MP3: Page Table Management

Vinay Balamurali
UIN: 936002032
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

This machine problem aims to setup and initialize a paging mechanism for a single address space or process on the x86 architecture. The structure of the memory is as follows:

- 32 MB of total memory.
- Frame size is 4 KB
- The first 4 MB are directly mapped to physical memory, meaning that the logical address will be the same as the physical address.
  - First 1 MB is reserved for global data and memory-mapped devices such as the video display.
  - Between 1 MB and 2 MB resides the Kernel code and stack.
  - Between 2 MB and 4 MB is the Kernel Frame Pool.
- Memory beyond 4 MB are freely mapped which means frames can be randomly allocated to pages based on availability.

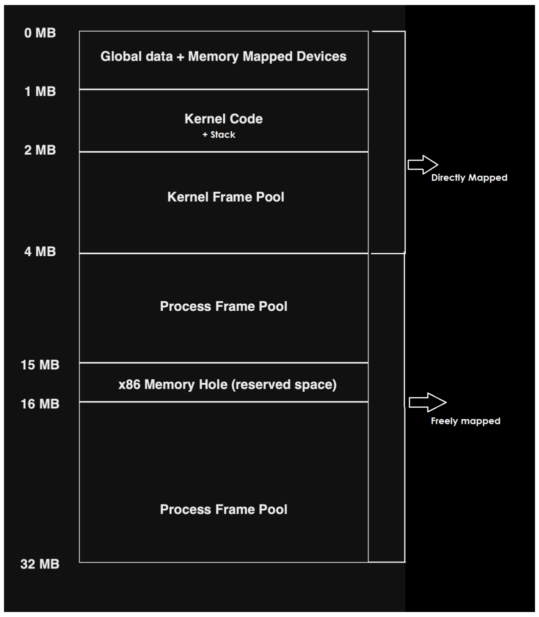


Figure 1: Memory Layout

Implementation of paging on the x86 architecture:

- 32-bit Virtual Address space employing a 2-level paging schema.
- 32-bit Virtual Address is split as follows:
  - Lower 12 bits for **Offset**
  - Middle 10 bits for **Page Table Page Index**
  - Higher 10 bits for **Page Directory Entry Index**
- Page Directory Table is allocated in a frame gotten from the Kernel Memory pool.
- Page Table Pages including the mapping for the first 4 MB are gotten from the Kernel Memory pool.
- Test code where memory is allocated in real-time mimics the application processes. The memory for this sector is pulled from the Process Memory pool.
- The Page directory is initialized and loaded into register CR3. Paging is enabled by writing into register CR0 and the CPU starts issuing logical memory addresses.
- When a memory address is determined to be not present, the exception handler 14 (for Page Faults) is called. The handler determines the faulting address, and tries to allocate a frame associated with that particular address in memory and re-issues the memory request.

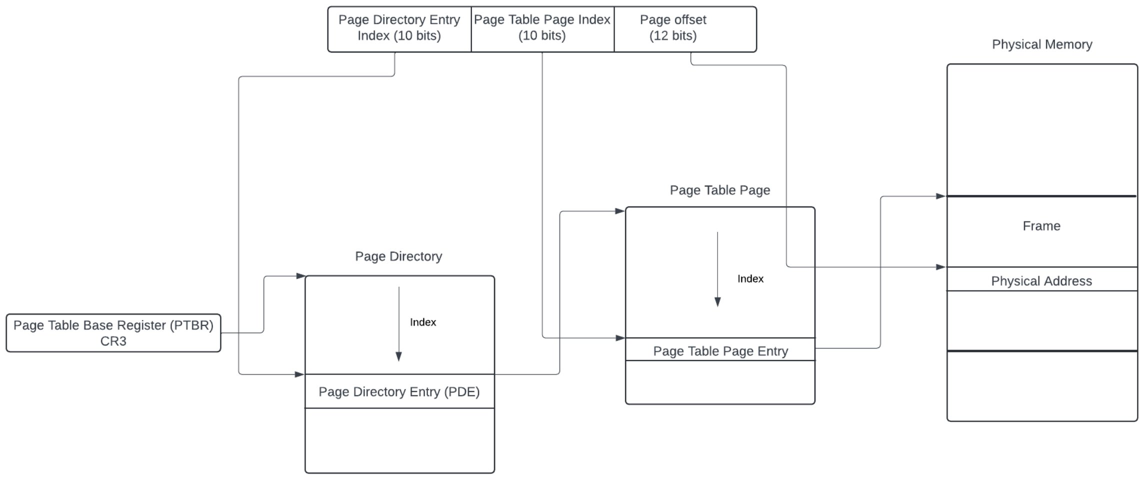

Page Directory Entry Index (10 bits)
Page Table Page Index (10 bits)
Page offset (12 bits)
Physical Memory
Page Table Page
Frame
Index
Physical Address
Page Directory
Index
Page Table Base Register (PTBR) CR3
Page Table Page Entry
Page Directory Entry (PDE)

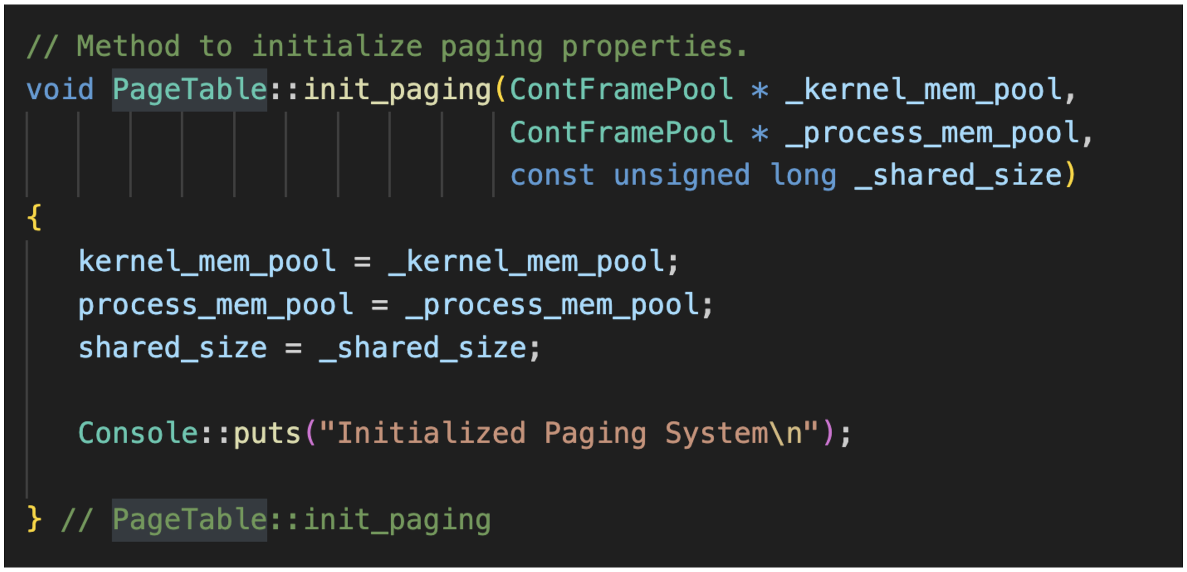
Figure 2: Two-level Paging mechanism on x86

Code Description

0.1 Files Modified

- page_table.C
- kernel.C
- cont_frame_pool.H
- makefile

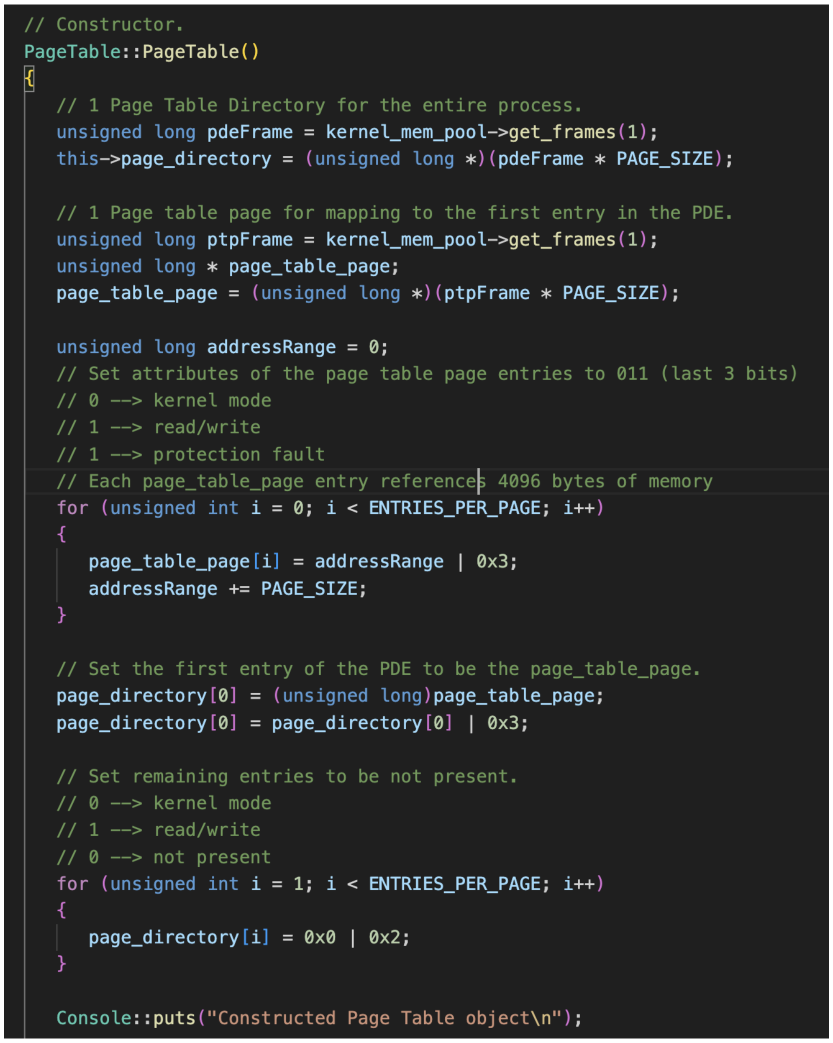
page_table.C: PageTable::init_paging() : This method is primarily responsible for setting the static members for this class. The static members are shared across instances of this class. This method is typically called before an instance of this class is declared.



```
// Method to initialize paging properties.
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
    kernel_mem_pool = _kernel_mem_pool;
    process_mem_pool = _process_mem_pool;
    shared_size = _shared_size;

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

} // PageTable::init_paging
```

Figure 3: Init paging method

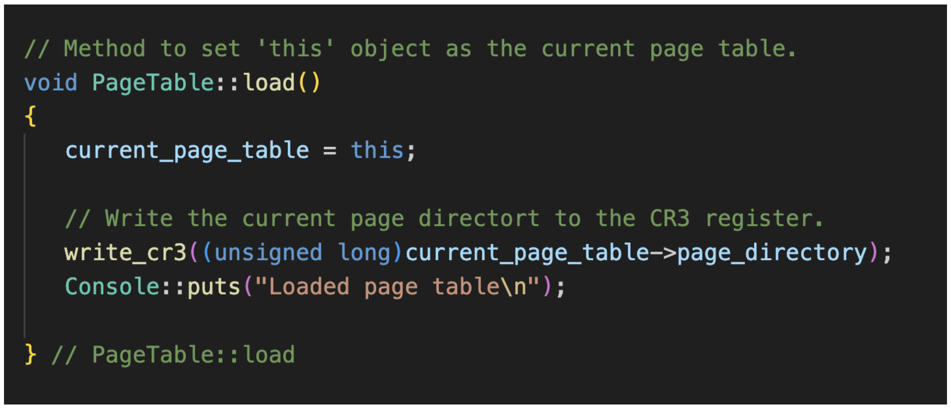
page_table.C: PageTable::PageTable() : The constructor is responsible for mapping the first 4 MP of memory by pulling frames from the Kernel memory pool and initializing the Page Directory table with the 0^{th} entry and its corresponding Page Table page.



```
// Constructor.
PageTable::PageTable()
{
    // 1 Page Table Directory for the entire process.
    unsigned long pdeFrame = kernel_mem_pool->get_frames(1);
    this->page_directory = (unsigned long *)(pdeFrame * PAGE_SIZE);

    // 1 Page table page for mapping to the first entry in the PDE.
    unsigned long ptpFrame = kernel_mem_pool->get_frames(1);
    unsigned long * page_table_page;
    page_table_page = (unsigned long *)(ptpFrame * PAGE_SIZE);

    unsigned long addressRange = 0;
    // Set attributes of the page table page entries to 011 (last 3 bits)
    // 0 --> kernel mode
    // 1 --> read/write
    // 1 --> protection fault
    // Each page_table_page entry references 4096 bytes of memory
    for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++)
    {
        page_table_page[i] = addressRange | 0x3;
        addressRange += PAGE_SIZE;
    }

    // Set the first entry of the PDE to be the page_table_page.
    page_directory[0] = (unsigned long)page_table_page;
    page_directory[0] = page_directory[0] | 0x3;

    // Set remaining entries to be not present.
    // 0 --> kernel mode
    // 1 --> read/write
    // 0 --> not present
    for (unsigned int i = 1; i < ENTRIES_PER_PAGE; i++)
    {
        page_directory[i] = 0x0 | 0x2;
    }

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

Figure 4: Constructor

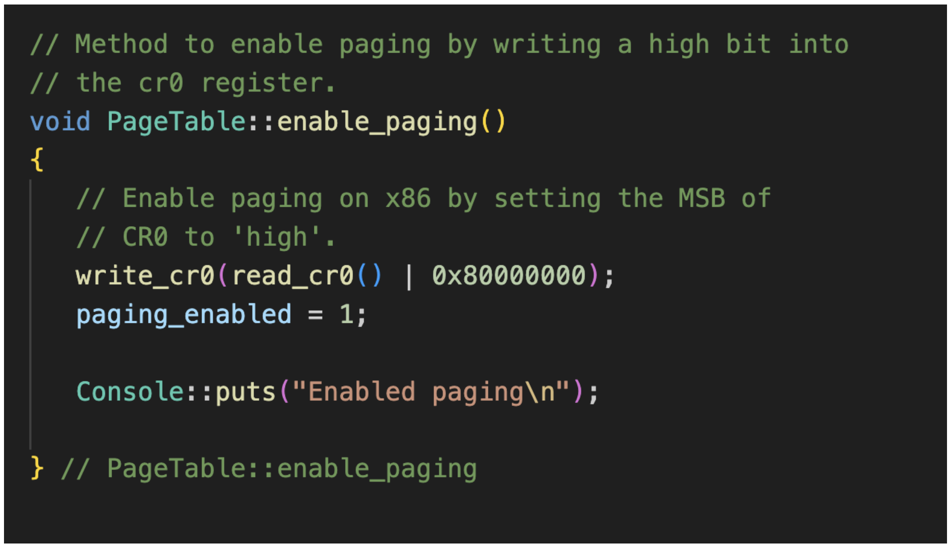
page_table.C: PageTable::load() : The appropriate page table object is pointed to in this method. Although it is not useful now, it would come in handy when there would be multiple processes to handle. Aside, from this the *load()* method also writes the initialized Page Directory to the register CR3.



```
// Method to set 'this' object as the current page table.
void PageTable::load()
{
    current_page_table = this;

    // Write the current page directort to the CR3 register.
    write_cr3((unsigned long)current_page_table->page_directory);
    Console::puts("Loaded page table\n");

} // PageTable::load
```

Figure 5: Load method

page_table.C: PageTable::enable_paging() : Paging on the x86 is enabled within this method This is done by setting the MSB of register CR0.

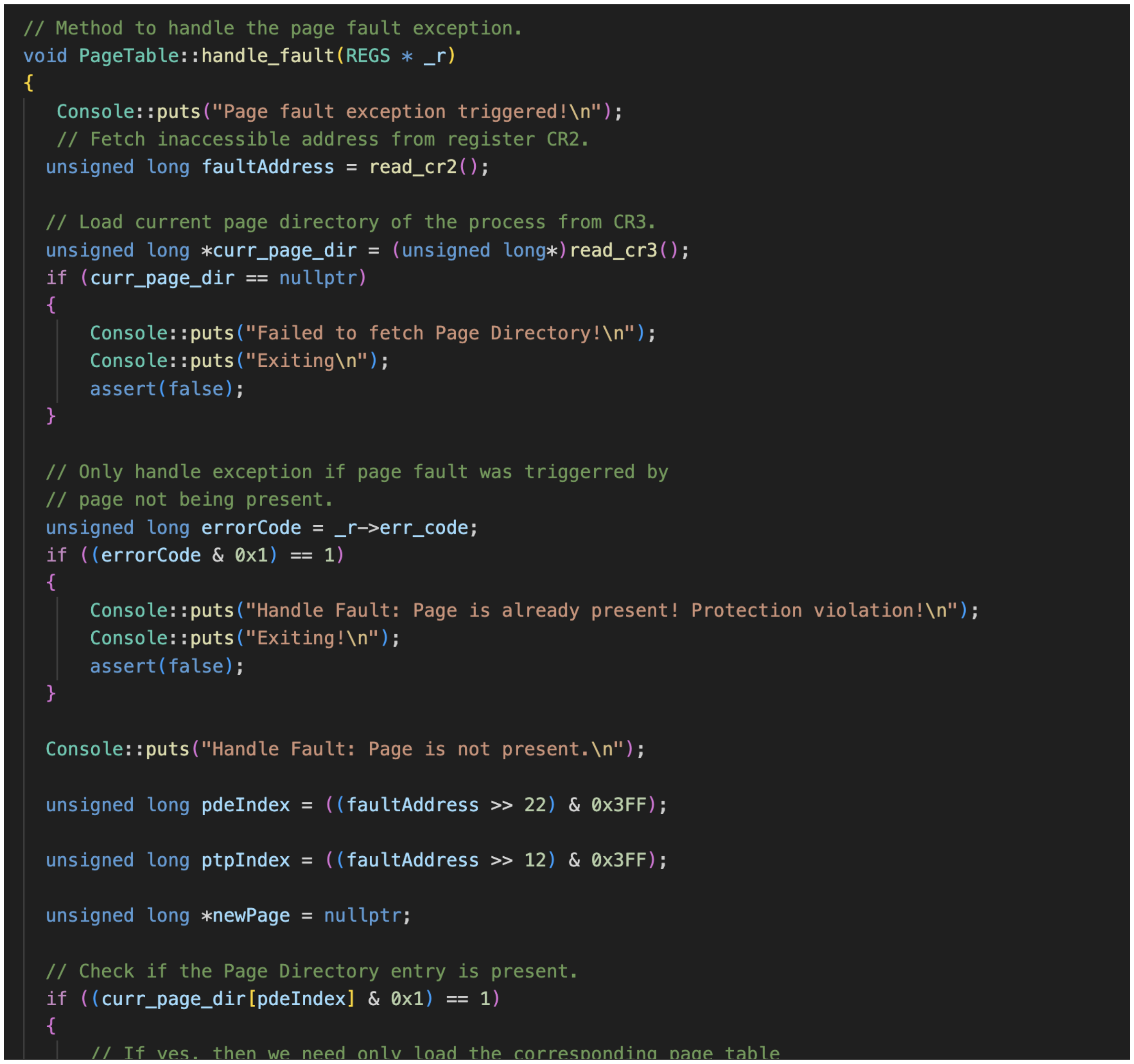


```
// Method to enable paging by writing a high bit into
// the cr0 register.
void PageTable::enable_paging()
{
    // Enable paging on x86 by setting the MSB of
    // CR0 to 'high'.
    write_cr0(read_cr0() | 0x80000000);
    paging_enabled = 1;

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

} // PageTable::enable_paging
```

Figure 6: Enable paging method

page_table.C: PageTable::handle_fault() : This method is the most essential part of the paging mechanism. The algorithm is described as follows:

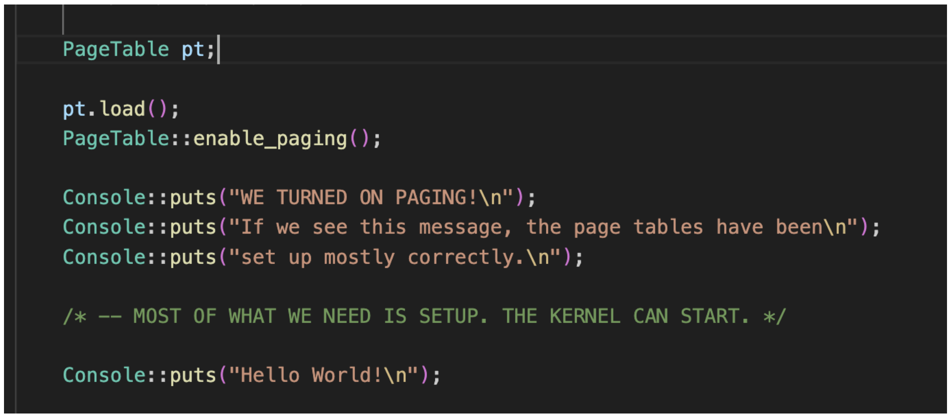
- Fetch the faulting address from register CR2.
- Fetch the current page directory from register CR3.
- Check the *err_code* pushed on to the stack. If the LSB is set, then a protection violation occurred and it can't be handled at present. This is out of scope. If the LSB is 0, then the page is not present in memory and further processing is required.
- Use bit masks to parse the *Page Directory Index* and the *Page Table Page Index*.
- If bit 0 (LSB) is set when indexed to to the *Current Page Directory* using the *Page Directory Index*, then the PDE reference is valid but a corresponding Page Table entry is not present. This must be done by first allocating a frame from ther *process memory pool* and setting the table entry.
- If the bit 0 is not set, then the PDE entry is invalid. First, a Page Table Page must be allocated in the kernel frame pool. Post that a mapping must be created to the new PDE entry.
- Return from page fault handler and the address is re-issued by the CPU.



```
// Method to handle the page fault exception.
void PageTable::handle_fault(REGS * _r)
{
   Console::puts("Page fault exception triggered!\n");
   // Fetch inaccessible address from register CR2.
  unsigned long faultAddress = read_cr2();

  // Load current page directory of the process from CR3.
  unsigned long *curr_page_dir = (unsigned long*)read_cr3();
  if (curr_page_dir == nullptr)
  {
      Console::puts("Failed to fetch Page Directory!\n");
      Console::puts("Exiting\n");
      assert(false);
  }

  // Only handle exception if page fault was triggerred by
  // page not being present.
  unsigned long errorCode = _r->err_code;
  if ((errorCode & 0x1) == 1)
  {
      Console::puts("Handle Fault: Page is already present! Protection violation!\n");
      Console::puts("Exiting!\n");
      assert(false);
  }

  Console::puts("Handle Fault: Page is not present.\n");

  unsigned long pdeIndex = ((faultAddress >> 22) & 0x3FF);

  unsigned long ptpIndex = ((faultAddress >> 12) & 0x3FF);

  unsigned long *newPage = nullptr;

  // Check if the Page Directory entry is present.
  if ((curr_page_dir[pdeIndex] & 0x1) == 1)
  {
      // If yes, then we need only load the corresponding page table
```

Figure 7: Page exception handler

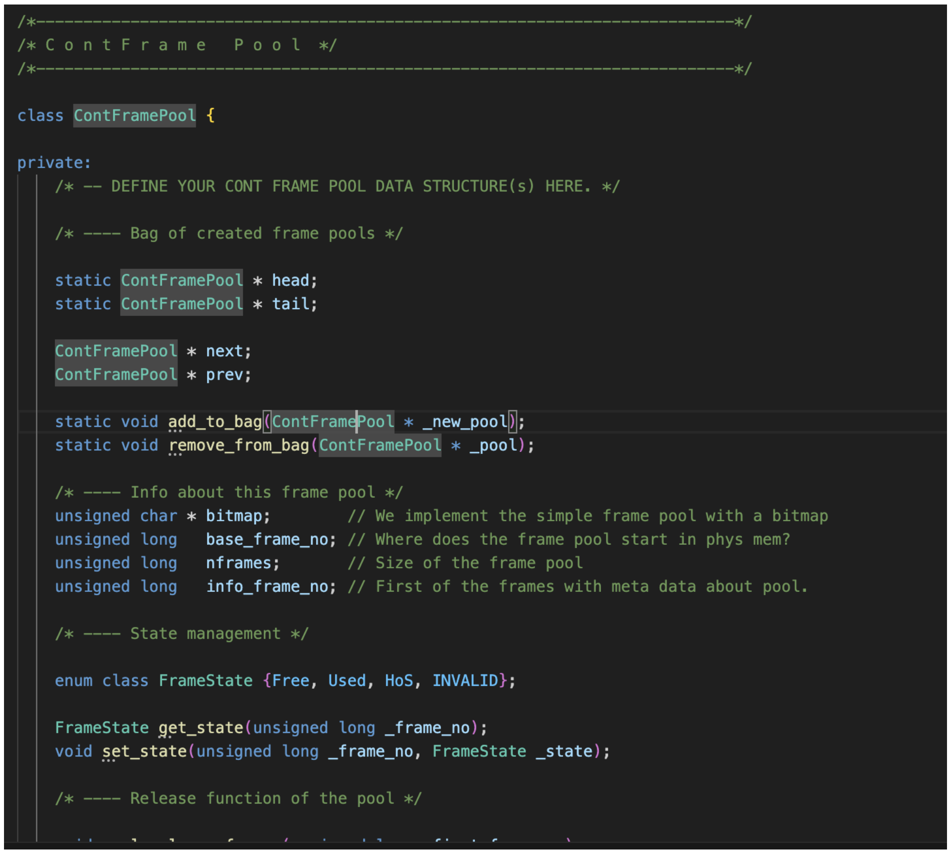
kernel.C: main() : The method *enable_paging()* has been moved before the setup messages are printed.



```
    PageTable pt;

    pt.load();
    PageTable::enable_paging();

    Console::puts("WE TURNED ON PAGING!\n");
    Console::puts("If we see this message, the page tables have been\n");
    Console::puts("set up mostly correctly.\n");

    /* -- MOST OF WHAT WE NEED IS SETUP. THE KERNEL CAN START. */

    Console::puts("Hello World!\n");
```

Figure 8: main()

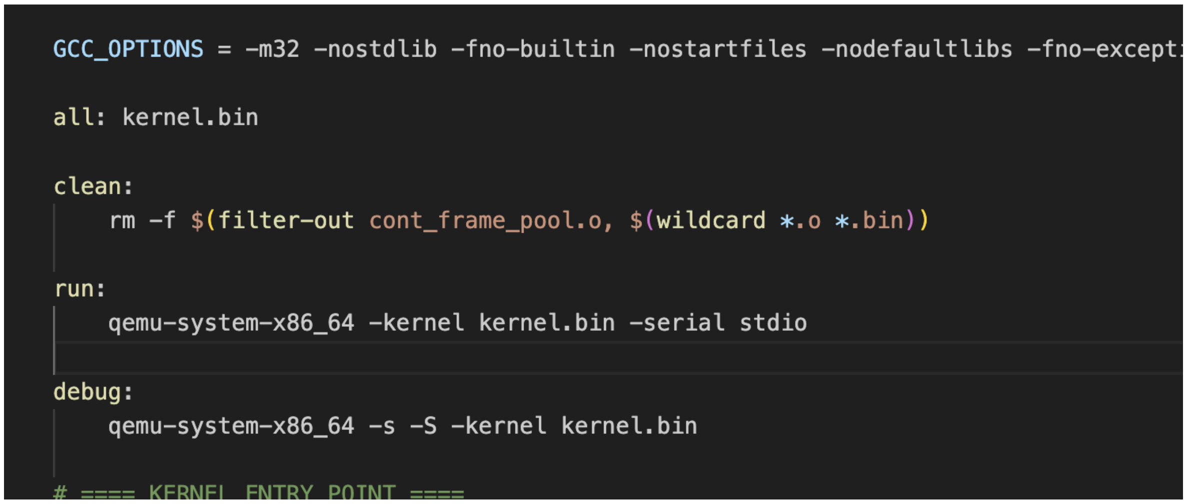
cont_frame_pool.H : This file has been taken from the official solutions as is. Please include the official solution object file *cont_frame_pool.o* to ensure 'make' succeeds in compilation.



```
/*--------------------------------------------------------------------------*/
/* C o n t F r a m e   P o o l */
/*--------------------------------------------------------------------------*/

class ContFramePool {

private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */

    /* ---- Bag of created frame pools */

    static ContFramePool * head;
    static ContFramePool * tail;

    ContFramePool * next;
    ContFramePool * prev;

    static void add_to_bag(ContFramePool * _new_pool);
    static void remove_from_bag(ContFramePool * _pool);

    /* ---- Info about this frame pool */
    unsigned char * bitmap;        // We implement the simple frame pool with a bitmap
    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; // First of the frames with meta data about pool.

    /* ---- State management */

    enum class FrameState {Free, Used, HoS, INVALID};

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

    /* ---- Release function of the pool */
```

Figure 9: cont_frame_pool.H

makefile : Since *cont_frame_pool.o* has been used from the official solution. We need to ensure this particular file is not removed when we run *make clean*. For this we slightly, modify the rule for *make clean* to filter out the particular file. Additionally, the rule to build *cont_frame_pool.C* has been removed as well.

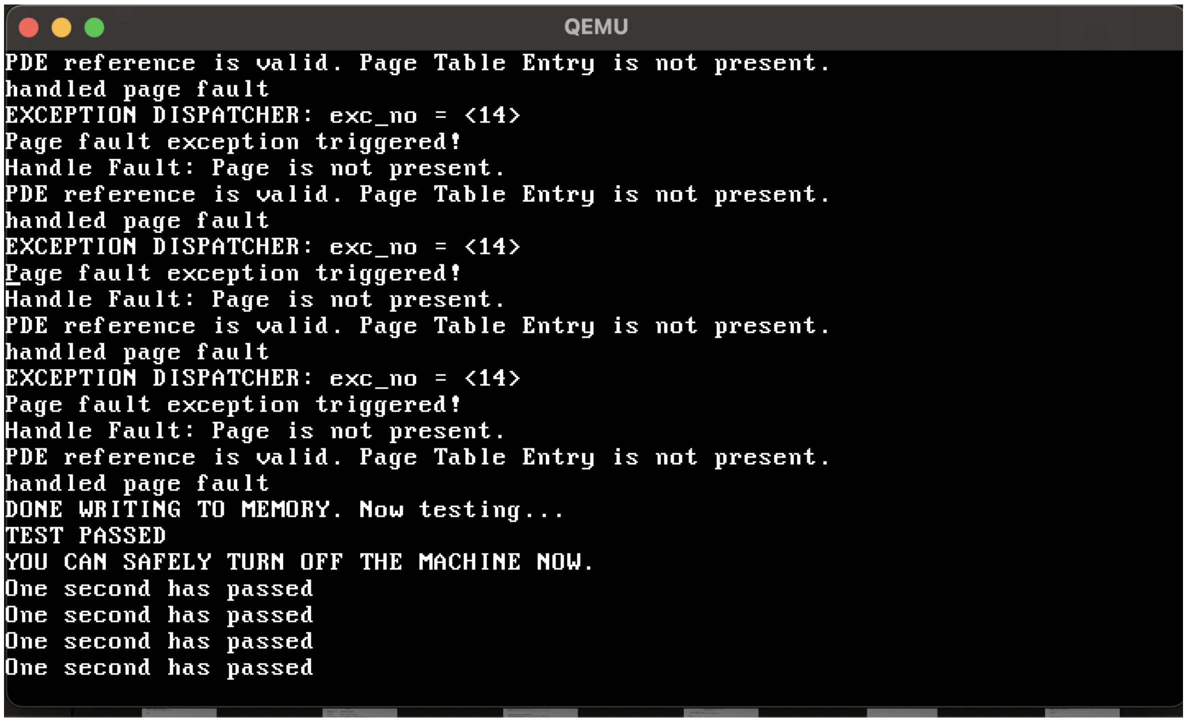


```
GCC_OPTIONS = -m32 -nostdlib -fno-builtin -nostartfiles -nodefaultlibs -fno-except

all: kernel.bin

clean:
    rm -f $(filter-out cont_frame_pool.o, $(wildcard *.o *.bin))

run:
    qemu-system-x86_64 -kernel kernel.bin -serial stdio

debug:
    qemu-system-x86_64 -s -S -kernel kernel.bin

# ==== KERNEL ENTRY POINT ====
```

Figure 10: Makefile

Testing

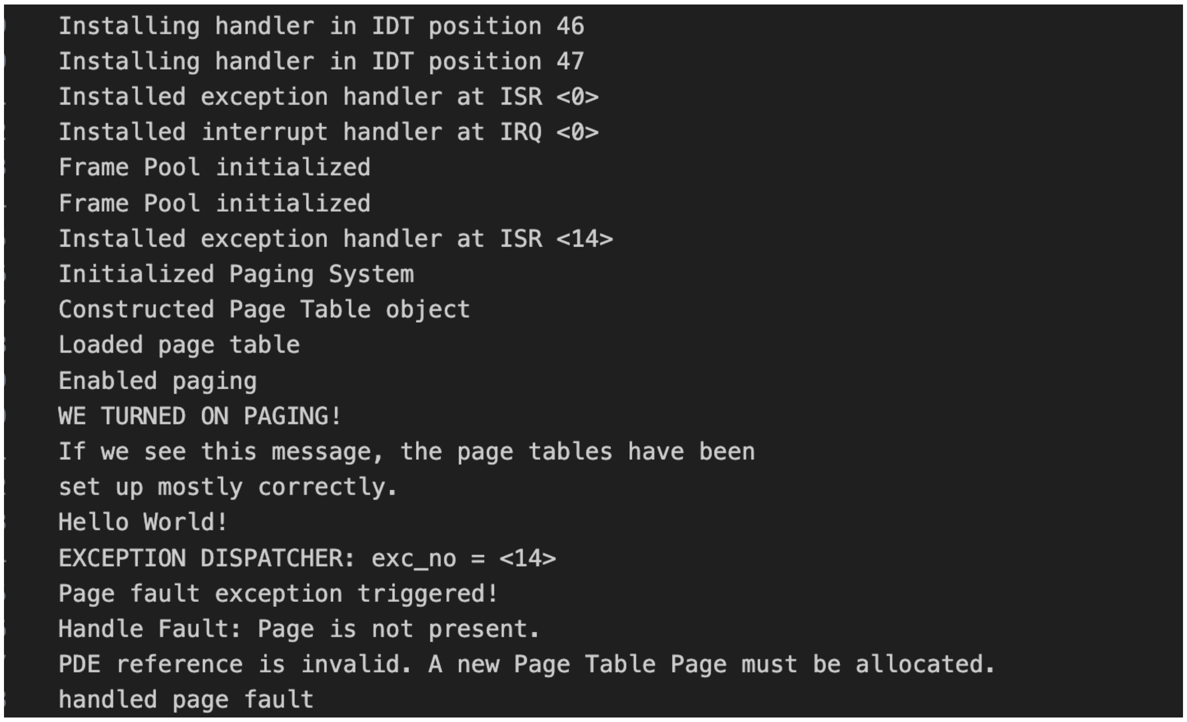
Testing was performed using the functionality provided in *kernel.C* wherein the fault address was at 4MB and requesting 1 MB of memory.



QEMU

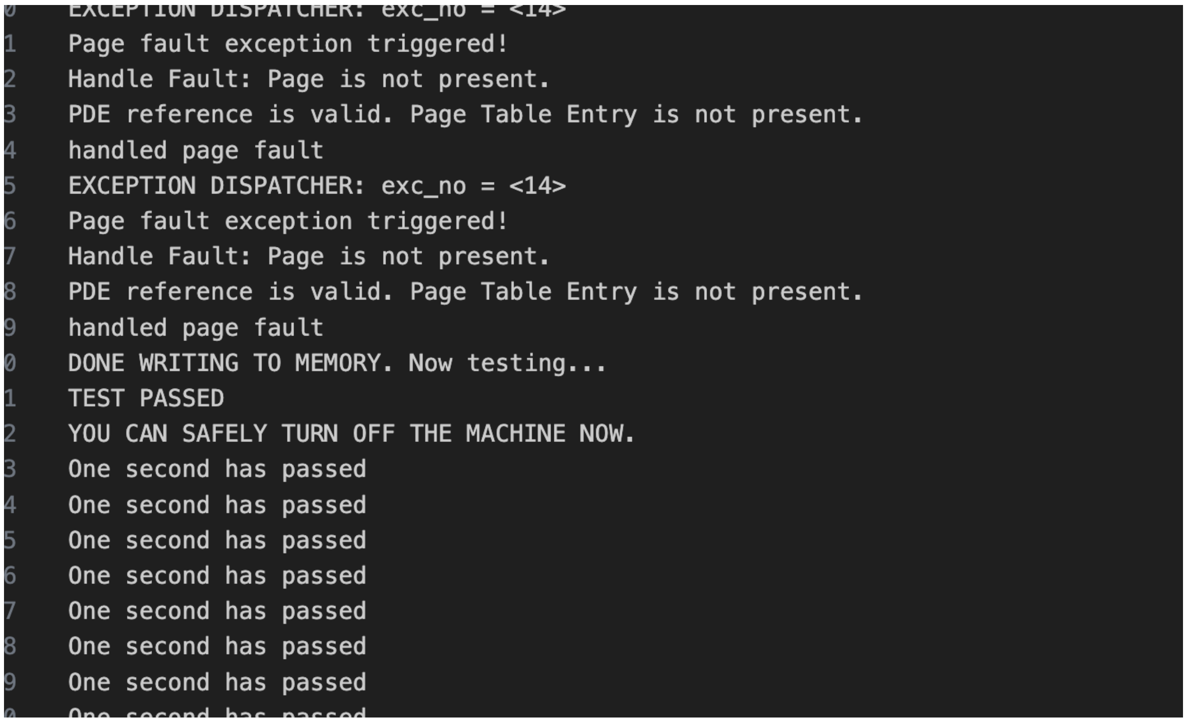
```
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
DONE WRITING TO MEMORY. Now testing...
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
```

Figure 11: QEMU output



```
Installing handler in IDT position 46
Installing handler in IDT position 47
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
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>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is invalid. A new Page Table Page must be allocated.
handled page fault
```

Figure 12: Console Logs: Paging setup

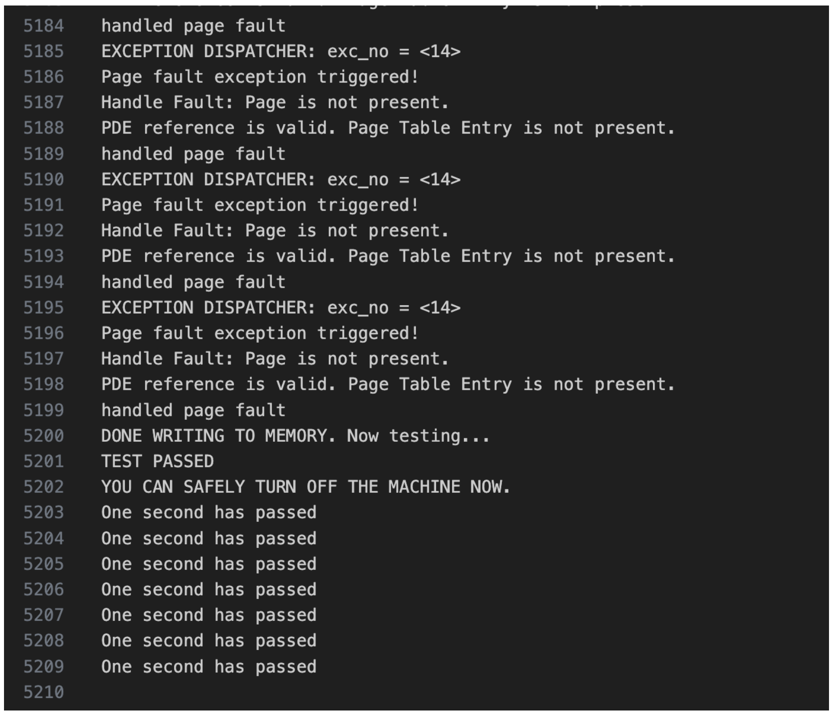


```
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
DONE WRITING TO MEMORY. Now testing...
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
One second has passed
One second has passed
One second has passed
```

Figure 13: Console Logs: Paging setup

Additional testing

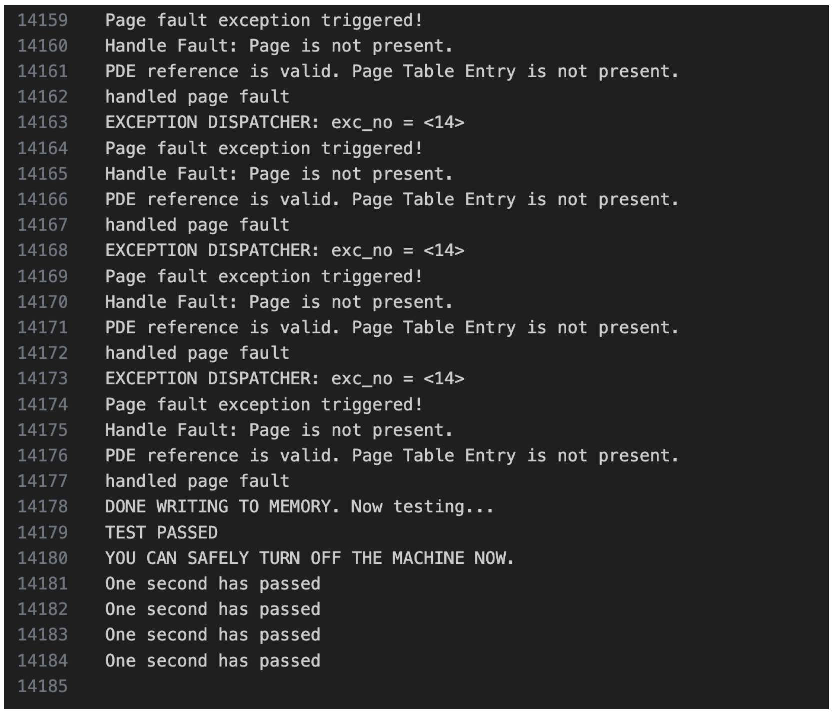
Some additional tests were performed by tweaking the macros *FAULT_ADDR* and *NACCESS*



```
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
DONE WRITING TO MEMORY. Now testing...
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
One second has passed
One second has passed
```

Figure 14: Console Logs: Fault Address = 10 MB, Request = 4 MB

1. Fault Address = 10 MB, Requested Memory = 4 MB



```
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
PDE reference is valid. Page Table Entry is not present.
handled page fault
DONE WRITING TO MEMORY. Now testing...
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
```

Figure 15: Console Logs: Fault Address = 5 MB, Request = 11 MB

2. Fault Address = 5 MB, Requested Memory = 11 MB

Limitations

The current Page Table Manager is restricted to only 1 address space. Also, swapping and virtual memory has not been implemented. Once, a frame is allocated then the memory persists and eventually the system runs out of memory if continuously allocated.