MP 3: Page Table Management

Pranav Anantharam

UIN: 734003886

CSCE611: Operating System

Assigned Tasks

Main: Completed

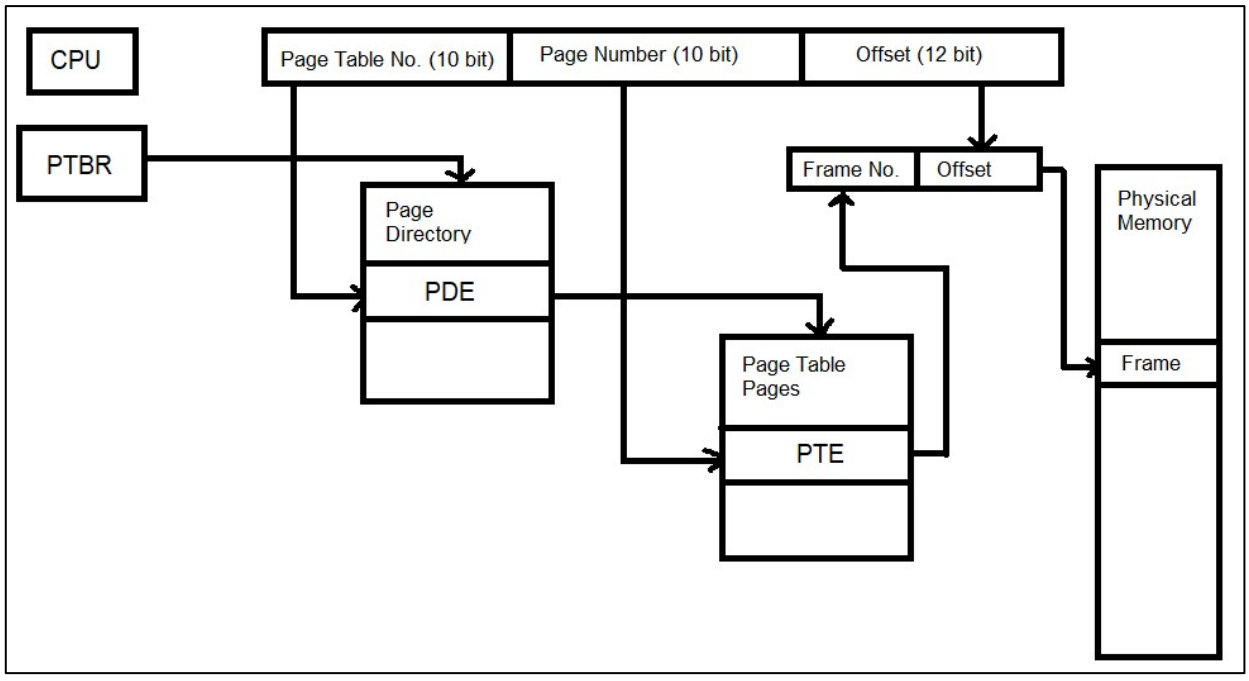
System Design

The goal of the machine problem is to setup and initialize the paging system and the page table infrastructure on the x86 architecture for a single address space. The x86 uses a two-level hierarchical paging scheme. The 32-bit logical address is split into a 10-bit page table number, a 10-bit page number and 12-bit offset. The page table base register points to the beginning of the page directory. The page table number is used to index into the page directory to find the page directory entry. The page directory entry contains the pointer to the beginning of the page table page associated with the page table number. The page number of the address is used to index into the page table page to identify the page table entry. Finally, the offset is concatenated with the frame number that is stored in the page table entry, and the result is the physical address. In this manner, the logical address is mapped to a physical address.

The memory layout of the machine is given below:

- Total memory in the machine = 32 MB
- Memory reserved for Kernel = 4 MB (Direct-mapped to physical memory)
- Process memory pool = 28 MB (Freely-mapped to physical memory)
- Frame size = 4 KB
- Inaccessible memory region = 15 MB – 16 MB

Two-level Hierarchical Paging Diagram:



Page faults occur when physical memory cannot be mapped to either the page directory or the page table page. In such cases, a new frame must be allocated using the frame manager, and the corresponding entries in the page directory or page table pages must be updated.

The paging system includes the following functionalities:

1. Page Table Initialization
2. Loading page table
3. Handling page faults (Algorithm discussed below)

List of files modified

1. page_table.C
2. cont_frame_pool.H
3. cont_frame_pool.C

Code Description

1. **page_table.C : init_paging():**

   This method is used to initialize the private variables for class PageTable. The global parameters for the paging subsystem are setup.

```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
   PageTable::kernel_mem_pool = _kernel_mem_pool;
   PageTable::process_mem_pool = _process_mem_pool;
   PageTable::shared_size = _shared_size;

   Console::puts("Initialized Paging System\n");
}
```

2. **page_table.C : PageTable():**

   This method is the constructor for class PageTable. It sets up entries in the page directory and the page table. The page directory is initialized by allocating a single frame from the kernel frame pool. The first page directory entry is marked as ‘valid’, while the remaining entries are marked ‘invalid’ using bitwise operations. Next, the page table is initialized by allocating a single frame from the kernel frame pool. The page table entries for the shared portion of the memory (first 4MB which is directly-mapped) are marked ‘valid’ using bitwise operations.

```
PageTable::PageTable()
{
   unsigned int index = 0;
   unsigned long address = 0;

   // Paging is disabled at first
   paging_enabled = 0;

   // Calculate number of frames shared - 4 MB / 4 KB = 1024
   unsigned long num_shared_frames = ( PageTable::shared_size / PAGE_SIZE );

   // Initialize page directory
   page_directory = (unsigned long *)( kernel_mem_pool->get_frames(1) * PAGE_SIZE );

   // Initialize page table
   unsigned long * page_table = (unsigned long *)( kernel_mem_pool->get_frames(1) * PAGE_SIZE );

   // Initializing page directory entries (PDEs) - Mark 1st PDE as valid
   page_directory[0] = ( (unsigned long)page_table | 0b11 );            // Supervisor level, R/W and Present bits are set

   // Remaining PDEs are marked invalid
   for( index = 1; index < num_shared_frames; index++ )
   {
      page_directory[index] = ( page_directory[index] | 0b10 );         // Supervisor level and R/W bits are set, Present bit is not set
   }

   // Mapping first 4 MB of memory for page table - All pages marked as valid
   for( index = 0; index < num_shared_frames; index++ )
   {
      page_table[index] = (address | 0b11);                             // Supervisor level, R/W and Present bits are set
      address = address + PAGE_SIZE;
   }

   Console::puts("Constructed Page Table object\n");
}
```

3. **page_table.C : load():**

This method is used to load in a page table by storing the address of the page directory in the page table base register by writing the address into register CR3. Overall, a new page table is loaded and made the current table.

```
void PageTable::load()
{
   current_page_table = this;

   // Write page directory address into CR3 register
   write_cr3((unsigned long)(current_page_table->page_directory));

   Console::puts("Loaded page table\n");
}
```

4. **page_table.C : enable_paging():**

This method is used to enable paging on the CPU. This is done by setting the 32nd bit to 1 and writing the value into register CR0. The ‘paging_enabled’ variable is also updated to 1.

```
void PageTable::enable_paging()
{
   write_cr0( read_cr0() | 0x80000000 );     // Set paging bit - 32nd bit
   paging_enabled = 1;                       // Set paging_enabled variable

   Console::puts("Enabled paging\n");
}
```

5. **page_table.C : handle_fault():**

This method is used to handle page-fault exceptions of the CPU. This method will look up the appropriate entry in the page table and decide how to handle the page fault. If there is no physical memory frame associated with the page, an available frame is brought in and the page table entry is updated. If a new page table has to be initialized, a new frame is allocated for that, and the new page table page and the directory are updated accordingly. This method follows the below algorithm:

a) Obtain the page fault address by reading register CR2.
b) Obtain the page directory address by reading register CR3.
c) Calculate the page directory index (first 10 bits) and page table index (next 10 bits) from the page fault address.
d) Read the error code of Exception 14 and check if bit 0 is not set to verify whether 'page not present' fault occurred.
e) If 'page not present' fault occurred, next we must check whether the page fault is in the page directory or the page table. This is done by checking the 'Present field' of the page directory address.

31 ··· 12	11 ··· 9	8 ··· 7	6	5	4 ··· 3	2	1	0
Page frame	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

f) If page fault occurred in the page directory, then we allocate a free frame from the kernel pool for the page directory entry and mark the page directory entry as 'valid' using bitwise operations.
g) Next, we initialize the corresponding page table by allocating a free frame from the process pool and mark all the entries in the page table as 'invalid' using bitwise operations.
h) If page fault occurred in the page table, then we allocate a free frame from the process pool for the page table entry and mark the page table entry as 'valid' using bitwise operations.
i) Return from the page fault handler.
j) If a page fault occurred in the page directory previously, then another page fault is raised since the corresponding page is not present in the page table. This time, an available free frame is allocated from the process pool and the page is marked as 'valid' using bitwise operations.
k) A page fault exception is not raised the next time since the logical address exists in the paging system.

```
void PageTable::handle_fault(REGS * _r)
{
    unsigned int index = 0;

    unsigned long error_code = _r->err_code;

    // Get the page fault address from CR2 register
    unsigned long fault_address = read_cr2();

    // Get the page directory address from CR3 register
    unsigned long * page_dir = (unsigned long *)read_cr3();

    // Extract page directory index - first 10 bits
    unsigned long page_dir_index = (fault_address >> 22);

    // Extract page table index using mask - next 10 bits
    unsigned long page_table_index = ( (fault_address >> 12) & 0x3FF );     // 0x3FF = 001111111111 - retain only last 10 bits

    unsigned long * new_page_table = NULL;

    // If page not present fault occurs
    if ((error_code & 1) == 0 )
    {
        // Check where page fault occured
        if ((page_dir[page_dir_index] & 1 ) == 0)
        {
            // Page fault occured in page directory - PDE is invalid

            // Allocate a frame from kernel pool for page directory and mark flags - supervisor, R/W, Present bits
            page_dir[page_dir_index] = (unsigned long)(kernel_mem_pool->get_frames(1)*PAGE_SIZE | 0b11);        // PDE marked valid

            // Clear last 12 bits to erase all flags using mask
            new_page_table = (unsigned long *)(page_dir[page_dir_index] & 0xFFFFF000);

            // Set flags for each page - PTEs marked invalid
            for(index=0; index < 1024; index++)
            {
                // Set user level flag bit
                new_page_table[index] = 0b100;
            }
        }
```

```
        else
        {
            // Page fault occured in page table page - PDE is present, but PTE is invalid

            // Clear last 12 bits to erase all flags using mask
            new_page_table = (unsigned long *)(page_dir[page_dir_index] & 0xFFFFF000);

            // Allocate a frame from process pool and mark flags - supervisor, R/W, Present bits
            new_page_table[page_table_index] =  PageTable::process_mem_pool->get_frames(1)*PAGE_SIZE | 0b11;     // PTE marked valid
        }
    }

    Console::puts("handled page fault\n");
}
```

6. **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

```
class ContFramePool {

private:

    unsigned char * bitmap;         // Bitmap for Cont Frame Pool
    unsigned int    nFreeFrames;    // Number of free frames
    unsigned long   base_frame_no;  // Frame number at start of physical memory region
    unsigned long   nframes;        // Number of frames in frame pool
    unsigned long   info_frame_no;  // Frame number at start of management info in physical memory
    ContFramePool * next;           // Frame Pool Linked List next pointer

    /* ---- STATE MANAGEMENT */

    enum class FrameState {Free, Used, HoS};

    FrameState get_state(unsigned long _frame_no);
    void set_state(unsigned long _frame_no, FrameState _state);

public:

    static ContFramePool * head;    // Frame Pool Linked List head pointer

    // The frame size is the same as the page size, duh...
    static const unsigned int FRAME_SIZE = Machine::PAGE_SIZE;
```

7. **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.

```
void release_frames_in_pool(unsigned long _first_frame_no);
```

8. **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.

```
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _n_frames,
                             unsigned long _info_frame_no)
{
    base_frame_no = _base_frame_no;
    nframes = _n_frames;
    info_frame_no = _info_frame_no;
    nFreeFrames = _n_frames;

    // 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
    if(info_frame_no == 0)
    {
        bitmap = (unsigned char *) (base_frame_no * FRAME_SIZE);
    }
    else
    {
        bitmap = (unsigned char *) (info_frame_no * FRAME_SIZE);
    }

    assert( (nframes%8) == 0 );

    // Initializing all bits in bitmap to zero
    for(int fno = 0; fno < _n_frames; fno++)
    {
        set_state(fno, FrameState::Free);
    }

    // Mark the first frame as being used if it is being used
    if( _info_frame_no == 0 )
    {
        set_state(0, FrameState::Used);
        nFreeFrames = nFreeFrames - 1;
    }
```

```
    // Creating a linked list and adding a new frame pool
    if( head == NULL )
    {
        head = this;
        head->next = NULL;
    }
    else
    {
        // Adding new frame pool to existing linked list
        ContFramePool * temp = NULL;
        for(temp = head; temp->next != NULL; temp = temp->next);
        temp->next = this;
        temp = this;
        temp->next = NULL;
    }

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

9. **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) → 3rd row
Bitmap column index = (11 % 4) * 2 = 6 → 7th bit
Hence, the desired frame state bits are the 7th and 8th bits of the 3rd bitmap element (bitmap[2]).

```
ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no)
{
    unsigned int bitmap_row_index = (_frame_no / 4);
    unsigned int bitmap_col_index = ((_frame_no % 4)*2);
    unsigned char mask_result = (bitmap[bitmap_row_index] >> (bitmap_col_index)) & 0b11;
    FrameState state_output = FrameState::Used;

#if DEBUG
        Console::puts("get_state row index ="); Console::puti(bitmap_row_index); Console::puts("\n");
        Console::puts("get_state col index ="); Console::puti(bitmap_col_index); Console::puts("\n");
        Console::puts("get_state bitmap value = "); Console::puti(bitmap[bitmap_row_index]); Console::puts("\n");
        Console::puts("get_state mask result ="); Console::puti(mask_result); Console::puts("\n");
#endif

    if( mask_result == 0b00 )
    {
        state_output = FrameState::Free;
#if DEBUG
        Console::puts("get_state state_output = Free\n");
#endif

    }
    else if( mask_result == 0b01 )
    {
        state_output = FrameState::Used;
#if DEBUG
        Console::puts("get_state state_output = Used\n");
#endif
    }
    else if( mask_result == 0b11 )
    {
        state_output = FrameState::HoS;
#if DEBUG
        Console::puts("get_state state_output = HoS\n");
#endif
    }

    return state_output;
}
```

10. **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.

```
void ContFramePool::set_state(unsigned long _frame_no, FrameState _state)
{
    unsigned int bitmap_row_index = (_frame_no / 4);
    unsigned int bitmap_col_index = ((_frame_no % 4)*2);

#if DEBUG
        Console::puts("set_state row index ="); Console::puti(bitmap_row_index); Console::puts("\n");
        Console::puts("set_state col index ="); Console::puti(bitmap_col_index); Console::puts("\n");
        Console::puts("set_state bitmap value before = "); Console::puti(bitmap[bitmap_row_index]); Console::puts("\n");
#endif

    switch(_state)
    {
        case FrameState::Free:
            bitmap[bitmap_row_index] &= ~(3<<bitmap_col_index);
            break;
        case FrameState::Used:
            bitmap[bitmap_row_index] ^= (1<<bitmap_col_index);
            break;
        case FrameState::HoS:
            bitmap[bitmap_row_index] ^= (3<<bitmap_col_index);
            break;
    }

#if DEBUG
        Console::puts("set_state bitmap value after = "); Console::puti(bitmap[bitmap_row_index]); Console::puts("\n");
#endif

    return;
}
```

11. **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.

```
unsigned long ContFramePool::get_frames(unsigned int _n_frames)
{
    if( (_n_frames > nFreeFrames) || (_n_frames > nframes) )
    {
        Console::puts("ContFramePool::get_frames Invalid Request - Not enough free frames available!\n ");
        assert(false);
        return 0;
    }

    unsigned int index = 0;
    unsigned int free_frames_start = 0;
    unsigned int available_flag = 0;
    unsigned int free_frames_count = 0;
    unsigned int output = 0;

    for( index = 0; index < nframes; index++)
    {
        if( get_state(index) == FrameState::Free )
        {
            if(free_frames_count == 0)
            {
                // Save free frames start frame_no
                free_frames_start = index;
            }

            free_frames_count = free_frames_count + 1;

            // If free_frames_count is equal to the required num of frames
            if( free_frames_count == _n_frames )
            {
                available_flag = 1;
                break;
            }
        }
```

```
        else
        {
            free_frames_count = 0;
        }
    }

    if( available_flag == 1 )
    {
        // Contiguous frames are available from free_frames_start
        for( index = free_frames_start; index < (free_frames_start + _n_frames); index++ )
        {
            if( index == free_frames_start )
            {
#if DEBUG
        Console::puts("get_frames Operation = HoS\n");
#endif
                set_state( index, FrameState::HoS);
            }
            else
            {
#if DEBUG
        Console::puts("get_frames Operation = Used\n");
#endif
                set_state( index, FrameState::Used );
            }
        }

        nFreeFrames = nFreeFrames - _n_frames;
        output = free_frames_start + base_frame_no;
    }
    else
    {
        output = 0;
        Console::puts("ContframePool::get_frames - Continuous free frames not available\n");
        assert(false);
    }

    return output;
}
```

12. **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.

```
void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
                                      unsigned long _n_frames)
{
    if( (_base_frame_no + _n_frames ) > (base_frame_no + nframes) || (_base_frame_no < base_frame_no) )
    {
        Console::puts("ContframePool::mark_inaccessible - Range out of bounds. Cannot mark inacessible.\n");
        assert(false);
        return;
    }

#if DEBUG
    Console::puts("Mark Inaccessible: _base_frame_no = "); Console::puti(_base_frame_no);
    Console::puts(" _n_frames ="); Console::puti(_n_frames);Console::puts("\n");
#endif

    unsigned int index = 0;
    for( index = _base_frame_no; index < (_base_frame_no + _n_frames); index++ )
    {
        if( get_state(index - base_frame_no) == FrameState::Free )
        {
            if( index == _base_frame_no )
            {
                set_state( (index - base_frame_no), FrameState::HoS);
            }
            else
            {
                set_state( (index - base_frame_no), FrameState::Used );
            }

            nFreeFrames = nFreeFrames - 1;
        }
#if DEBUG
        else
        {
            Console::puts("ContframePool::mark_inaccessible - Frame = "); Console::puti(index); Console::puts(" already marked inaccessible.\n");
            assert(false);
        }
#endif
    }

    return;
}
```

13. **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.

```
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    unsigned int found_frame = 0;
    ContFramePool * temp = head;

#if DEBUG
    Console::puts("In release_frames: First frame no ="); Console::puti(_first_frame_no); Console::puts("\n");
#endif

    // To find which pool the frame belongs to
    while( temp != NULL )
    {

#if DEBUG
        Console::puts("In release_frames: Base frame lower ="); Console::puti(temp->base_frame_no); Console::puts("\n");
        Console::puts("In release_frames: Base frame upper ="); Console::puti(temp->base_frame_no+temp->nframes); Console::puts("\n");
#endif
        if( (_first_frame_no >= temp->base_frame_no) && (_first_frame_no < (temp->base_frame_no + temp->nframes)) )
        {
            found_frame = 1;
            temp->release_frames_in_pool(_first_frame_no);
            break;
        }

        temp = temp->next;
    }

    if( found_frame == 0 )
    {
        Console::puts("ContframePool::release_frames - Cannot release frame. Frame not found in frame pools.\n");
        assert(false);
    }

    return;
}
```

14. **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.

```
void ContFramePool::release_frames_in_pool(unsigned long _first_frame_no)
{
    unsigned int index = 0;

    // Get the state of frame
    if( get_state(_first_frame_no - base_frame_no) == FrameState::HoS )
    {
        for( index = _first_frame_no; index < (_first_frame_no + nframes); index++)
        {
            set_state((index - base_frame_no), FrameState::Free);
            nFreeFrames = nFreeFrames + 1;
        }
    }
    else
    {
        Console::puts("ContframePool::release_frames_in_pool - Cannot release frame. Frame state is not HoS.\n");
        assert(false);
    }
}
```

15. **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

```
unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
{
    // Since we use 2 bits per frame
    return ( (_n_frames*2) / (4*1024*8) ) + ( ( (_n_frames*2) % (4*1024*8) ) > 0 ? 1 : 0 );
}
```

Testing

Page faults were generated and outputs were observed for the following cases:

Serial No.	Testcase	Desired Result	Actual Result
1	Fault Address: 4 MB, Requested Memory: 1 MB	All page faults are handled; data written and read are read are equivalent	PASS All page faults are handled; data written and read are read are equivalent
2	Fault Address: 4 MB, Requested Memory: 16 MB	All page faults are handled; data written and read are equivalent	PASS All page faults are handled; data written and read are equivalent
2	Fault Address: 4 MB, Requested Memory: 27 MB	All page faults are handled; data written and read are equivalent	PASS All page faults are handled; data written and read are equivalent
3	Fault Address: 4 MB, Requested Memory: 28 MB	Not enough free frames are available to allocate	PASS Insufficient number of free frames available
4	Fault Address: 8 MB, Requested Memory: 16 MB	All page faults are handled; data written and read are equivalent	PASS All page faults are handled; data written and read are equivalent
5	Fault Address: 1024 MB, Requested Memory: 16 MB	All page faults are handled; data written and read are equivalent	PASS All page faults are handled; data written and read are equivalent

Minor modifications were made in kernel.C file to perform the above testcases and call each of the implemented methods (init_paging, load, enable_paging, handle_fault); and verify the output. I have tested the basic functionality for all implemented methods for varying fault addresses and memory allocation requests.

Outputs:

1. Fault Address: 4 MB, Requested Memory: 1 MB

```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((1 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

```
Installing handler in IDT position 47
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <1>
Frame Pool initialized
Frame Pool initialized
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
WE TURNED ON PAGING!
If we see this message, the page tables have been
set up mostly correctly.
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
```

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
DONE WRITING TO MEMORY. Press keyboard to continue testing...
One second has passed
One second has passed
TEST PASSED.
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
```

2. Fault Address: 4 MB, Requested Memory: 27 MB

```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((27 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
DONE WRITING TO MEMORY. Press keyboard to continue testing...
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
TEST PASSED.
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
```

3. Fault Address: 4 MB, Requested Memory: 28 MB

```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((28 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
ContFramePool::get_frames Invalid Request - Not enough free frames available!
 Assertion failed at file: cont_frame_pool.C line: 263 assertion: false
```

4. Fault Address: 1024 MB, Requested Memory: 16 MB

```
#define FAULT_ADDR (1024 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((16 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
DONE WRITING TO MEMORY. Press keyboard to continue testing...
One second has passed
One second has passed
One second has passed
TEST PASSED.
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
```