# Bare Metal OS Project Documentation

## Overview

This document provides a comprehensive analysis of a bare metal operating system implementation written in Rust. The OS runs directly on hardware without an underlying operating system, implementing a simple Pong game to demonstrate core functionality.

## Project Structure

```
/home/shane/CMKL/os/sys101-s25-baremetal/
├── kernel/
│   └── src/
│       ├── main.rs       # Kernel entry point
│       ├── screen.rs     # Display management
│       ├── allocator.rs  # Memory allocation system
│       ├── frame_allocator.rs # Physical memory management
│       ├── interrupts.rs # Interrupt handling
│       ├── gdt.rs        # Global Descriptor Table setup
│       └── pong.rs       # Game implementation
```

## File Summaries

### 1. main.rs

Purpose: Kernel entry point and primary initialization

Key Components:

- Boot configuration (256 KiB kernel stack, dynamic memory mapping)
- Framebuffer initialization
- Memory mapping and allocation
- GDT initialization
- Interrupt handling setup
- Game initialization
- Input processing

Key Functions:

- kernel_main(): Main entry point that initializes system components
- start(): Displays welcome message
- tick(): Game update function called by timer interrupts
- key(): Keyboard input handler

Code Highlights:

```
rust
const BOOTLOADER_CONFIG: BootloaderConfig = {
    let mut config = BootloaderConfig::new_default();
    config.mappings.physical_memory = Some(Dynamic);
    config.kernel_stack_size = 256 * 1024; // 256 KiB
    config
};

// Kernel initialization
fn kernel_main(boot_info: &'static mut BootInfo) -> ! {
    // Initialize display
    screen::init(framebuffer);

    // Memory management
    allocator::init_heap((physical_offset + usable_region.start) as usize);
    let mut mapper = frame_allocator::init(VirtAddr::new(physical_offset));

    // Initialize system components
    gdt::init();
    pong::init_game();

    // Start interrupt handling
    let lapic_ptr = interrupts::init_apic(...);
    HandlerTable::new()
        .keyboard(key)
        .timer(tick)
        .startup(start)
        .start(lapic_ptr)
}
```

## 2. screen.rs

Purpose: Display management and rendering

Key Functions:

- init(): Initialize framebuffer
- screenwriter(): Access the screen writer singleton
- draw_pixel(): Draw pixels with RGB values
- Text rendering capabilities

## 3. allocator.rs

Purpose: Memory allocation management

Key Functions:

- init_heap(): Initialize the heap allocator
- Allocation and deallocation functionality

### 4. frame_allocator.rs

**Purpose: Physical memory frame allocation**

**Key Functions:**

- **init(): Initialize the frame allocator**
- **BootInfoFrameAllocator: Manage physical memory frames**

### 5. interrupts.rs

**Purpose: Interrupt handling system**

**Key Functions:**

- **init_apic(): Initialize Advanced Programmable Interrupt Controller**
- **Interrupt handler setup for keyboard and timer**

### 6. gdt.rs

**Purpose: Global Descriptor Table setup**

**Key Functions:**

- **init(): Setup memory segmentation**

### 7. pong.rs

**Purpose: Game implementation**

**Key Functions:**

- **init_game(): Initialize game state**
- **update_game(): Update game logic on timer ticks**
- **set_key_w(), set_key_s(): Handle paddle movement**
- **start_game(): Start game functionality**

## System Architecture

### Boot Process

- **Bootloader loads kernel**
- **kernel_main() initializes system**
- **Memory regions mapped**
- **Graphics initialized**
- **Interrupt handling started**

### Memory Model

- **Dynamic physical memory mapping**
- **Custom heap allocator**
- **Frame-based physical memory management**
- **Global Descriptor Table for memory segmentation**

### Input System

- Interrupt-driven keyboard handling
- Support for both Unicode and raw keycodes
- Game controls via W/S keys

### Game Loop

- Timer interrupts trigger tick() function
- Game state updated
- Screen redrawn
- Keyboard input processed asynchronously

## Key Implementation Details

### Input Processing

```
rust
fn key(key: DecodedKey) {
    match key {
        DecodedKey::Unicode(character) => {
            match character {
                'w' => pong::set_key_w(true),
                's' => pong::set_key_s(true),
                ' ' => pong::start_game(),
                'q' => {
                    pong::set_key_w(false);
                    pong::set_key_s(false);
                },
                _ => write!(Writer, "{}", character).unwrap(),
            }
        },
        DecodedKey::RawKey(key) => {
            match key {
                KeyCode::W => pong::set_key_w(true),
                KeyCode::S => pong::set_key_s(true),
                _ => write!(Writer, "{:?}", key).unwrap(),
            }
        }
    }
}
```

### Graphics Initialization

```
rust
let frame_info = boot_info.framebuffer.as_ref().unwrap().info();
let framebuffer = boot_info.framebuffer.as_mut().unwrap();
screen::init(framebuffer);
for x in 0..frame_info.width {
    screenwriter().draw_pixel(x, frame_info.height-15, 0xff, 0, 0);
    screenwriter().draw_pixel(x, frame_info.height-10, 0, 0xff, 0);
    screenwriter().draw_pixel(x, frame_info.height-5, 0, 0, 0xff);
}
```

## Conclusion

This bare metal OS implementation demonstrates fundamental operating system concepts using Rust's safety features. The project successfully implements:

- Memory management
- Hardware interaction
- Interrupt handling
- Real-time game processing
- Graphics rendering

The architecture balances simplicity with functionality, providing a solid foundation for bare metal development.