part of the interface.)

The Assignment

1. Implement a simple frame pool manager as defined in file `cont_frame_pool.H`. The file `cont_frame_pool.C` contains empty definitions for the functions declared in the `.H` file. It also contains a detailed recipe for how to implement the frame pool. If you follow the steps described in this file, you will see that the implementation will be very simple and concise. If your code gets complicated you are doing something wrong.

2. If you need directions on how to implement a simple bitmap and how to use bit-wise operations to manipulate a bitmap, check out class `SimpleFramePool` in files `simple_frame_pool.H/C`. This is a rather poor and incomplete implementation of a frame pool that allocates one frame at a time. Because `SimpleFramePool` does not support contiguous allocations, and so does not have to deal with external fragmentation, it is presumably somewhat simpler to implement.

3. When you are done with the coding part of this assignment, submit your code as explained in the next section ("What to Hand in").

You should have access to a set of source files, BOCHS environment files, and a makefile that should make your implementation easier. Note that the file `cont_frame_pool.C` has empty implementations of the functions defined in `cont_frame_pool.H`. The compilation will succeed, but

the execution will crash as soon as any of these functions get called. It is your job to appropriately populate these functions.

Keep efficiency of your code in mind. For example, it may be helpful to not optimize the implementation of the bitmap initially if you choose to represent free frames in a bitmap. This makes the code easy to write and to debug. In the final submission you should avoid overly wasteful or inefficient implementations, however. You may run the risk of having points deducted.

On the Importance of Keeping Your Code DRY

The file `simple_frame_pool.C` has been provided to illustrate how bit operations can be used to implement a bitmap. For example, it illustrates how a getter/setter function pair is used to hide the implementation of the bitmap: The function `FrameState get_state(int frame_no)` would return the state of the given frame in the bitmap, while the function `void set_state(int frame_no, FrameState state)` would set the state for the given frame in the bitmap. This approach has several benefits:

1. You are not repeating your self, therefore the code is DRY. If your code has an error in the setting of the state, you need to modify the code in only one location, i.e. in the setter.

2. If you decide to change the implementation of the bitmap, you only need to change the setter and the getter (and the initialization of the bitmap, of course). This is super-simple!

What to Hand In

You are to hand in a ZIP file, with name `mp2.zip`, containing the following files:

1. A design document, called `design.pdf` (in PDF format) that describes your implementation of the frame pool. This document should clearly list which files have been modified. The assignment should not require modifications to files other than `cont_frame_pool.H` and `cont_frame_pool.C`. If you find yourself modifying and/or adding new files, clearly describe in the design document why you are modifying/adding these files.

2. All the source files and the `makefile` needed to compile the code.

The grader expects to unzip the submission file, type `make`, and then run your kernel binary. If these steps fail, the TA will deduct points.

Keep in mind that the grading of these MPs is very tedious. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the graders do not overlook any of your efforts.

Failure to follow the handin instructions will result in lost points.