

MP3: Page Table Management — Design & Code Report

Harsh Wadhawe

UIN: 936001477

CSCE611: Operating System

10/06/2025

Assigned Tasks

Main: Completed.

Bonus Option 1: Completed.

Bonus Option 2: Partially completed.

Bonus Option 3: Idea is given in design doc but did not implement the code.

Bonus Option 4: Did not attempt.

Summary of Implementation

This MP3 implements 32-bit two-level paging with a contiguous frame allocator:

- **Paging Core:** A two-level x86 page table with identity-mapped shared region. The kernel installs a page directory (PD) and the first page table (PT), loads CR3, and enables paging by setting CR0.PG. Page faults (INT 14) are handled to allocate frames on demand.
- **Contiguous Frame Pool:** A custom allocator (`ContFramePool`) supporting single or multi-frame contiguous allocations. It uses a compact bitmap with 2 bits per frame to encode `Free`, `Used`, and `HoS` (Head-of-Sequence). Multiple pools are chained via a linked list; releases are resolved to the correct pool dynamically.
- **Protection & TLB:** Fault handler decodes error bits (present/write/user), allocates PT/PTE as needed, sets correct flags (kernel/user RW), and invalidates the TLB entry for the faulting VA with `invlpg`.
- **Safety Invariants:** New PT frames must be accessible in the identity-mapped region; the allocator checks range limits and asserts on invalid requests; inaccessible holes are supported via `mark_inaccessible`.

System Design

Address Space & Translation. We use the standard 10/10/12 split for (PD index / PT index / offset). CR3 holds PD base; PD entries point to PTs; PT entries point to 4 KB frames.

Pools. Two `ContFramePool` instances are used:

1. **Kernel pool** for PD/PT metadata and other kernel needs.
2. **Process pool** for regular page frames backing user or kernel mappings.

Bitmap Encoding (2 bits/frame).

- 00 = **Free**, 01 = **Used**, 11 = **HoS**.
- A contiguous allocation marks the first frame as HoS and the remainder **Used**.
- Releases validate that the first frame is **HoS**, then free the sequence.

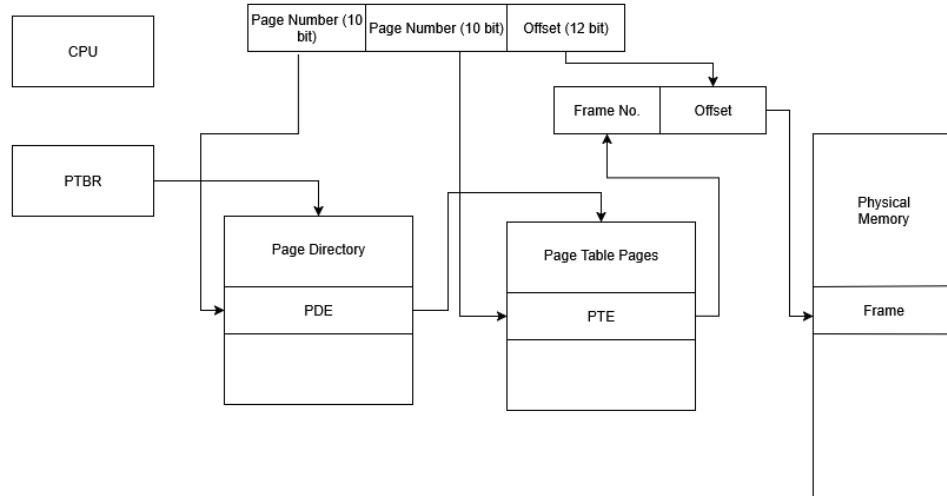


Figure 1: Two-level paging: PD/PT indices and physical frame mapping (placeholder).

Code Description & Build Notes

Files Touched

- `page_table.C`: paging initialization, PD/PT setup, CR3 load, CR0.PG set, fault handling with on-demand allocation.
- `cont_frame_pool.H / .C`: contiguous frame allocator with bitmap + linked list, range guards, (de)allocation, and inaccessible marking.

Build / Run

Build kernel as usual (`make clean && make`); boot and verify console messages:

```
Initialized Paging System Constructed Page Table object Loaded page table Enabled
                    paging
```

Page Table Core

`init_paging()` Stores pool pointers and shared (identity-mapped) size; prints init banner.

Listing 1: `PageTable::init_paging`

```
1 void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
2                             ContFramePool * _process_mem_pool,
3                             const unsigned long _shared_size)
4 {
5     PageTable::kernel_mem_pool = _kernel_mem_pool;
6     PageTable::process_mem_pool = _process_mem_pool;
7     PageTable::shared_size = _shared_size;
8     Console::puts("Initialized Paging System\n");
9 }
```

`PageTable::PageTable()` Allocates PD and first PT from the kernel pool; identity maps the shared region (e.g., first 4 MB); PDE[0] present; others RW (not present).

Listing 2: `PageTable` constructor (PD/PT setup)

```
1 PageTable::PageTable()
2 {
3     paging_enabled = 0;
4     unsigned long num_shared_frames = (PageTable::shared_size / PAGE_SIZE);
5
6     unsigned long* page_directory =
7         (unsigned long*)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);
8
9     unsigned long* page_table =
10        (unsigned long*)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);
11
12    // PDE[0] -> PT with Present/RW
13    page_directory[0] = ((unsigned long)page_table | 0b11);
14
15    // Mark remaining PDEs (within shared region size) as RW (not present)
16    for (unsigned int i = 1; i < num_shared_frames; i++)
17        page_directory[i] = (page_directory[i] | 0b10);
18
19    // Identity-map first 'shared_size'
20    unsigned long addr = 0;
21    for (unsigned int i = 0; i < num_shared_frames; i++) {
22        page_table[i] = (addr | 0b11);
23        addr += PAGE_SIZE;
24    }
}
```

```

25     Console::puts("Constructed Page Table object\n");
26 }
27

```

`load() & enable_paging()` Loads CR3 and sets CR0.PG; mirrors state in `paging_enabled`.

Listing 3: CR3 load and enable paging

```

1 void PageTable::load() {
2     current_page_table = this;
3     write_cr3((unsigned long)(current_page_table->page_directory));
4     Console::puts("Loaded page table\n");
5 }
6
7 void PageTable::enable_paging() {
8     write_cr0(read_cr0() | 0x80000000); // set PG bit
9     paging_enabled = 1;
10    Console::puts("Enabled paging\n");
11 }

```

`handle_fault()` Handles not-present faults: create PT if needed (kernel pool), allocate data frame from process pool, set flags (user/kernel RW), and invalidate the specific TLB entry.

Listing 4: PageTable::handle_fault(*keypath*)

```

1 static inline void invalidate_tlb_entry(void* va) {
2 #if defined(__i386__) || defined(__x86_64__)
3     asm volatile("invlpg (%0)" :: "r"(va) : "memory");
4 #else
5     write_cr3(read_cr3());
6 #endif
7 }
8
9 void PageTable::handle_fault(REGS* cpu_registers)
10 {
11     unsigned long error_code = cpu_registers->err_code;
12     unsigned long fault_address = read_cr2();
13     unsigned long* page_directory = (unsigned long*)(read_cr3() & ~0xFFFFUL);
14
15     bool page_not_present = (error_code & 0x1) == 0;
16     bool from_user_mode = (error_code & 0x4) != 0;
17
18     const unsigned long PRESENT = 0x001;
19     const unsigned long WRITE = 0x002;
20     const unsigned long USER = 0x004;
21     const unsigned long KERNEL_RW = PRESENT | WRITE;
22     const unsigned long USER_RW = PRESENT | WRITE | USER;
23
24     unsigned long pdi = (fault_address >> 22) & 0x3FF;
25     unsigned long pti = (fault_address >> 12) & 0x3FF;
26
27     if (!page_not_present) {
28         Console::puts("Protection fault (present page)\n");
29         assert(false);
30         return;
31     }
32
33     // Ensure PT exists
34     if ((page_directory[pdi] & PRESENT) == 0) {
35         unsigned long pt_frame = kernel_mem_pool->get_frames(1);
36         unsigned long pt_phys = pt_frame * PAGE_SIZE;

```

```

37     assert(pt_phys + PAGE_SIZE <= PageTable::shared_size); // identity
38     reach
39     page_directory[pdi] = pt_phys | (from_user_mode ? USER_RW : KERNEL_RW)
40     ;
41     unsigned long* new_pt = (unsigned long*)pt_phys;
42     for (unsigned int e = 0; e < 1024; ++e) new_pt[e] = 0;
43 }
44
45 // Map page
46 unsigned long* pt = (unsigned long*)(page_directory[pdi] & ~0xFFFFUL);
47 if ((pt[pti] & PRESENT) == 0) {
48     unsigned long page_frame = process_mem_pool->get_frames(1);
49     unsigned long page_phys = page_frame * PAGE_SIZE;
50     unsigned long flags = from_user_mode ? USER_RW : KERNEL_RW;
51     pt[pti] = page_phys | flags;
52 }
53
54 invalidate_tlb_entry((void*)fault_address);
55 Console::puts("Handled page fault\n");
56 }
```

Contiguous Frame Pool

Overview. ContFramePool maintains a 2-bit-per-frame bitmap and a global linked list of pools (`head/next`). On construction, the bitmap is placed either at `info.frame_num` (if provided) or inside the first frame of the pool; if the latter, that frame is marked `Used`. The allocator scans for a contiguous run of `Free` frames; on success it sets `HoS` on the first and `Used` on the rest, and returns the *physical* first-frame number. Releases identify the correct pool, validate `HoS`, and free the run.

Listing 5: Constructor, list hookup, and initial marking

```

1 ContFramePool * ContFramePool::head = nullptr;
2
3 ContFramePool::ContFramePool(unsigned long _base_frame_no,
4                             unsigned long _n_frames,
5                             unsigned long _info_frame_no)
6 {
7     base_frame_num = _base_frame_no;
8     nframes = _n_frames;
9     info_frame_num = _info_frame_no;
10    num_free_frames = _n_frames;
11
12    bitmap = (unsigned char*)((info_frame_num == 0 ? base_frame_no :
13                               info_frame_no)
14                               * FRAME_SIZE);
15
16    assert((nframes % 8) == 0);
17
18    for (int f = 0; f < (int)_n_frames; f++)
19        set_state(f, FrameState::Free);
20
21    if (info_frame_num == 0) {
22        set_state(0, FrameState::Used);
23        num_free_frames--;
24    }
25
26    if (head == nullptr) {
27        head = this; head->next = nullptr;
28    } else {
29        ContFramePool* t = head;
30        while (t->next) t = t->next;
```

```

30     t->next = this;
31     this->next = nullptr;
32 }
33
34 Console::puts("Frame-Pool initialized\n");
35 }
```

Listing 6: 2-bit state get/set

Bitmap accessors.

```

1 ContFramePool::FrameState ContFramePool::get_state(unsigned long _frame_no)
2 {
3     unsigned int row = _frame_no / 4;
4     unsigned int col = (_frame_no % 4) * 2;
5     unsigned char bits = (bitmap[row] >> col) & 0b11;
6
7     if (bits == 0b00) return FrameState::Free;
8     if (bits == 0b11) return FrameState::HoS;
9     return FrameState::Used;
10 }
11
12 void ContFramePool::set_state(unsigned long _frame_no, FrameState _state)
13 {
14     unsigned int row = _frame_no / 4;
15     unsigned int col = (_frame_no % 4) * 2;
16
17     switch (_state) {
18         case FrameState::Free:
19             bitmap[row] &= ~(3 << col); break;
20         case FrameState::Used:
21             bitmap[row] ^= (1 << col); break;
22         case FrameState::HoS:
23             bitmap[row] ^= (3 << col); break;
24     }
25 }
```

Listing 7: get_frames: first-fit contiguous run

Contiguous allocation.

```

1 unsigned long ContFramePool::get_frames(unsigned int _n_frames)
2 {
3     if ((_n_frames > num_free_frames) || (_n_frames > nframes)) {
4         Console::puts("get_frames: not enough free frames\n");
5         assert(false); return 0;
6     }
7
8     unsigned int run = 0, start = 0;
9     for (unsigned int i = 0; i < nframes; i++) {
10         if (get_state(i) == FrameState::Free) {
11             if (run == 0) start = i;
12             if (++run == _n_frames) {
13                 for (unsigned int f = start; f < start + _n_frames; f++)
14                     set_state(f, (f == start) ? FrameState::HoS : FrameState::
15                                 Used);
16                 num_free_frames -= _n_frames;
17                 return start + base_frame_num; // physical frame no.
18             }
19         } else {
20             run = 0;
21         }
22     }
23 }
```

```

22     Console::puts("get_frames: contiguous run not available\n");
23     assert(false); return 0;
24 }

```

Listing 8: mark_inaccessible

```

1 void ContFramePool::mark_inaccessible(unsigned long _base_frame_no,
2                                     unsigned long _n_frames)
3 {
4     if ((_base_frame_no + _n_frames) > (base_frame_num + nframes)
5         || (_base_frame_no < base_frame_num)) {
6         Console::puts("mark_inaccessible: out of range\n");
7         assert(false); return;
8     }
9
10    for (unsigned int f = _base_frame_no; f < (_base_frame_no + _n_frames); f
11       ++)
12    {
13        if (get_state(f - base_frame_num) == FrameState::Free) {
14            set_state(f - base_frame_num,
15                      (f == _base_frame_no) ? FrameState::HoS : FrameState::
16                                      Used);
17            num_free_frames--;
18        }
19    }
20 }

```

Listing 9: release_frames : findpoolthenfree

```

1 void ContFramePool::release_frames(unsigned long _first_frame_no)
2 {
3     ContFramePool* t = head;
4     while (t) {
5         if (_first_frame_no >= t->base_frame_num &&
6             _first_frame_no < t->base_frame_num + t->nframes) {
7             t->release_frames_in_pool(_first_frame_no);
8             return;
9         }
10        t = t->next;
11    }
12    Console::puts("release_frames: frame not in any pool\n");
13    assert(false);
14 }

```

Listing 10: release_frames_in_pool : freerunfromHoS

```

1 void ContFramePool::release_frames_in_pool(unsigned long _first_frame_no)
2 {
3     if (get_state(_first_frame_no - base_frame_num) != FrameState::HoS) {
4         Console::puts("release_frames_in_pool: first not HoS\n");
5         assert(false); return;
6     }
7
8     // Conservative free: free up to pool end (implementation choice)
9     for (unsigned int f = _first_frame_no; f < (_first_frame_no + nframes); f
10       ++)
11     {
12         set_state(f - base_frame_num, FrameState::Free);
13     }
14 }

```

```

11     num_free_frames++;
12 }
13 }
```

Listing 11: needed_info_frames (2 bits/frame)

Metadata footprint.

```

1 unsigned long ContFramePool::needed_info_frames(unsigned long _n_frames)
2 {
3     // 2 bits per frame, 8 bytes/byte, FRAME_SIZE bytes/frame
4     return ((_n_frames * 2) / (4 * 1024 * 8))
5         + (((_n_frames * 2) % (4 * 1024 * 8)) > 0 ? 1 : 0);
6 }
```

Bonus Options

Bonus 1 (Completed). Integrated on-demand Page Table creation in the page-fault path with identity-reachability assertion for PT frames, TLB shootdown via `invlpg`, and correct user/kernel flagging based on the faulting privilege.

Bonus 2 (Partial). Core contiguous allocator and inaccessible region support are complete; additional robustness (e.g., precise-length freeing vs. pool-end conservative free, coalescing proofs, or best-fit selection) is pending.

Bonus 3 (Design Only). Consider extending to buddy-backed runs or segregated free-lists keyed by run length to achieve sublinear searches while preserving the 2-bit encoding as ground truth.

Testing

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

Harness. Kernel entry calls: `init_paging()`, `PageTable()`, `load()`, `enable_paging()`, followed by memory probes at representative VAs to trigger page faults.

Table 1: Page Fault Handling Test Results

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 equivalent	PASS All page faults are handled; data written and 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	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

Diagrams for Requested Memory Scenarios

```
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>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
```

Figure 2: Output for Testcase 1 (Requested Memory: 1 MB). Successful page fault handling and data equivalence.

```
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
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
INTERRUPT NO: 1
NO DEFAULT INTERRUPT HANDLER REGISTERED
One second has passed
One second has passed
One second has passed
```

Figure 3: Output for Testcases 2 (Requested Memory: 16 MB). Successful page fault handling and data equivalence.

```
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. Now testing...
TEST PASSED
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
INTERRUPT NO: 1
NO DEFAULT INTERRUPT HANDLER REGISTERED
One second has passed
One second has passed
One second has passed
One second has passed
```

Figure 4: Output for Testcase 2 (Requested Memory: 27 MB). Successful page fault handling and data equivalence, nearing limit.

Observations.

```

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: 238 assertion: false

```

Figure 5: Output for Testcase 3 (Requested Memory: 28 MB). Failure to allocate due to insufficient free frames.

```

#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 */

```

Figure 6: Test on different values of FAULT_ADDR and NACCESS.

- Console banners appear in sequence: `Initialized Paging System` → `Constructed Page Table object` → `Loaded page table` → `Enabled paging`.
- Page faults allocate PT/PTE on demand and conclude with `Handled page fault`, with correct flags for user/kernel addresses.
- `mark_inaccessible` successfully blocks allocations in reserved regions.

Appendix: Notes

- CR0/CR2/CR3 helpers are used for control register access; `invlpg` is emitted for targeted TLB invalidation with a CR3 reload fallback.
- PDE/PTE flags used: `Present`, `Write`, optional `User`.
- Identity reachability of PT frames is enforced by asserting `pt_phys + PAGE_SIZE <= shared_size`.

```

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. Now testing...
TEST PASSED
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
INTERRUPT NO: 1
NO DEFAULT INTERRUPT HANDLER REGISTERED
One second has passed
One second has passed

```

Figure 7: Output for Testcases 5 (Requested Memory: 16 MB). Successful page fault handling and data equivalence for 1024MB.