Building a NASM and C-based 32-Bit Protected Mode Kernel for Tetris

NEA PROJECT
By Sam kierdelewicz

Contents

Analysis

Introduction

Welcome to the document's introduction on a NASM and C-based 32-bit protected mode kernel that runs Tetris. This project exemplifies the strength and adaptability of low-level programming languages and their capacity to produce sophisticated applications. A timeless example of a straightforward but difficult game is the Tetris game.
The project consists of two primary parts: a NASM-written boot loader and a C-written kernel. The boot loader loads the kernel into memory, which also initiates the Tetris game's execution. The C-written kernel handles the game logic, graphics rendering, and user input. This project aims to showcase low-level programming languages' strengths and capacity to enable bare-metal hardware applications. For individuals interested in game creation and low-level programming, the project also acts as a learning tool.
Overall, this project exemplifies the strength and adaptability of low-level programming languages and their capacity to produce sophisticated applications like the well-known Tetris game. I want to encourage others to learn more about game creation and low-level programming by sharing this project.

Background

How kernel works:
The core component of an operating system, known as the kernel, controls the system's resources and offers services to user programmes. It is in charge of managing memory, scheduling jobs, and controlling the hardware. Device drivers, which offer a standardised interface to the hardware, are used by the kernel to interact with the hardware.
You must have a solid grasp of the computer's hardware specifications and those for the operating system to design a kernel. A stable environment for user applications to execute must be provided by the kernel, which must be able to interface with the hardware. Setting up and reading the disk, configuring the global descriptor table, converting from 16 to 32-bit mode, and configuring the kernel itself are all steps in constructing a kernel.

Setting up the disk:
Setting up and reading the disk is the first stage in the kernel-building process. Before the kernel can begin operating, the disk must be loaded into memory. The bootloader, a compact programme that launches the kernel's execution after loading it into memory, is located on the disk. The boot sector, commonly known as the disk's first sector, is where the bootloader normally sits.

The bootloader transfers control to the kernel's entry point after reading the kernel image from the disk into memory. The kernel then continues the boot procedure.

Global Descriptor Table:
The CPU uses the Global Descriptor Table (GDT) as a data structure to associate memory chunks with logical addresses. The GDT has entries that specify the characteristics of memory segments, including their size, access privileges, and level.

The kernel must configure the GDT before it may begin operating. The GDT data structure must first be created before entries defining the memory regions used by the kernel and user applications can be added to it. The GDTR register in the CPU refers to the GDT, which is normally stored in memory at a defined address.

Switching different bit modes:
The CPU has different operating modes 16-bit, 32-bit and 64-bit. The processor's memory access is constrained to a small portion in 16-bit mode, and instructions are only allowed to be 16 bits long. The processor may access more memory when operating in 32-bit mode, and instructions are 32 bits wide in this mode.

To make use of the more RAM and 32-bit instructions, the kernel must force the processor into 32-bit mode. In order to do this, the segment registers and control registers of the CPU must be configured to support 32-bit mode.

Interrupt Descriptor Table (IDT):
The processor uses a data structure called the Interrupt Descriptor Table (IDT) to manage interrupts. Signals are known as interrupts are produced by hardware or software and force the processor to pause its ongoing work to respond to the interrupt. The IDT includes elements that provide interrupt handler attributes, including memory location, permission level, and handling method.

To handle interruptions produced by hardware components and system internal operations, the kernel must configure the IDT. The interrupt handlers and their properties are defined by establishing the IDT data structure and adding entries. The IDTR register of the CPU refers to the IDT, which is normally stored in memory at a set address.

Interrupt Service Routines (ISR):
The processor's Interrupt Service Routines (ISRs) are actions taken in response to an interrupt signal. The ISR for each interrupt is in charge of processing the interrupt and carrying out any necessary operations.

For the interruptions it manages, the kernel must implement the ISRs. Writing code is required to handle the interrupt and return the system to its initial state. To carry out their functions, the ISRs frequently interface with the kernel's data structures, such as the process scheduler and memory management.

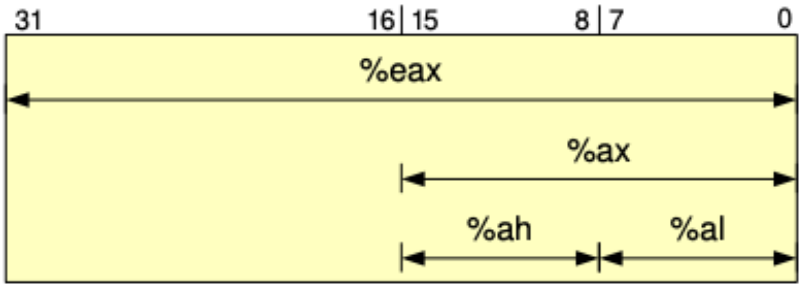
Interrupt Requests (IQR):

Hardware devices send interrupt requests (IRQs) signals to the processor to demand attention. A hardware component that needs to communicate with the CPU will produce an IRQ, a sort of interrupt. Events like disk I/O, network activity, and user input from things like keyboards and mice are handled by IRQs.

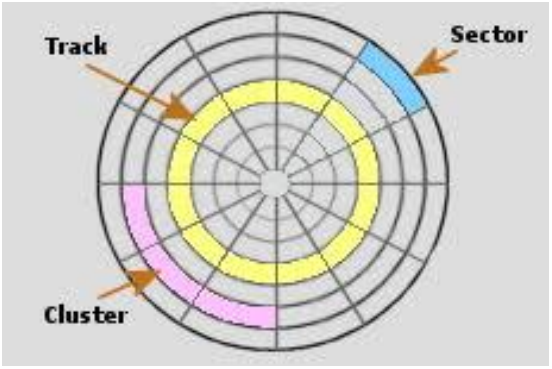
The system gives each IRQ a unique number used to identify it. The system timer, for instance, is commonly assigned IRQ0, and the keyboard controller, IRQ1. An individual IRQ number is given to each interrupt-generating piece of hardware.

To guarantee that each interrupt is handled appropriately and effectively, the kernel must manage IRQs. This entails registering interrupt handlers for each IRQ and configuring the interrupt controller. A hardware component known as the interrupt controller controls the flow of interrupts between the CPU and hardware components. The kernel communicates with the interrupt controller to receive and manage interrupts from hardware devices.

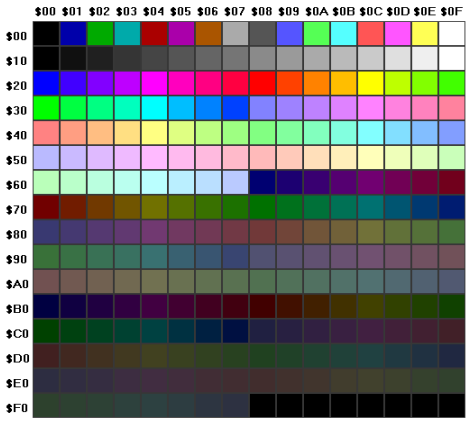
How registers are represented:



How disks are represented:



VGA colour palette:



The target Audience

Tech enthusiasts: People interested in technology and enjoy experimenting with different operating systems and software may be interested in trying out the kernel.

Game enthusiasts: People who enjoy playing Tetris and are looking for a new and unique way to experience the game may be interested in a kernel specifically made for Tetris that offers advanced features and customisation options.

Developers: Developers interested in operating systems and kernel development may be interested in experimenting with a kernel for Tetris as a learning tool or a platform for developing their games.

Retro enthusiasts: People who like the retro style of games and enjoy playing games similar to the classic games from the past may be interested in a kernel that runs Tetris that offers a retro gameplay style.

Low-level programmers: People interested in programming at a low level and in operating systems, kernels, and hardware may be interested in trying a kernel made for a game to explore these topics.

Interview

Interview with Emily Smith, a software engineer at Intel and video game enthusiast:

The following is an interview with Emily Smith, an enthusiastic video game player and software engineer at Intel. In this interview, we talk about her opinions on a kernel which runs Tetris and how it might be useful for both amusement and education. We examine the qualities that would make a fantastic kernel with Tetris, such as the core gameplay, the visual appeal, and potential use cases.

Q: What aspects of the gameplay do you think are most important?

Emily Smith:
The most important aspect is the ability to rotate and move the Tetrominoes smoothly without glitches. The game should also have responsive controls so the player can move the pieces quickly and accurately.

Q: Do you believe that a kernel for Tetris like this could be useful in any practical applications, or is it mainly for entertainment?

Emily Smith:

While it may not have practical applications, having fun and entertaining projects like this is important. It can also be a great learning tool for beginner programmers interested in operating system development.

Q: What are some acceptable limitations of this program?

Emily Smith:
As a simple kernel, some acceptable limitations could be the lack of advanced features such as a scoring system or sound effects. Additionally, the graphics may be limited to basic shapes and colours rather than high-resolution images.

Q: Do you believe that a kernel like this could be used as a learning tool for beginner programmers interested in operating system development?

Emily Smith:
A kernel for Tetris like this could be a useful learning tool for beginner programmers interested in operating system development. It could help them understand the basics of developing a kernel, including writing code in assembly and C, managing memory, and interacting with hardware.

Q: In your opinion, what would be an acceptable level of graphical quality for a kernel made to run Tetris like this?

Emily Smith:
A basic level of graphical quality would be acceptable for a simple kernel designed to run Tetris like this. The graphics could consist of simple shapes such as squares, rectangles, and circles, and the colours could be limited to a basic palette. However, the graphics should still be clear and easily distinguishable to allow for smooth gameplay.

Acceptable limitations

Although my kernel that runs Tetris has successfully implemented the game's basic mechanics, it has some limitations that could impact the user experience. One of these limitations is the need for a scoring system. While the game can still be enjoyable without keeping score, a scoring system could provide players with a sense of achievement and motivation to improve their gameplay. Future iterations of the kernel could include a scoring system to enhance the game's competitiveness and replay value. To add a scoring system, I would first research different scoring methods used in Tetris games and choose one that aligns with my game's mechanics. Then, I would add code to track the number of lines cleared and calculate the score accordingly. Finally, I would integrate the scoring system into the game's user interface.

Another area for improvement of the kernel is the need for sound. Sound effects such as music and sound effects can add an extra layer of immersion and enjoyment to the game. With sound, the gameplay experience may feel comfortable and engaging. Adding sound effects would be an improvement that could enhance the overall user experience of the

game. I would combine frequency buzzes for each sound effect to make clear audio for the user.

Finally, it's important to note that the current kernel has been coded for macOS C compilers. This means that it may not be compatible with other operating systems. This limitation could prevent potential users from enjoying the game using a different operating system. Future iterations of the kernel could be made more widely accessible by making it compatible with multiple operating systems. If I were to implement the code running for another operating system, I would need to change the compilers in the make file, which will be able to create binary files on other systems.

System requirements

To run the kernel, your system will need to meet some requirements and install some packages:

1) '/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"': This command installs Homebrew, which is a package manager for macOS that simplifies the process of installing and managing software.
2) 'brew install i386-elf-binutils': This command installs the i386-elf-binutils package, which is a collection of binary utilities for the i386-elf target, including programs like "as" (the GNU Assembler), "ld" (the GNU linker), "objdump" (an object file disassembler), and others.
3) 'brew install i386-elf-gcc': This command installs the i386-elf-gcc package, which is a compiler for the i386-elf target that can be used to compile C, C++, and other languages to produce machine code that runs on the i386 architecture.
4) 'brew install i386-elf-gdb': This command installs the i386-elf-gdb package, which is a debugger for the i386-elf target that allows developers to debug their programs by examining their memory, setting breakpoints, and stepping through code.
5) 'brew install qemu nasm': These commands install the qemu and nasm packages. QEMU is a virtualization tool that can be used to emulate a computer system, while NASM is an assembler that can be used to convert assembly language code into machine code.
6) 'make run': To run the kernel and compile all the code.

Objectives

The project's main goal is to create a unique operating system kernel that enables Tetris to operate in 32-bit protected mode on a computer. With interactive input handling and visual presentation in video mode, the kernel should offer a fluid and seamless gaming experience. Additionally, it must incorporate crucial features like a game engine, a scoring system, and other crucial game logic. The overall objective is to use a customised operating system kernel to produce a fully functional Tetris game experience.

Here are some detailed objectives of how I could approach this:

1. Have a kernel that can run in 32-bit protected mode on a PC, with a custom bootloader.

a) Read from disk, which involves reading the bootloader from the disk into memory and transferring control to it.

i) Read the bootloader data from disk: This objective involves reading the bootloader data from the storage device where it is stored and storing it in memory in a location accessible to the CPU. The bootloader data must be validated to ensure its integrity and to detect any errors or corruption that may have occurred during the storage process.

ii) Implement error handling and recovery mechanisms: This objective involves developing strategies to handle errors that may occur while reading the bootloader data from the disk. This includes detecting errors, implementing error handling procedures to mitigate the impact of errors, and developing recovery mechanisms to allow the system to continue the boot process even if the bootloader data is damaged or corrupted. This will help to ensure that the boot process is robust and can continue even in the face of potential issues.

b) Set up a GDT to allow the system to run in protected mode.

i) Define the structure of the Global Descriptor Table (GDT): This objective involves defining the layout and properties of the memory segments used by the CPU in the GDT. The GDT must accurately reflect the different memory segments used by the system and support efficient and secure system operation.

ii) Initialise the GDT: This objective involves making the defined GDT accessible to the CPU and setting it up for use in managing memory access and enforcing protection levels. This will involve writing the necessary code and executing it to initialise the GDT, ensuring that it is properly set up and functioning as intended. The goal is to ensure that the CPU can use the GDT to manage memory access and enforce protection levels efficiently, contributing to the secure and efficient operation of the system.

c) Switch to 32-bit protected mode.

i) Disable Interrupts: This objective ensures a smooth transition to protected mode by disabling any potential interruptions.

ii) Load a GDT with a flat memory model: This objective prepares the system for protected mode by loading a flat memory model, which is necessary for memory management and protection.

iii) Set the protected mode bit in the Control Register 0 (CR0): This objective signals the CPU to switch to protected mode by setting the protected mode bit in the CR0 register.
iv) Jump to protected mode: This objective transfers control to protected mode, enabling the system to operate in a secure mode of operation.

d) Write the Master Boot Record.

i) Locate the operating system: The MBR will use information stored in the partition table to locate the operating system and load it into memory.
ii) Store the boot drive information: The code will store the boot drive information, so that it can be used later to load the game data.
iii) Provide a stable platform: The code will set up the stack and switch to 32-bit protected mode, creating a stable platform for the game to run on.

e) Create a kernel entry point.

i) Establish a starting point for the kernel: The code will define a clear entry point for the kernel, which will be used as the starting point for the kernel's execution.

2. Implement input handling for the user with the keyboard. Ensure the inputs are processed and dealt with correctly.

a) Set up the IDT (Interrupt Descriptor Table) to handle keyboard interrupts.

i) Allocate and initialise the Interrupt Descriptor Table (IDT): The IDT will be set up in memory, with sufficient space allocated to accommodate all of the interrupt vectors. The IDT will be initialised with default values, if required.
ii) Define the structure of an IDT entry to describe the handler routine for each interrupt: The IDT entry structure will be defined, with fields for the handler routine address, segment selector, flags, and other necessary information.
iii) Register the keyboard interrupt handler routine in the IDT: The keyboard interrupts handler routine will be registered in the IDT, mapping the interrupt vector for the keyboard to the handler routine address.
iv) Configure the Programmable Interrupt Controller (PIC) to send keyboard interrupts to the processor: The PIC will be configured to send keyboard interrupts to the processor, ensuring that they are properly handled.
v) Ensure the IDT is properly loaded and used by the processor to handle keyboard interrupts: The IDT will be loaded into the processor, and the processor will be configured to use the IDT to handle interrupts. This will ensure that keyboard interrupts are properly handled.
vi) Test the IDT to confirm that it correctly handles keyboard interrupts and routes them to the appropriate handler routine: The IDT will be tested to confirm that it correctly handles keyboard interrupts and routes them to the appropriate handler routine. This will ensure that the keyboard input is processed and dealt with correctly.

b) Write the ISR (Interrupt Service Routine) for the keyboard interrupt to process the incoming scan codes.

i) Define the keyboard interrupt vector in the Interrupt Descriptor Table (IDT): This objective will ensure that the ISR for the keyboard interrupt is assigned a unique identifier or vector in the IDT, allowing the processor to locate it when a keyboard interrupt occurs.

ii) Create the Interrupt Service Routine (ISR) for the keyboard interrupt: This objective will require the creation of a subroutine that will be executed whenever a keyboard interrupt occurs.
iii) Extract the scan codes from the keyboard interrupt signal: This objective will require the implementation of a mechanism to extract the data that is sent by the keyboard during an interrupt, which will contain information about which keys have been pressed or released.
iv) Implement the logic to interpret the extracted scan codes and translate them into meaningful inputs: This objective will require the implementation of a mechanism to process the extracted scan codes and translate them into meaningful inputs that the system, such as the movements of the Tetris pieces can use.
v) Update the system state based on the processed keyboard inputs: This objective will require the implementation of a mechanism to update the system's state based on the processed keyboard inputs, such as moving the Tetris pieces in response to the user's inputs.
vi) Restore the previous context after the ISR is complete: This objective will require the implementation of a mechanism to restore the processor's state after the ISR has been executed, allowing the processor to continue its normal operation.

c) Map the scan codes to the corresponding keyboard keys using an IRQ (Interrupt Request) handler.
i) Determine IRQ handler structure: The code will outline the structure of the IRQ handler that maps scan codes to keyboard keys.
ii) Implement IRQ handler: The code will implement the IRQ handler to process scan codes and map to keyboard keys.
iii) Verify IRQ handler correctness: The code will test the IRQ handler to confirm it maps scan codes correctly to keyboard keys.
iv) Integrate IRQ handler with input handling system: The code will ensure the IRQ handler integrates correctly with the rest of the keyboard input handling system.

d) Implement a technique to take keyboard inputs from the buffer and process them according to the game's rules.

3. Create a visually appealing and easy-to-use user interface.

a) Determine the design requirements, such as essential design elements and colour palettes.
i) Conduct research: Research best practices and trends in user interface design and explore how design elements and colour palettes have been used in other Tetris games.
ii) Consider game mechanics: Evaluate how design elements and colour palettes can enhance the game mechanics, such as optimising the design of blocks to improve gameplay.

b) Design the user interface layout and create visually appealing interactive elements.
i) Establish the UI design specifications: Compile data on user needs and expectations to establish the essential components and design of the user interface.

ii) Create preliminary sketches of the interface layout to describe where interactive elements like buttons, menus, and text boxes should be placed. Make drawings of the interface to give a more accurate portrayal.
iii) Pick a colour scheme and font: To build an aesthetically pleasing interface that is simple to read and use, pick the right colour scheme and font.
iv) Integrate user interface design with software application: Integrate the user interface design with the software programme, making sure that all interactive features are responsive and functioning.

4. Develop a kernel for Tetris that implements the basic mechanics of the game
a) Define the game mechanics
i) Research rules and mechanics: Research how blocks fall, rotate and how they clear lines
ii) Use correct algorithms: Using the correct mathematical operations to make a piece rotate correctly
b) Develop the game for the kernel
i) Design the kernel architecture: Plan the structure of the kernel and decide on the modules and functions needed to implement the game mechanics.
ii) Write the kernel code: Write the code for the modules and functions outlined in the kernel architecture.
iii) Test and debug the kernel: Test the kernel to ensure it implements the Tetris game mechanics correctly and debug any errors.
c) Test and refine the game
i) Conduct usability testing: Test the game with real users to identify any usability issues, user preferences, and areas for improvement.
ii) Collect feedback: Collect feedback from users through surveys, feedback forms, and customer support channels to gain insight into user needs and expectations.
iii) Refine and optimize the game: Use insights gained from testing, feedback, and analytics to refine and optimize the game mechanics and user interface.

NASM intel x86 Assembly language instruction sheet

NASM Intel x86 Assembly Language Cheat Sheet

Instruction	Effect	Examples
Copying Data		
mov *dest,src*	Copy src to dest	mov eax,10 mov eax,[2000]
Arithmetic		
add *dest,src*	dest = dest + src	add esi,10
sub *dest,src*	dest = dest – src	sub eax, ebx
mul *reg*	edx:eax = eax * reg	mul esi
div *reg*	edx = edx:eax **mod** reg eax = edx:eax ÷ reg	div edi
inc *dest*	Increment destination	inc eax
dec *dest*	Decrement destination	dec word [0x1000]
Function Calls		
call *label*	Push eip, transfer control	call format_disk
ret	Pop eip and return	ret
push *item*	Push item (constant or register) to stack. I.e.: esp=esp-4; memory[esp] = item	push dword 32 push eax
pop *[reg]*	Pop item from stack and store to register I.e.: reg=memory[esp]; esp=esp+4	pop eax
Bitwise Operations		
and *dest, src*	dest = src & dest	and ebx, eax
or *dest,src*	dest = src | dest	or eax,[0x2000]
xor *dest, src*	dest = src ^ dest	xor ebx, 0xffffffff
shl *dest,count*	dest = dest << count	shl eax, 2
shr *dest,count*	dest = dest >> count	shr dword [eax],4
Conditionals and Jumps		
cmp *b,a*	Compare b to a; must immediately precede any of the conditional jump instructions	cmp eax,0
je *label*	Jump to label if b == a	je endloop
jne *label*	Jump to label if b != a	jne loopstart
jg *label*	Jump to label if b > a	jg exit
jge *label*	Jump to label if b ≥ a	jge format_disk
jl *label*	Jump to label if b < a	jl error
jle *label*	Jump to label if b ≤ a	jle finish
test *reg,imm*	Bitwise compare of register and constant; should immediately precede the jz or jnz instructions	test eax,0xffff
jz *label*	Jump to label if bits were **not** set ("zero")	jz looparound
jnz *label*	Jump to label if bits **were** set ("not zero")	jnz error
jmp *label*	Unconditional relative jump	jmp exit
jmp *reg*	Unconditional absolute jump; arg is a register	jmp eax
Miscellaneous		
nop	No-op (opcode 0x90)	nop
hlt	Halt the CPU	hlt

Instructions with no memory references must include 'byte', 'word' or 'dword' size specifier.
Arguments to instructions: Note that it is not possible for **both** src and dest to be memory addresses.

Constant (decimal or hex):	10 or 0xff	Fixed address:	[200] or [0x1000+53]
Register:	eax bl	Dynamic address:	[eax] or [esp+16]

32-bit registers: eax, ebx, ecx, edx, esi, edi, ebp, esp (points to first used location on top of stack)
16-bit registers: ax, bx, cx, dx, si, di, sp, bp
8-bit registers: al, ah, bl, bh, cl, ch, dl, dh

Document Design

System Overview

Tetris on a kernel is designed to run on bare-metal systems with no underlying operating system. It uses NASM for the bootloader and is programmed in the C programming language. The game's logic, graphics rendering, and user input are only a few examples of the components that make up the kernel.

The boot loader handles the system's initialisation, CPU configuration, and kernel memory loading. It sets up the GDT, a processor-used data structure for controlling memory access. To prepare the system for kernel execution, it also initialises other hardware parts, such as the disk drive.

The kernel takes control and starts controlling system resources as soon as it is loaded into memory. Handling hardware signals and ensuring the kernel reacts appropriately fall under the purview of the interrupt handler. It enables the kernel to communicate with hardware components like the mouse and keyboard.

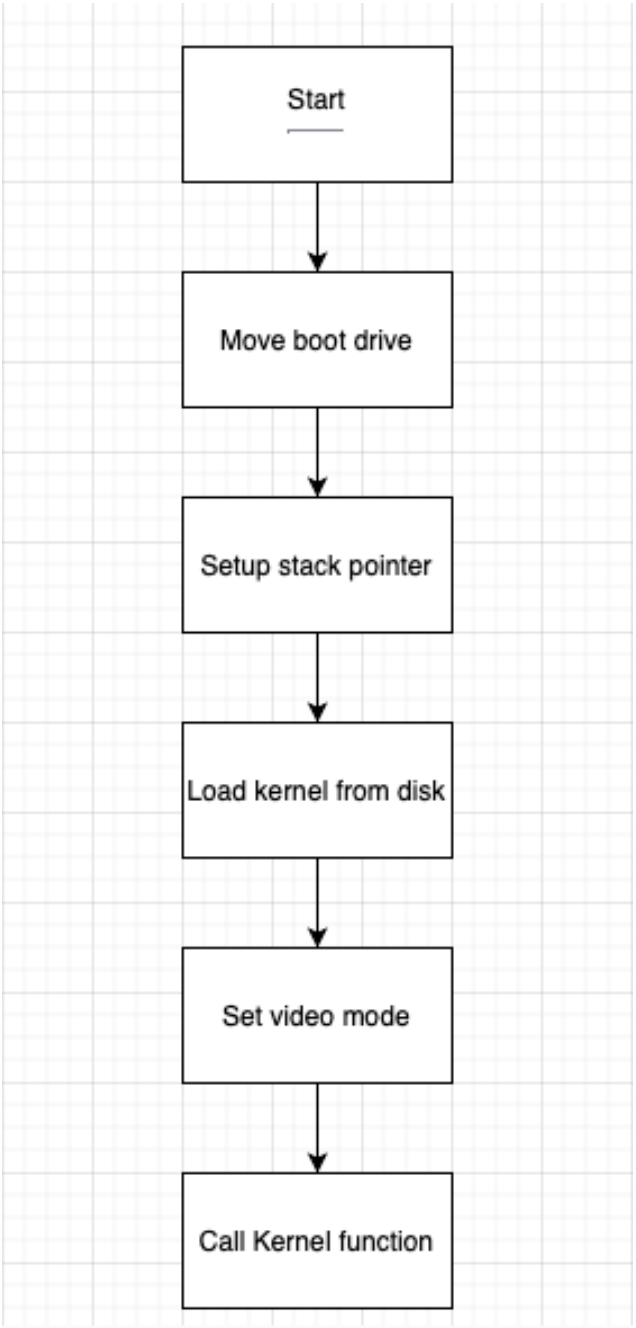
The graphics rendering functionality used by the kernel is provided via a proprietary graphics library. The game is shown on the screen using the VGA graphics mode. The game's graphics, including the blocks and game board, are rendered using the library.

The game's behaviour is governed by the game logic, which is implemented in the kernel. It manages collision detection, clearing finished rows, and producing and moving falling blocks. Additionally, it controls the game's level and score, which are visible on the screen.

The interrupt handler, which watches for keyboard and mouse events, handles user input. The game kernel converts these signals into actions like rotating or shifting the falling block to the left or right.

In conclusion, the kernel is a standalone system created to function on bare-metal platforms, to sum up. It implements the game logic to regulate the game's behaviour and uses a bespoke graphics library to render the game's visuals. The interrupt handler handles user input, and the kernel controls the game's score and level. Overall, the kernel provides the Tetris gaming experience, showcasing the strength and adaptability of low-level programming languages.

Bootloader Process



Bootloader overview:

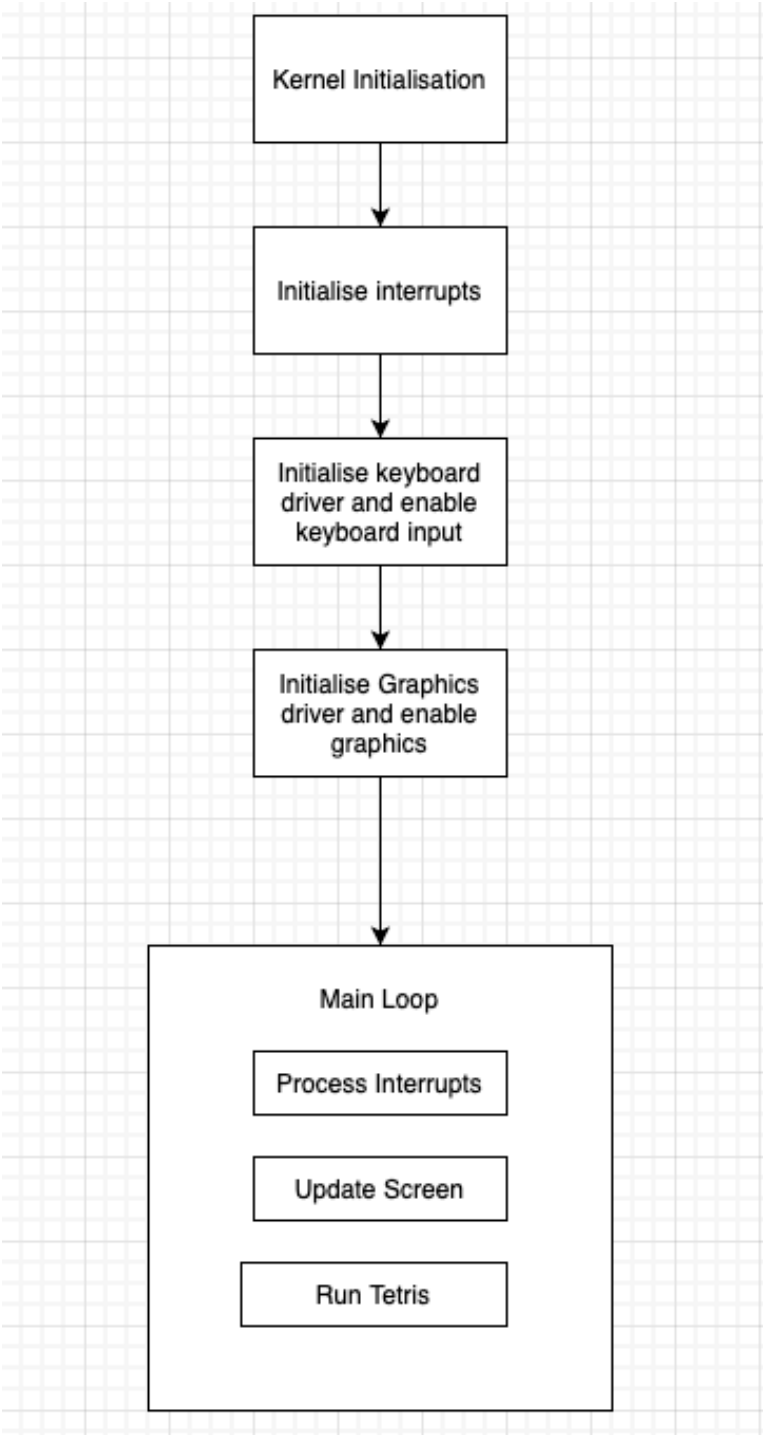
When a computer is powered on or restarted, the BIOS runs a programme known as a boot loader. The boot loader's main job is to execute the operating system kernel by loading it from a storage medium like a hard drive or USB drive into memory. In addition, the boot loader carries out some crucial activities, including initialising hardware devices, configuring memory, and setting up the system's environment.

The boot loader code is located in the hard drive's first sector, the Master Boot Record (MBR). The boot loader begins operating when the BIOS loads the MBR into memory and jumps to its address. Next, the assembly language file containing the boot loader code loads the kernel from the storage device, configures the video mode, and enters the 32-bit protected mode.

The operating system can utilise virtual memory and other cutting-edge capabilities in 32-bit protected mode. The Global Descriptor Table (GDT), which specifies the memory segments the operating system can access, is also set up by the boot loader. The boot loader finally hands off the command to the kernel, which assumes control and begins running the operating system.

In conclusion, the boot loader is required because it creates the environment needed for the operating system to execute and loads the operating system kernel into memory. In addition, the assembly language file's boot loader code handles crucial functions such as loading the kernel, configuring the video mode, and switching to 32-bit protected mode.

Kernel Process



Kernel overview:
A kernel, at its most basic level, is a piece of software that acts as the brains of an operating system. It oversees the computer's hardware resources and offers support to programmes that run on top of the operating system.

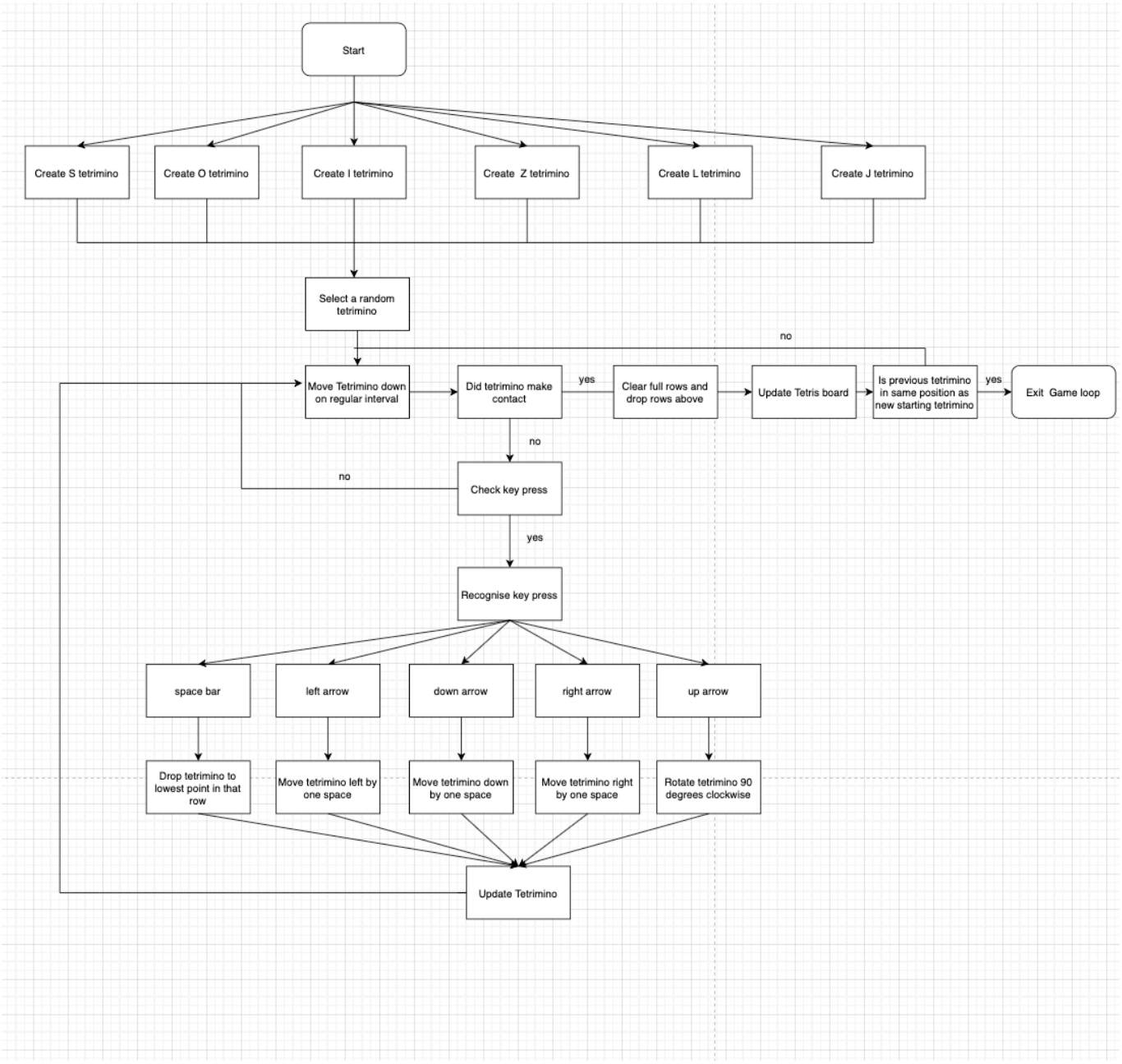
Managing interruptions is one of a kernel's main responsibilities. Interrupts are signals that hardware devices, like a keyboard or mouse, send to the CPU to let it know that something has happened that needs to be dealt with. The kernel assigns each interrupt to a specific Interrupt Service Routine (ISR), which responds to the interrupt and takes any necessary actions using an Interrupt Descriptor Table (IDT). This functionality is handled by the IDT and ISR files that you referenced in your kernel code.

The kernel comprises drivers for many hardware devices, including the keyboard and controlling interrupts. For example, a keyboard driver manages keyboard input and converts it into a language that the operating system and programmes can understand. This driver is probably implemented in your kernel code's keyboard.c file.

Managing visuals is one of a kernel's many common tasks. Typically, this entails choosing the video mode, which controls the display's resolution and colour depth, and then adding images and text to the screen. Your kernel code's graphics functionality is probably based on Video Mode 13, a typical VGA mode that offers a resolution of 320x200 pixels with 256 colours.

Overall, the kernel is an essential part of every operating system since it lays the groundwork for the proper operation of all other software and hardware. Depending on the operating system and the hardware it is intended to run on a kernel's specific features and functions can change.

Tetris Process



Tetris Overview:
Playing the popular puzzle game, Tetris involves moving pieces called tetrominoes. Tetrominos fall down the playing area, and the game's goal is to place and rotate them to form horizontal lines of ten blocks. A line that has been finished is cleared, and any blocks that are above it will fall to fill the void. When the playing surface is completely covered in tetrominoes, the game is ended.

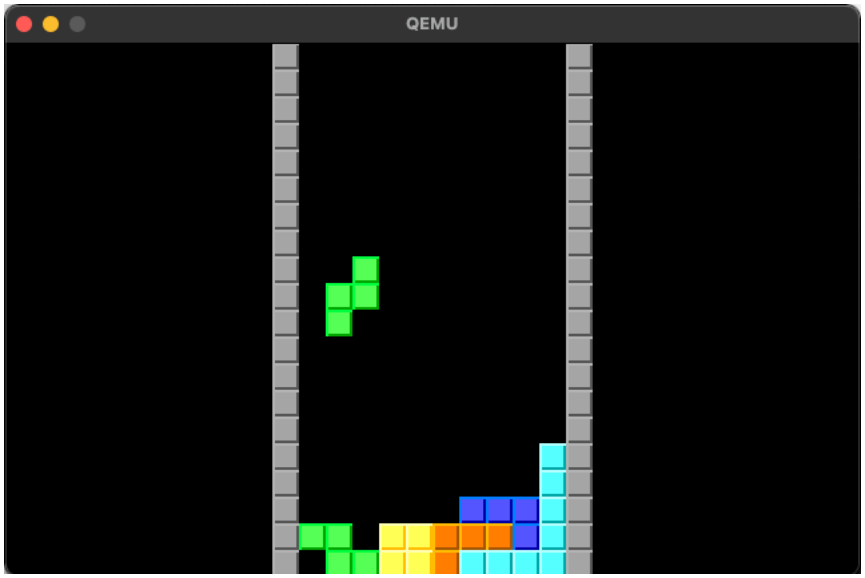
A random tetromino drops into the playing field at the start of the game. The tetromino can be rotated and moved horizontally to fit into the existing blocks below. In addition, the tetromino can be rotated clockwise, moved left or right by one block, dropped to the bottom of the playing area, or any combination of these actions.

The game speeds up as the player clears more lines, making it harder to place the tetrominoes carefully. However, the game continues until there are no more spaces that the player can fill in using tetrominoes.

The line(s) will be cleared, and any blocks above the line(s) will fall to fill the gaps when the player completes one or more horizontal lines of ten blocks.

Human-Computer Interaction (HCI)

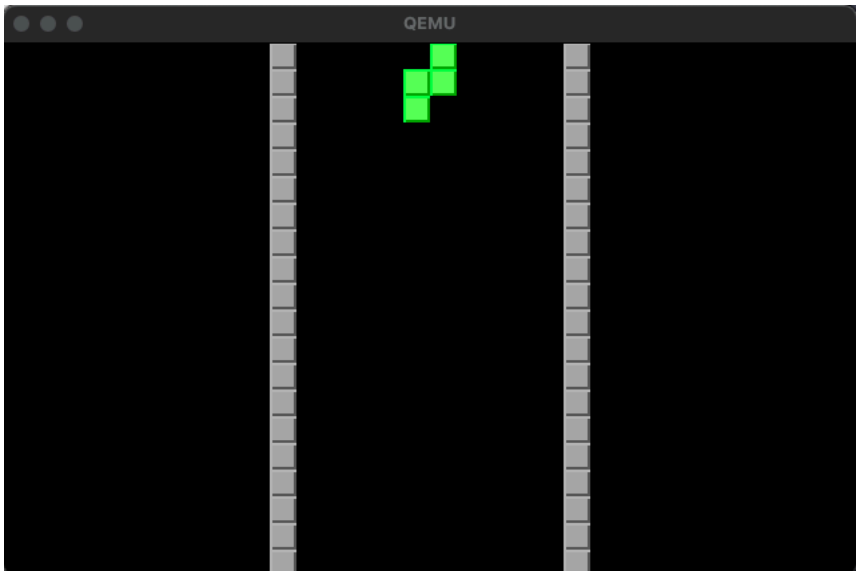
The diagram below shoes what my Tetris game will look like running on the kernel, including the tetromino, borders and how tetrominoes are placed.



The Tetris game's user interface has a black background and grey borders around the playing field. Each of the geometric forms known as tetrominoes, which fall onto the game board, is represented by a distinct colour. For example, the tetromino in the I shape is light blue, the tetromino in the J shape is dark blue, the tetromino in the L shape is orange, the tetromino in the O shape is yellow, the tetrimino in the S shape is green, the tetromino in the T shape is purple. The tetromino in the Z shape is red. A good and in-depth picture of the gameplay is provided by the gaming board's 320 x 200 pixel resolution. Overall, the user interface is appealing to the eye and simple to use, enabling players to immerse themselves in the game and its dynamics fully.

Error screen:
I have designed my kernel to handle various types of errors that may occur during execution. These errors include but are not limited to division by zero, array out of bounds, and others.
The system will halt when an error is detected and show a blue screen, crashing the system.

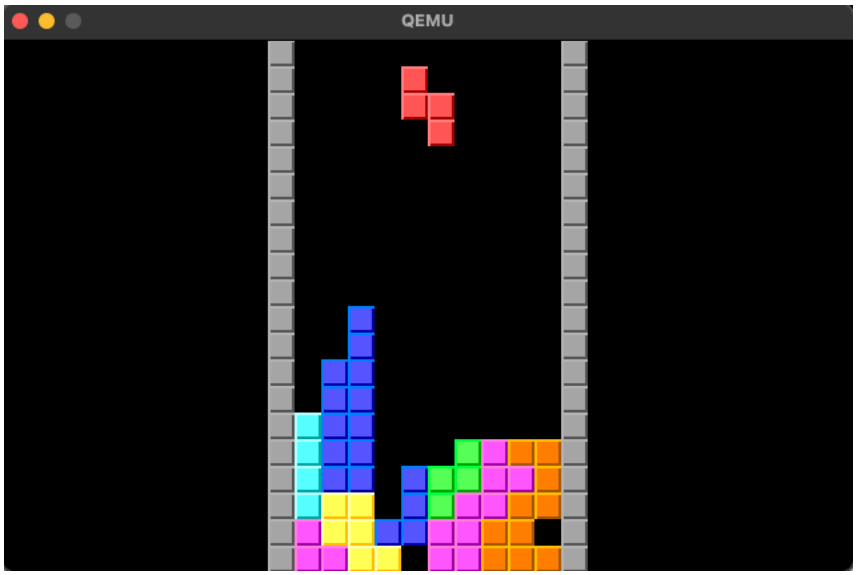
Startup image:
When the kernel is run the user is instantly taken to the game and the mechanics of the game take place.



When the game is first loaded, the user interface shows a black background with grey borders and a random tetromino at the top of the screen, ready to drop. The tetromino is coloured according to its shape, as per the standard Tetris game.

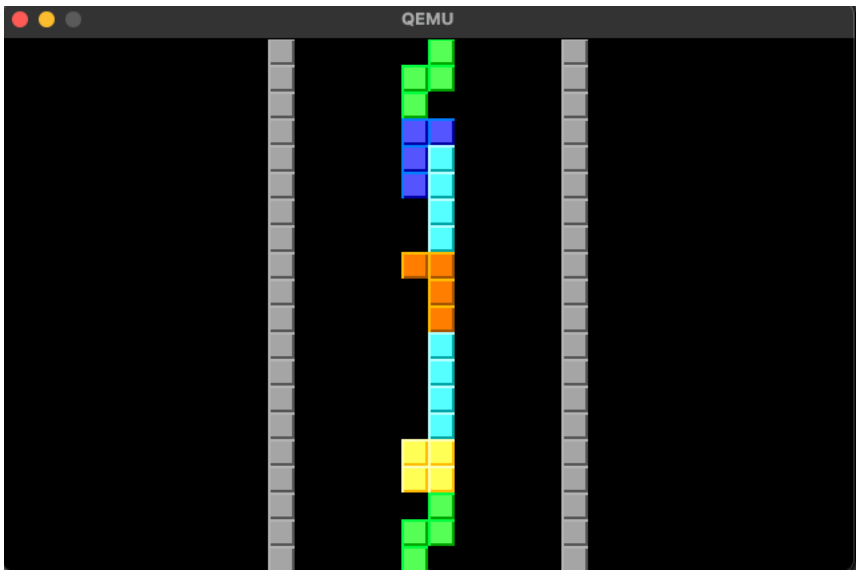
Game in progress:

The following is what the kernel would display while being run and a user playing the game.

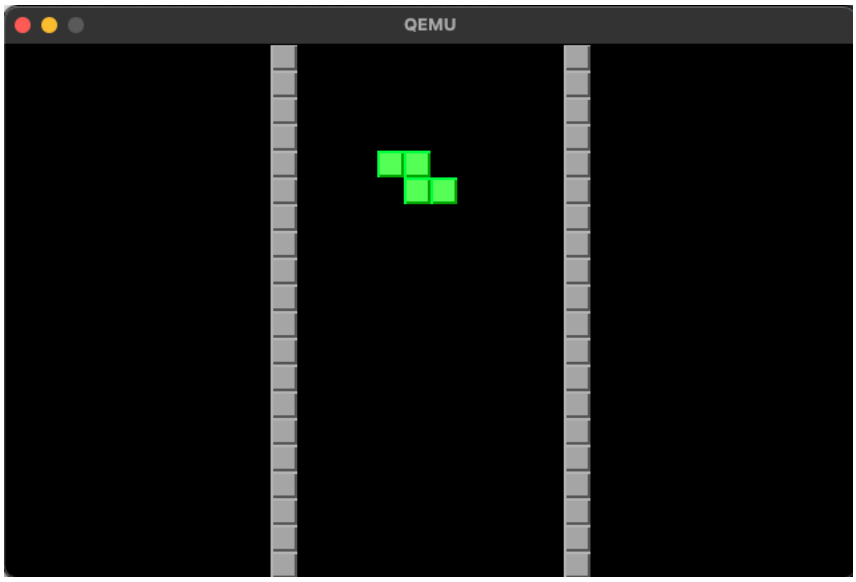


In the picture I provided, the user interface shows a Tetris game in progress, with a falling tetromino and various tetrominoes already placed on the game board. Some lines have already been destroyed due to the placement of the tetrominoes. Each tetromino is displayed in its corresponding colour, and the grey borders around them help to distinguish each piece. The background of the game is black, which makes the brightly coloured tetrominoes stand out even more. Overall, the user interface looks clean and intuitive, making it easy for the user to understand the game mechanics and progress through the levels.

Game Ended:
The following represents what will occur once the game is over and there is an existing piece in the starting position for the new tetromino.



Controls:



Use the arrow keys on your keyboard and the space bar to control the Tetris game on this kernel. The current tetromino can be moved using the right and left arrow keys to the right and left. To change the current tetromino's position on the board, press the up arrow key to rotate it 90 degrees clockwise. You can carefully position the current tetromino before dropping it by using the down arrow key to move it down one cell at a time. Finally, pressing the space bar will quickly advance you to the following piece by dropping the current tetromino to the bottom of the board.

Input	Result
Left arrow key	Tetromino moves left by one square
Right arrow key	Tetromino moves right by one square
Up arrow key	Tetromino rotates 90 degrees clockwise
Down arrow key	Tetromino moves down by one square
Space bar key	Tetromino falls to the lowest position in the current row

Pseudocode for key algorithms

Load disk:

The following code shows how the disk is loaded.

```
disk_load:
   pusha ; push edi, esi, ebp, esp, ebx, edx, ecx, eax

   ; Save the value of 'num_sectors' that we will overwrite with the read command
   push dx

   ; Set the values of the registers required by the BIOS read function
   mov ah, 0x02 ; Read sectors function
   mov al, dh ; Number of sectors to read
   mov cl, 0x02 ; First sector to read
   mov ch, 0x00 ; Cylinder number
   mov dh, 0x00 ; Head number
   int 0x13 ; BIOS interrupt
   jc disk_error ; Jump if carry flag is set (i.e., an error occurred)

   ; Check if the expected number of sectors was read
   pop dx ; Pop the value of 'num_sectors' that we saved earlier
   cmp al, dh ; Compare the number of sectors read with the expected number
   jne sectors_error ; Jump to sectors_error if they are not equal

   popa ; pop eax, ecx, edx, ebx, esp, ebp, esi, edi
   ret ; Return from the function
```

The "disk_load" method in this code loads data from the disc. First, it reads a certain number of sectors from the disc into a buffer pointed to by a pointer given as an argument using BIOS interrupt 0x13. The read instruction briefly saves the dh register on the stack before overwriting it with the number of sectors to read. The function then determines whether the anticipated number of sectors were read and, if so, returns with the data. If not, the process diverges to error-handling code to address the particular error scenario.

Interrupts:

The following code saves the state of the CPU, calls the interrupt service routine (ISR), and restores the state of the CPU before returning from the interrupt.

```
PROCEDURE isr_common_stub
   pusha              ; push values of edi, esi, ebp, esp, ebx, edx, ecx, and eax onto the stack
   mov ax, ds          ; move the data segment register into the lower 16-bits of eax
   push eax           ; save the value of the data segment descriptor
   mov ax, 0x10        ; set the value of the kernel data segment descriptor
   mov ds, ax          ; set ds to the kernel data segment descriptor
   mov es, ax          ; set es to the kernel data segment descriptor
   mov fs, ax          ; set fs to the kernel data segment descriptor
   mov gs, ax          ; set gs to the kernel data segment descriptor
   push esp           ; push the pointer to the registers_t structure on the stack
   call interupt_service_routine   ; call the interrupt service routine function
   pop eax            ; clear the pointer to the registers_t structure from the stack
   pop eax            ; pop the value of the data segment descriptor from the stack
   mov ds, ax          ; restore ds to its original value
   mov es, ax          ; restore es to its original value
   mov fs, ax          ; restore fs to its original value
   mov gs, ax          ; restore gs to its original value
   popa              ; pop the values of edi, esi, ebp, esp, ebx, edx, ecx, and eax from the stack
   add esp, 8          ; clean up the pushed error code and pushed ISR number
   iret             ; return from the interrupt

; Common IRQ code. Identical to ISR code except for the 'call'
; and the 'pop ebx'
```

Explanation of isr_common_stub:

The code provides a common framework for handling interrupts by saving and restoring the state of the CPU, calling a C function that handles the interrupt, and then returning from the interrupt.

The code begins with a common ISR stub for all types of interrupts. This stub saves the state of the CPU by pushing the values of various registers onto the stack, saves the value of the data segment descriptor, sets the data segment descriptor to the kernel data segment, calls the C handler function by pushing the pointer to the registers_t structure on the stack and calling the function, restores the state of the CPU by popping values from the stack and restoring the data segment descriptor to its original value, and returns from the interrupt using the "iret" instruction.

The code also is an example of an IRQ handler, a type of ISR that is called in response to a hardware interrupt from a device such as a keyboard. The IRQ handler is identical to the ISR stub except for the function call and the pop of the ebx register.

Key Pressed:
The following subroutine shows how the kernel responds to inputs made by the keyboard, with each key having a corresponding scancode.

```
PROCEDURE key_entered(scancode: uint8_t)
   switch scancode
      case 0x39 ->  // space bar arrow key
         fall()

      case 0x4b ->  // left arrow key
         shiftleft()

      case 0x50 ->  // down arrow key
         shiftdown()

      case 0x4d ->  // right arrow key
         shiftright()

      case 0x48 ->  // up arrow key
         shiftup()
         shiftdown()
         Rotate()
         draw_tetrimino()

      otherwise -> // handle other keys (optional)
         // do nothing or print an error message
   end switch
end PROCEDURE
```

Updating the tetris board:
The following code shows how the tetris board is updated in my program.

```
PROCEDURE draw_tetrimino()
  DECLARE checkX, checkY as INTEGER

  FOR i FROM 0 TO (BOARD_HEIGHT * BOARD_WIDTH) - 1 DO
    checkX ← i MOD 10
    checkY ← i DIV 10

    IF board[i] = 'X' THEN
      IF TetriminoType = 0 THEN
        Cyanblock(checkX + 11, checkY)
      ELSE IF TetriminoType = 1 THEN
        Greenblock(checkX + 11, checkY)
      ELSE IF TetriminoType = 2 THEN
        Yellowblock(checkX + 11, checkY)
      ELSE IF TetriminoType = 3 THEN
        Redblock(checkX + 11, checkY)
      ELSE IF TetriminoType = 4 THEN
        Pinkblock(checkX + 11, checkY)
      ELSE IF TetriminoType = 5 THEN
        Blueblock(checkX + 11, checkY)
      ELSE IF TetriminoType = 6 THEN
        Orangeblock(checkX + 11, checkY)
      END IF
    ELSE IF board[i] = '1' THEN
      Cyanblock(checkX + 11, checkY)
    ELSE IF board[i] = '2' THEN
      Greenblock(checkX + 11, checkY)
    ELSE IF board[i] = '3' THEN
      Yellowblock(checkX + 11, checkY)
    ELSE IF board[i] = '4' THEN
      Redblock(checkX + 11, checkY)
    ELSE IF board[i] = '5' THEN
      Pinkblock(checkX + 11, checkY)
    ELSE IF board[i] = '6' THEN
      Blueblock(checkX + 11, checkY)
    ELSE IF board[i] = '7' THEN
      Orangeblock(checkX + 11, checkY)
    ELSE IF board[i] = '.' THEN
      BlackBlock(checkX + 11, checkY)
    END IF
  END FOR
END PROCEDURE
```

Placing a tetromino:
The following code checks if a tetromino should be placed.

```
PROCEDURE CheckIfTetriminoPlaced()
  FOR i ← 190 TO 199
    IF board[i] = 'X' THEN
      // Mark all blocks on the board with '1' to indicate they are fixed
      FOR j ← 0 TO BOARD_WIDTH * BOARD_HEIGHT - 1
        IF board[j] = 'X' THEN
          board[j] ← TetriminoType + 49
        ENDIF
      ENDFOR

      // Set the 'placed' flag to indicate the current tetrimino is no longer being controlled
      placed ← 1

      // Set the starting position of the next tetrimino
      xStart ← 3
      yindex ← 0
    ENDIF
  ENDFOR

  FOR i ← 0 TO BOARD_HEIGHT * BOARD_WIDTH - 1
    IF board[i] = 'X' AND (board[i + 10] = '1' OR board[i + 10] = '2' OR board[i + 10] = '3' OR
board[i + 10] = '4' OR board[i + 10] = '5' OR board[i + 10] = '6' OR board[i + 10] = '7') THEN
      FOR j ← 0 TO BOARD_HEIGHT * BOARD_WIDTH - 1
        IF board[j] = 'X' THEN
          board[j] ← TetriminoType + 49
        ENDIF
      ENDFOR

      // Set the 'placed' flag to indicate the current tetrimino is no longer being controlled
      placed ← 1

      // Set the starting position of the next tetrimino
      xStart ← 3
      yindex ← 0
    ENDIF
  ENDFOR
ENDPROCEDURE
```

Rotate Tetromino:
The following code shows how a tetromino is rotates in my program.

```
PROCEDURE Rotate
  // Loop through the rows and columns of the tetrimino shape
  for y ← 0 to 3 do
    for x ← 0 to 3 do
      // Store the new rotated tetrimino shape in the Newtetrimino array
      Newtetrimino[TetriminoType][oldX] ← tetrimino[TetriminoType][12 + y - (x * 4)]
      oldX ← oldX + 1
    end for
  end for

  // Reset the number of rotations and old X position of the tetrimino
  rotations ← 0
  oldX ← 0

  // Copy the new rotated tetrimino shape to the current tetrimino shape
  for i ← 0 to 15 do
    tetrimino[TetriminoType][i] ← Newtetrimino[TetriminoType][i]
  end for
end PROCEDURE
```

Subroutines and Key Variables

MBR.asm

Subroutines:

Identifier	Parameters	Description
load_kernel	-	Loads the kernel from the disk.
init_kernel	-	Initialises the kernel.
disk_load	-	This subroutine is designed to load data from a disk drive by using the BIOS interrupt 0x13. It saves the values of all the registers, sets up the necessary registers, reads the specified number of sectors, checks for errors, and restores the register values before returning to the caller.
disk_error	-	Handles a disk read error.
sectors_error	-	Handles a incorrect number of sectors read error.
disk_loop	-	Loops indefinitely in the case of a disk error.
DISK_ERROR_MSG	-	A null-terminated string that contains the error message for disk read errors.
SECTORS_ERROR_MSG	-	A null-terminated string that contains the error message for an incorrect number of sectors read.
print16	-	Prints a string in 16-bit real mode.
print16_hex	-	Prints the hexadecimal representation of a value in 16-bit real mode.
PRINT16_HEX_OUT	-	A buffer for storing the output string of print16_hex..

Key Variables:

Identifier	Data Type	Description
boot	Byte	Stores the boot drive in memory.
kernel_offset	Word	Offset of the kernel in memory.

GlobalDescriptorTable.asm

Subroutines:

Identifier	Parameters	Description
gdt_start	-	This is a method used to mark the beginning of the Global Descriptor Table (GDT). The GDT is a data structure used by the operating system to define memory segments used by the CPU. In this code, gdt_start is defined with two null 4-byte entries, representing the first 8 bytes of the GDT.
gdt_code	-	This is a method used to define the code segment in the GDT. The code segment specifies the base address and length of the memory region used for code execution. In this code, the code segment starts at address 0x00000000 and has a length of 0xfffff bytes. The segment is defined with 6 entries, including the length, base address, and various flags used to define the segment's attributes.
gdt_data	-	Label marking the beginning of the GDT data segment descriptor.
gdt_end	-	Label marking the end of the GDT.
global_descriptor_table	-	This method is used to define the GDT descriptor. The descriptor is used by the CPU to locate the GDT in memory. It consists of two parts: a 16-bit size field, which represents the size of the GDT in bytes minus one, and a 32-bit address field, which points to the start of the GDT in memory. The global_descriptor_table label points to the start of the gdt_start label, and the size field is calculated by subtracting gdt_start from gdt_end and subtracting one byte.

Key Variables:

Identifier	Data Type	Description

CODE_SEG	Integer	Represents the offset of the code segment GDT entry from the beginning of the GDT.
DATA_SEG	Integer	Represents the offset of the data segment GDT entry from the beginning of the GDT.

Enter_32bit.asm

Subroutines:

Identifier	Parameters	Description
switch_to_32bit	-	Used to switch the processor from 16-bit mode to 32-bit mode. It disables interrupts, loads the global descriptor table, sets the 32-bit mode bit in the control register (cr0), and performs a far jump to init_32bit using the code segment selector.
init_32bit	-	The init_32bit subroutine is responsible for initializing the system for 32-bit mode. It updates the segment registers, sets the stack pointer to the top of the free space.

Interrupt_Descriptor_Table.c

Subroutines:

Identifier	Parameters	Description
set_idt_gate	int n, uint32_t handler	uses the Interrupt Descriptor Table (IDT) array, a table of interrupt handlers, to manage various hardware or software interrupts. The handler function address and the interrupt number (n) are the two arguments required by the function. The correct fields are then set in the IDT gate structure for the provided interrupt number, including the segment selector (KERNEL_CS), gate type (0x8E), low and high offsets of the handler code, and the always zero field.
load_idt	-	used to load the IDT into memory. It sets the base and limit of the IDT register to the address of the IDT array and its size respectively. Then, it loads the IDT into memory by executing the "lidt" instruction with the address of the IDT register using

		inline assembly. The IDT is now ready to be used by the operating system to handle interrupts.

Key Variables:

Identifier	Data Type	Description
my_idt	idt_gate_t[256]	Interrupt Descriptor Table (IDT) gate array.
my_idt_reg	idt_register_t	IDT register.
KERNEL_CS	uint16_t	Constant for kernel code segment selector.
idt_entry	unt	Number of IDT entries.
idt_gate_t	struct	Structure pf representing an IDT gate.
idt_register_t	struct	Structure of representing the IDT register.
low_offset	uint16_t	Lower 16 bits of the address of the interrupt handler function.
sel	uint16_t	Kernel code segment selector.
always0	uint8_t	Always must be 0.
flags	uint8_t	Flags for the IDT gate.
high_offset	uint16_t	Higher 16 bits of the address of the interrupt handler function.
limit	uint16_t	16-bit unsigned integer representing the IDT limit value.
base	uint32_t	32-bit unsigned integer representing the IDT base address.

Interrupt_Service_Routine.c

Subroutines:

Identifier	Parameters	Description
isr_install	-	This method installs Interrupt Service Routines (ISRs) in the Interrupt Descriptor Table (IDT). The IDT is used to map each interrupt to a specific interrupt service routine. It loops through all 32 possible CPU exceptions, maps

		them to the corresponding ISR and installs them. It then remaps the Programmable Interrupt Controller (PIC) and installs the IRQs.
interupt_service_routine	registers_t *r	This method is called when an interrupt occurs. It takes a pointer to the “registers_t” structure which contains the state of the processor at the time of the interrupt. It then handles the interrupt based on the type of interrupt that occurred. In this particular code, if any error is found, it displays a red screen with the corresponding error message.
register_interrupt_handler	uint8_t number, isr_t handler	This method is used to register an interrupt handler for a specific interrupt number. It takes two arguments - the interrupt number and the address of the interrupt handler. It stores the interrupt handler address in the “interrupt_handlers” array.
interrupt_request_handler	registers_t *regs	This method is called when an Interrupt Request (IRQ) occurs. It takes a pointer to the “registers_t” structure which contains the state of the processor at the time of the interrupt. It then checks if the interrupt was caused by a hardware device and calls the corresponding interrupt handler if it was. If not, it just sends an end-of-interrupt signal to the PIC to acknowledge the interrupt.

Key Variables:

Identifier	Data Type	Description
interrupt_handlers	isr_t[256]	An array of interrupt service routines (ISRs) to handle interrupts.
handler	isr_t	define an Interrupt Service Routine (ISR) handler function.

Interrupts.asm

Subroutines:

Identifier	Parameters	Description
isr_common_stub	-	This method saves the state of the CPU, calls the C interrupt service routine function, and restores the CPU state before returning.
irq_common_stub	-	This method is identical to isr_common_stub except it calls the C interrupt request handler function.

isr0-isr31	-	These methods represent the various CPU exceptions, and each one pushes an error code and ISR number to the stack before calling isr_common_stub.
irq0-irq15	-	These methods represent the various hardware interrupts, and each one pushes an error code and ISR number to the stack before calling irq_common_stub.

ports.c

Subroutines:

Identifier	Parameters	Description
port_byte_in	uint16_t port	Reads a byte from the port and returns it. It uses the “in” instruction to read the value from the port and maps the port to the “dx” register and the result to the “al” register.
port_byte_out	uint16_t port, uint8_t data	Writes a byte to the port. It uses the “out” instruction to write the value to the port and maps the port to the “dx” register and the data to the “al” register.

display.c

Subroutines:

Identifier	Parameters	Description
clear_screen	-	This method clears the screen by calling the putpixel method with a color code of 0x00 for each pixel on the screen.
putpixel	int pos_x, int pos_y, unsigned char VGA_COLOR	This method draws a pixel on the screen with the specified x and y positions and the specified color.

BlackBlock	int x, int y	This method draws a black block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Grayblock	int x, int y	This method draws a gray block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Yellowblock	int x, int y	This method draws a yellow block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Pinkblock	int x, int y	This method draws a pink block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Cyanblock	int x, int y	This method draws a cyan block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Blueblock	int x, int y	This method draws a blue block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Orangeblock	int x, int y	This method draws a orange block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Greenblock	int x, int y	This method draws a green block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.
Redblock	int x, int y	This method draws a red block on the screen with the specified x and y positions. The block is made up of 100 pixels with a size of 10 x 10.

Key Variables:

Identifer	Data Type	Description
pos_x	int	X-coordinate of the pixel being drawn in the “putpixel” function.
pos_y	int	Y-coordinate of the pixel being drawn in the “putpixel” function.

VGA_COLOUR	unsigned char	Colour of the pixel being drawn in the "putpixel" function.
location	unsigned char pointer	Pointer to the memory location of the pixel being drawn in the "putpixel" function.
square_size	int	Size of the block being drawn in block functions.

keyboard.c

Subroutines:

Identifier	Parameters	Description
key_entered	uint8_t scancode	This method takes in a single parameter, "scancode", which is used in a switch statement to determine which action to take. If the "scancode" matches a case, a corresponding function is called.
keyboard_callback	registers_t *regs	This method is called when an interrupt is triggered by the keyboard. It reads the scancode from the keyboard port and calls "key_entered" with the scancode as a parameter.
init_keyboard	-	This function initializes the keyboard by registering "keyboard_callback" as the interrupt handler for IRQ1.

Key Variables:

Identifier	Data Type	Description
scancode	int	Holds the scancode of the keyboard key that was pressed.
regs	registers_t*	A pointer to a structure that holds the register values during the interrupt.

kernel.c

Subroutines:

Identifier	Parameters	Description
random_number	Unsigned int *seed	Generates a random integer number using the seed passed as parameter and returns a value from 0 to 6.
init_board	-	Initialises the game board array with dots representing empty cells.
placeTetrimino	int tetriminoType, int xPos, int yPos	Places a tetromino on the game board at the starting position. The tetriminoType parameter represents the type of the tetrimino, xPos represents the X position, and yPos represents the Y position. The function updates the board array with the tetromino cells.
Rotate	-	Rotates the current tetromino shape 90 degrees clockwise and stores the new rotated shape in the Newtetrimino array. The function updates the current tetrimino shape with the rotated shape.
draw_tetrimino	-	Draws the current tetromino on the game board. The function updates the board array with the Tetromino cells.
start_kernel	-	Main kernel function of the containing code running Tetris combining all necessary functions to run the game.
shiftleft	-	Shifts the current tetrimino one cell to the left on the game board. If there is no space, the tetrimino stays in place.
shiftright	-	Shifts the current tetrimino one cell to the right on the game board. If there is no space, the tetrimino stays in place.
shiftdown	-	Shifts the current tetrimino one cell down on the game board. If there is no space, the tetrimino is placed on the board.
shiftup	-	Shifts the current tetrimino one cell up on the game board. If there is no space, the tetrimino stays in place.

CheckIfTetriminoPlaced	-	Checks if the current tetrimino can be placed at the given x and y position on the game board. Returns true if the tetrimino can be placed, false otherwise.
clear_line	-	Clears the specified line on the game board, and moves all lines above it down by one cell.
fall	-	Calculates the number of empty spaces that the blocks can fall through and moves the blocks down by the calculated distance.
GameEnd	-	Checks whether the game has ended. If the starting tetromino already has a block present it will exit the game loop.

Key Variables:

Identifier	Data Type	Description
TETRIMINO_AMOUNT	int	Number of different tetromino shapes available.
TETRIMINO_SIZE	int	Size of each tetromino shape in 1D array.
BOARD_WIDTH	int	Width of the game board.
BOARD_HEIGHT	int	Height of the game board.
TETRIMINO_ROWS	int	Number of rows in each tetromino shape.
TETRIMINO_COLS	int	Number of columns in each tetromino shape.
xStart	int	Starting X position for the tetromino.
yindex	int	Starting Y position for the tetromino.
placed	int	Flag indicating wether the tetromino is placed.
TetriminoType	int	Index of the cuttent tetromino shape.
seed	unsinged int	Seed used to generate a random number.
loop	int	Loop counter used for timing.
timer	int	Timer value used for timing.
rotations	int	Number of times the current tetromino has rotated.

oldX	int	Previous X position of the tetromino.
newX	int	New X position of the tetromino.
falldistance	int	Distance the tetromino has fallen.
row	int	Row on which the current tetromino is located.
checkX	int	Calculates the X position of the current cell.
checkY	int	Calculates the current Y position of the current cell.
board	char	Array for the Tetris board.
boardIndex	int	Calculates the starting index in the board 1D array.
boardOffset	int	Calculates the index of the current cell in the board array.
tetriminoIndex	int	Calculates the index of the current cell in the tetromino array.

Technical Solution

boot folder

MBR.asm:

```
[org 0x7c00]

; Move the boot drive to boot
mov [boot], dl

; Set up the stack pointer
mov bp, 0x9000
mov sp, bp

; Load the kernel from disk
call load_kernel
push ax

; Set the video mode
mov ah, 0x00
mov al, 0x13
int 0x10
pop ax

; Switch to 32-bit protected mode
call switch_to_32bit
jmp $ ; Never executed

%include "boot/GlobalDescriptorTable.asm"
%include "boot/Enter_32bit.asm"

[bits 16]
load_kernel:
   ; Load the kernel from disk
   mov bx, kernel_offset
   mov dh, 31
   mov dl, [boot]
   call disk_load
   ret

[bits 32]
init_kernel:
   ; Call the kernel
   call kernel_offset

   ; Stay here when the kernel returns control to us (if ever)
   jmp $



; Set the kernel offset
kernel_offset equ 0x1000
```

```
; Store the boot drive in memory
boot db 0

disk_load:
   pusha

   ; Save the value of 'num_sectors' that we will overwrite with the read command
   push dx

   ; Set the values of the registers required by the BIOS read function
   mov ah, 0x02   ; Read sectors function
   mov al, dh
   mov cl, 0x02   ; First available sector
   mov ch, 0x00   ; Cylinder
   mov dh, 0x00   ; Head number



   int 0x13       ; BIOS interrupt
   jc disk_error  ; Jump if carry flag is set (i.e., an error occurred)

   ; Check if the expected number of sectors was read
   pop dx
   cmp al, dh
   jne sectors_error

   popa
   ret

disk_error:
   ; Print error message and code
   mov bx, DISK_ERROR_MSG
   call print16
   mov dh, ah
   call print16_hex
   jmp disk_loop

sectors_error:
   ; Print error message
   mov bx, SECTORS_ERROR_MSG
   call print16

disk_loop:
   jmp $

DISK_ERROR_MSG: db "Disk read error", 0
```

```
SECTORS_ERROR_MSG: db "Incorrect number of sectors read", 0


; This subroutine prints a null-terminated string in 16-bit real mode
print16:
   pusha
   ; Print each character until a null byte is found

print16_hex:
   pusha
   mov cx, 0 ; Initialize index variable to 0


; Define the output string buffer for print16_hex
PRINT16_HEX_OUT:
   db '0x0000',0 ; reserve memory for our new string




; padding
times 510 - ($-$$) db 0
dw 0xaa55
```

GlobalDescriptorTable.asm:

```
gdt_start: ; don't remove the labels, they're needed to compute sizes and jumps
   ; the GDT starts with a null 8-byte
   dd 0x0 ; 4 byte
   dd 0x0 ; 4 byte

; GDT for code segment. base = 0x00000000, length = 0xfffff
; for flags, refer to os-dev.pdf document, page 36
gdt_code:
   dw 0xffff    ; segment length, bits 0-15
   dw 0x0       ; segment base, bits 0-15
   db 0x0       ; segment base, bits 16-23
   db 10011010b ; flags (8 bits)
   db 11001111b ; flags (4 bits) + segment length, bits 16-19
   db 0x0       ; segment base, bits 24-31

; GDT for data segment. base and length identical to code segment
```

```
; some flags changed, again, refer to os-dev.pdf
gdt_data:
   dw 0xffff
   dw 0x0
   db 0x0
   db 10010010b
   db 11001111b
   db 0x0

gdt_end:

; GDT descriptor
global_descriptor_table:
   dw gdt_end - gdt_start - 1 ; size (16 bit), always one less of its true size
   dd gdt_start ; address (32 bit)

; define some constants for later use
CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

Enter_32bit.asm:

```
[bits 16]
switch_to_32bit:
   cli ; Disable interrupts
   lgdt [global_descriptor_table] ; Load the GDT descriptor
   mov eax, cr0
   or eax, 0x1 ; Set 32-bit mode bit in cr0
   mov cr0, eax
   jmp CODE_SEG:init_32bit ; Far jump using a different segment

[bits 32]
init_32bit: ; We are now using 32-bit instructions
   mov ax, DATA_SEG ; Update the segment registers
   mov ds, ax
   mov ss, ax
   mov es, ax
   mov fs, ax
   mov gs, ax

   mov ebp, 0x90000 ; Set the stack right at the top of the free space
   mov esp, ebp
```

```
   call init_kernel ; Call a well-known label with useful code
```

kernel_entry.asm:

```
global _start;
[bits 32]

_start:
   [extern start_kernel] ; Define calling point. Must have same name as kernel.c 'main'
function
   call start_kernel ; Calls the C function. The linker will know where it is placed in memory
   jmp $
```

cpu folder

Interrupt_Descriptor_Table.c

```
#include "Interrupt_Descriptor_Table.h"
#include "../kernel/util.h"

// Declare IDT (Interrupt Descriptor Table) gate and register arrays
idt_gate_t my_idt[idt_entry];
idt_register_t my_idt_reg;

// Function to set an IDT gate
void set_idt_gate(int n, uint32_t handler) {
   my_idt[n].low_offset = low_16(handler);
   my_idt[n].sel = KERNEL_CS;
   my_idt[n].always0 = 0;
   my_idt[n].flags = 0x8E;
   my_idt[n].high_offset = high_16(handler);
}

// Function to load the IDT
void load_idt() {
```

```
   my_idt_reg.base = (uint32_t) &my_idt;
   my_idt_reg.limit = idt_entry * sizeof(idt_gate_t) - 1;
   /* The IDT is loaded by executing the "lidt" instruction with the address of the IDT register
*/
   asm volatile("lidt (%0)" : : "r" (&my_idt_reg));
}
```

Interrupt_Descriptor_Table.h

```
#include <stdint.h>

#define KERNEL_CS 0x08

#define idt_entry 256

void set_idt_gate(int n, uint32_t handler);

void load_idt();

typedef struct {
   uint16_t low_offset;
   uint16_t sel;
   uint8_t always0;
   uint8_t flags;
   uint16_t high_offset;
} __attribute__((packed)) idt_gate_t;

typedef struct {
   uint16_t limit;
   uint32_t base;
} __attribute__((packed)) idt_register_t;
```

Interrupt_Service_Routine.c

```
#include "Interrupt_Service_Routine.h"
#include "Interrupt_Descriptor_Table.h"
```

```
#include "../kernel/util.h"

isr_t interrupt_handlers[256];

/* Can't do this with a loop because we need the address
 * of the function names */
void isr_install() {
    set_idt_gate(0, (uint32_t) isr0);
    set_idt_gate(1, (uint32_t) isr1);
    set_idt_gate(2, (uint32_t) isr2);
    set_idt_gate(3, (uint32_t) isr3);
    set_idt_gate(4, (uint32_t) isr4);
    set_idt_gate(5, (uint32_t) isr5);
    set_idt_gate(6, (uint32_t) isr6);
    set_idt_gate(7, (uint32_t) isr7);
    set_idt_gate(8, (uint32_t) isr8);
    set_idt_gate(9, (uint32_t) isr9);
    set_idt_gate(10, (uint32_t) isr10);
    set_idt_gate(11, (uint32_t) isr11);
    set_idt_gate(12, (uint32_t) isr12);
    set_idt_gate(13, (uint32_t) isr13);
    set_idt_gate(14, (uint32_t) isr14);
    set_idt_gate(15, (uint32_t) isr15);
    set_idt_gate(16, (uint32_t) isr16);
    set_idt_gate(17, (uint32_t) isr17);
    set_idt_gate(18, (uint32_t) isr18);
    set_idt_gate(19, (uint32_t) isr19);
    set_idt_gate(20, (uint32_t) isr20);
    set_idt_gate(21, (uint32_t) isr21);
    set_idt_gate(22, (uint32_t) isr22);
    set_idt_gate(23, (uint32_t) isr23);
    set_idt_gate(24, (uint32_t) isr24);
    set_idt_gate(25, (uint32_t) isr25);
    set_idt_gate(26, (uint32_t) isr26);
    set_idt_gate(27, (uint32_t) isr27);
    set_idt_gate(28, (uint32_t) isr28);
    set_idt_gate(29, (uint32_t) isr29);
    set_idt_gate(30, (uint32_t) isr30);
    set_idt_gate(31, (uint32_t) isr31);

    // Remap the PIC
    port_byte_out(0x20, 0x11);
    port_byte_out(0xA0, 0x11);
    port_byte_out(0x21, 0x20);
    port_byte_out(0xA1, 0x28);
    port_byte_out(0x21, 0x04);
    port_byte_out(0xA1, 0x02);
```

```
    port_byte_out(0x21, 0x01);
    port_byte_out(0xA1, 0x01);
    port_byte_out(0x21, 0x0);
    port_byte_out(0xA1, 0x0);

    // Install the IRQs
    set_idt_gate(32, (uint32_t)irq0);
    set_idt_gate(33, (uint32_t)irq1);
    set_idt_gate(34, (uint32_t)irq2);
    set_idt_gate(35, (uint32_t)irq3);
    set_idt_gate(36, (uint32_t)irq4);
    set_idt_gate(37, (uint32_t)irq5);
    set_idt_gate(38, (uint32_t)irq6);
    set_idt_gate(39, (uint32_t)irq7);
    set_idt_gate(40, (uint32_t)irq8);
    set_idt_gate(41, (uint32_t)irq9);
    set_idt_gate(42, (uint32_t)irq10);
    set_idt_gate(43, (uint32_t)irq11);
    set_idt_gate(44, (uint32_t)irq12);
    set_idt_gate(45, (uint32_t)irq13);
    set_idt_gate(46, (uint32_t)irq14);
    set_idt_gate(47, (uint32_t)irq15);

    load_idt(); // Load with ASM
}



void interupt_service_routine(registers_t *r) {
    char s[3];

                                // if any interupt error found displays a red screen:
                                //errors:
                                //    "Divide by zero",
                                //    "Debug",
                                //    "NMI",
                                //    "Breakpoint",
                                //    "Overflow",
                                //    "OOB",
                                //    "Invalid opcode",
                                //    "No coprocessor",
                                //    "Double fault",
                                //    "Coprocessor segment overrun",
                                //    "Bad TSS",
                                //    "Segment not present",
                                //    "Stack fault",
                                //    "General protection fault",
```

```
                    //   "Page fault",
                    //   "Unrecognized interrupt",
                    //   "Coprocessor fault",
                    //   "Alignment check",
                    //   "Machine check",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED",
                    //   "RESERVED"

for(int i = 0; i < (320*200); i++){
   putpixel(i, 0, 0x01);
}


}


void register_interrupt_handler(uint8_t number, isr_t handler) {
   interrupt_handlers[number] = handler;
}

void interrupt_request_handler(registers_t *regs) {
   if (interrupt_handlers[regs->int_no] != 0) {
      isr_t handler = interrupt_handlers[regs->int_no];
      handler(regs);
   }
   if (regs->int_no >= 40) {
      // Send the EOI signal to the slave PIC
      port_byte_out(0xA0, 0x20);
   }
   // Send the EOI signal to the master PIC
   port_byte_out(0x20, 0x20);
}
```

Interrupt_Service_Routine.h

```
#pragma once

#include

/* ISRs reserved for CPU exceptions */
extern void isr0();

extern void isr1();

extern void isr2();

extern void isr3();

extern void isr4();

extern void isr5();

extern void isr6();

extern void isr7();

extern void isr8();

extern void isr9();

extern void isr10();

extern void isr11();

extern void isr12();

extern void isr13();

extern void isr14();

extern void isr15();

extern void isr16();

extern void isr17();

extern void isr18();

extern void isr19();

extern void isr20();
```

```
extern void isr21();

extern void isr22();

extern void isr23();

extern void isr24();

extern void isr25();

extern void isr26();

extern void isr27();

extern void isr28();

extern void isr29();

extern void isr30();

extern void isr31();

/* IRQ definitions */
extern void irq0();

extern void irq1();

extern void irq2();

extern void irq3();

extern void irq4();

extern void irq5();

extern void irq6();

extern void irq7();

extern void irq8();

extern void irq9();

extern void irq10();

extern void irq11();
```

```
extern void irq12();

extern void irq13();

extern void irq14();

extern void irq15();

#define IRQ0 32
#define IRQ1 33
#define IRQ2 34
#define IRQ3 35
#define IRQ4 36
#define IRQ5 37
#define IRQ6 38
#define IRQ7 39
#define IRQ8 40
#define IRQ9 41
#define IRQ10 42
#define IRQ11 43
#define IRQ12 44
#define IRQ13 45
#define IRQ14 46
#define IRQ15 47

/* Struct which aggregates many registers.
 * It matches exactly the pushes on interrupt.asm. From the bottom:
 * - Pushed by the processor automatically
 * - `push byte`s on the isr-specific code: error code, then int number
 * - All the registers by pusha
 * - `push eax` whose lower 16-bits contain DS
 */
typedef struct {
   uint32_t ds; /* Data segment selector */
   uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax; /* Pushed by pusha. */
   uint32_t int_no, err_code; /* Interrupt number and error code (if applicable) */
   uint32_t eip, cs, eflags, useresp, ss; /* Pushed by the processor automatically */
} registers_t;

void isr_install();

void interupt_service_routine(registers_t *regs);

typedef void (*isr_t)(registers_t *);

void register_interrupt_handler(uint8_t number, isr_t handler);
```

interrupts.asm:

```
[extern interrupt_service_routine]
[extern interrupt_request_handler]

global isr0
global isr1
global isr2
global isr3
global isr4
global isr5
global isr6
global isr7
global isr8
global isr9
global isr10
global isr11
global isr12
global isr13
global isr14
global isr15
global isr16
global isr17
global isr18
global isr19
global isr20
global isr21
global isr22
global isr23
global isr24
global isr25
global isr26
global isr27
global isr28
global isr29
global isr30
global isr31


isr_common_stub:
   ; Save the state of the CPU
   pusha ; Pushes the values of edi, esi, ebp, esp, ebx, edx, ecx, and eax onto the stack
```

```
    mov ax, ds ; Move the data segment register into the lower 16-bits of eax
    push eax ; Save the value of the data segment descriptor
    mov ax, 0x10 ; Set the value of the kernel data segment descriptor
    mov ds, ax ; Set ds to the kernel data segment descriptor
    mov es, ax ; Set es to the kernel data segment descriptor
    mov fs, ax ; Set fs to the kernel data segment descriptor
    mov gs, ax ; Set gs to the kernel data segment descriptor

    ; Call the C handler
    push esp ; Push the pointer to the registers_t structure on the stack
    call interupt_service_routine ; Call the interrupt service routine function
    pop eax ; Clear the pointer to the registers_t structure from the stack

    ; Restore the state of the CPU
    pop eax ; Pop the value of the data segment descriptor from the stack
    mov ds, ax ; Restore ds to its original value
    mov es, ax ; Restore es to its original value
    mov fs, ax ; Restore fs to its original value
    mov gs, ax ; Restore gs to its original value
    popa ; Pop the values of edi, esi, ebp, esp, ebx, edx, ecx, and eax from the stack
    add esp, 8 ; Clean up the pushed error code and pushed ISR number
    iret ; Return from the interrupt

; Common IRQ code. Identical to ISR code except for the 'call'
; and the 'pop ebx'
irq_common_stub:
    ; Save the state of the CPU
    pusha ; Pushes the values of edi, esi, ebp, esp, ebx, edx, ecx, and eax onto the stack
    mov ax, ds ; Move the data segment register into the lower 16-bits of eax
    push eax ; Save the value of the data segment descriptor
    mov ax, 0x10 ; Set the value of the kernel data segment descriptor
    mov ds, ax ; Set ds to the kernel data segment descriptor
    mov es, ax ; Set es to the kernel data segment descriptor
    mov fs, ax ; Set fs to the kernel data segment descriptor
    mov gs, ax ; Set gs to the kernel data segment descriptor

    ; Call the C handler
    push esp ; Push the pointer to the registers_t structure on the stack
    call interrupt_request_handler ; Call the interrupt request handler function
    pop ebx ; Clear the pointer to the registers_t structure from the stack

    ; Restore the state of the CPU
    pop ebx ; Pop the value of ebx from the stack
    mov ds, bx ; Restore ds to its original value
    mov es, bx ; Restore es to its original value
    mov fs, bx ; Restore fs to its original value
    mov gs, bx ; Restore gs to its original value
```

```
    popa ; Pop the values of edi, esi, ebp, esp, ebx, edx, ecx, and eax from the stack
    add esp, 8 ; Clean up the pushed error code and pushed ISR number
    iret ; Return from the interrupt


; 0: Divide By Zero Exception
isr0:
    push byte 0
    push byte 0
    jmp isr_common_stub

; 1: Debug Exception
isr1:
    push byte 0
    push byte 1
    jmp isr_common_stub

; 2: Non Maskable Interrupt Exception
isr2:
    push byte 0
    push byte 2
    jmp isr_common_stub

; 3: Int 3 Exception
isr3:
    push byte 0
    push byte 3
    jmp isr_common_stub

; 4: INTO Exception
isr4:
    push byte 0
    push byte 4
    jmp isr_common_stub

; 5: Out of Bounds Exception
isr5:
    push byte 0
    push byte 5
    jmp isr_common_stub

; 6: Invalid Opcode Exception
isr6:
    push byte 0
    push byte 6
    jmp isr_common_stub
```

```
; 7: Coprocessor Not Available Exception
isr7:
   push byte 0
   push byte 7
   jmp isr_common_stub

; 8: Double Fault Exception (With Error Code!)
isr8:
   push byte 8
   jmp isr_common_stub

; 9: Coprocessor Segment Overrun Exception
isr9:
   push byte 0
   push byte 9
   jmp isr_common_stub

; 10: Bad TSS Exception (With Error Code!)
isr10:
   push byte 10
   jmp isr_common_stub

; 11: Segment Not Present Exception (With Error Code!)
isr11:
   push byte 11
   jmp isr_common_stub

; 12: Stack Fault Exception (With Error Code!)
isr12:
   push byte 12
   jmp isr_common_stub

; 13: General Protection Fault Exception (With Error Code!)
isr13:
   push byte 13
   jmp isr_common_stub

; 14: Page Fault Exception (With Error Code!)
isr14:
   push byte 14
   jmp isr_common_stub

; 15: Reserved Exception
isr15:
   push byte 0
   push byte 15
   jmp isr_common_stub
```

```
; 16: Floating Point Exception
isr16:
   push byte 0
   push byte 16
   jmp isr_common_stub

; 17: Alignment Check Exception
isr17:
   push byte 0
   push byte 17
   jmp isr_common_stub

; 18: Machine Check Exception
isr18:
   push byte 0
   push byte 18
   jmp isr_common_stub

; 19: Reserved
isr19:
   push byte 0
   push byte 19
   jmp isr_common_stub

; 20: Reserved
isr20:
   push byte 0
   push byte 20
   jmp isr_common_stub

; 21: Reserved
isr21:
   push byte 0
   push byte 21
   jmp isr_common_stub

; 22: Reserved
isr22:
   push byte 0
   push byte 22
   jmp isr_common_stub

; 23: Reserved
isr23:
   push byte 0
   push byte 23
```

```
    jmp isr_common_stub

; 24: Reserved
isr24:
    push byte 0
    push byte 24
    jmp isr_common_stub

; 25: Reserved
isr25:
    push byte 0
    push byte 25
    jmp isr_common_stub

; 26: Reserved
isr26:
    push byte 0
    push byte 26
    jmp isr_common_stub

; 27: Reserved
isr27:
    push byte 0
    push byte 27
    jmp isr_common_stub

; 28: Reserved
isr28:
    push byte 0
    push byte 28
    jmp isr_common_stub

; 29: Reserved
isr29:
    push byte 0
    push byte 29
    jmp isr_common_stub

; 30: Reserved
isr30:
    push byte 0
    push byte 30
    jmp isr_common_stub

; 31: Reserved
isr31:
    push byte 0
```

```
    push byte 31
    jmp isr_common_stub

; IRQs
global irq0
global irq1
global irq2
global irq3
global irq4
global irq5
global irq6
global irq7
global irq8
global irq9
global irq10
global irq11
global irq12
global irq13
global irq14
global irq15

; IRQ handlers
irq0:
        push byte 0
        push byte 32
        jmp irq_common_stub

irq1:
        push byte 1
        push byte 33
        jmp irq_common_stub

irq2:
        push byte 2
        push byte 34
        jmp irq_common_stub

irq3:
        push byte 3
        push byte 35
        jmp irq_common_stub

irq4:
        push byte 4
        push byte 36
        jmp irq_common_stub
```

```
irq5:
        push byte 5
        push byte 37
        jmp irq_common_stub

irq6:
        push byte 6
        push byte 38
        jmp irq_common_stub

irq7:
        push byte 7
        push byte 39
        jmp irq_common_stub

irq8:
        push byte 8
        push byte 40
        jmp irq_common_stub

irq9:
        push byte 9
        push byte 41
        jmp irq_common_stub

irq10:
        push byte 10
        push byte 42
        jmp irq_common_stub

irq11:
        push byte 11
        push byte 43
        jmp irq_common_stub

irq12:
        push byte 12
        push byte 44
        jmp irq_common_stub

irq13:
        push byte 13
        push byte 45
        jmp irq_common_stub

irq14:
        push byte 14
```

```
        push byte 46
        jmp irq_common_stub

irq15:
        push byte 15
        push byte 47
        jmp irq_common_stub
```

drivers folder

display.c

```
#include
#include "../kernel/util.h"

void clear_screen() {
for(int i = 0; i < (320*200); i++){
   putpixel(i, 0, 0x00);
}
}




void putpixel(int pos_x, int pos_y, unsigned char VGA_COLOR)
{
   unsigned char* location = (unsigned char*)0xA0000 + 320 * pos_y + pos_x;
   *location = VGA_COLOR;
}


void BlackBlock(int x, int y)
{
   x = x*10;
   y = y*10;
   int sqaure_size = 10;

    for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         putpixel(i, j, 0x00);
      }
    }
```

```
}

void Grayblock(int x , int y)
   {
   x = x*10;
   y = y*10;
   int sqaure_size = 10;


   for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         //light sides
         if(i - x == 0){
            putpixel(i, j, 0x1c);
         }
         else if(j - y == 0){
            putpixel(i, j, 0x1c);
         }
         //dark sides
         else if(i - x == 9){
            putpixel(i, j, 8);
         }
         else if(j - y == 9){
            putpixel(i, j, 8);
         }
         //fill
         else{
            putpixel(i, j, 7);
         }

      }
   }
   }



      void Pinkblock(int x , int y)
   {
   x = x*10;
   y = y*10;
   int sqaure_size = 10;

   for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         //light sides
         if(i - x == 0){
            putpixel(i, j, 0x3d);
```

```
        }
        else if(j - y == 0){
           putpixel(i, j, 0x3d);
        }
        //dark sides
        else if(i - x == 9){
           putpixel(i, j, 5);
        }
        else if(j - y == 9){
           putpixel(i, j, 5);
        }
        //fill
        else{
           putpixel(i, j, 13);
        }

     }
  }

}


void Yellowblock(int x , int y)
  {
  x = x*10;
  y = y*10;
  int sqaure_size = 10;

  for (int i = x; i < x + sqaure_size; i++) {
     for (int j = y; j < y + sqaure_size; j++) {
        //light sides
        if(i - x == 0){
           putpixel(i, j, 0x5c);
        }
        else if(j - y == 0){
           putpixel(i, j, 0x5c);
        }
        //dark sides
        else if(i - x == 9){
           putpixel(i, j, 0x2b);
        }
        else if(j - y == 9){
           putpixel(i, j, 0x2b);
        }
        //fill
        else{
           putpixel(i, j, 14);
```

```
        }

      }
   }

}


void Cyanblock(int x , int y)
   {
   x = x*10;
   y = y*10;
   int sqaure_size = 10;

   for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         //light sides
         if(i - x == 0){
            putpixel(i, j, 0x64);
         }
         else if(j - y == 0){
            putpixel(i, j, 0x64);
         }
         //dark sides
         else if(i - x == 9){
            putpixel(i, j, 3);
         }
         else if(j - y == 9){
            putpixel(i, j, 3);
         }
         //fill
         else{
            putpixel(i, j, 11);
         }

      }
   }

}



void Blueblock(int x , int y)
   {
   x = x*10;
   y = y*10;
   int sqaure_size = 10;
```

```
   for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         //light sides
         if(i - x == 0){
            putpixel(i, j, 0x36);
         }
         else if(j - y == 0){
            putpixel(i, j, 0x36);
         }
         //dark sides
         else if(i - x == 9){
            putpixel(i, j, 1);
         }
         else if(j - y == 9){
            putpixel(i, j, 1);
         }
         //fill
         else{
            putpixel(i, j, 9);
         }

      }
   }

}


void Orangeblock(int x , int y)
   {
   x = x*10;
   y = y*10;
   int sqaure_size = 10;

   for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         //light sides
         if(i - x == 0){
            putpixel(i, j, 0x2b);
         }
         else if(j - y == 0){
            putpixel(i, j, 0x2b);
         }
         //dark sides
         else if(i - x == 9){
            putpixel(i, j, 6);
         }
```

```
        else if(j - y == 9){
           putpixel(i, j, 6);
        }
        //fill
        else{
           putpixel(i, j, 0x2a);
        }

     }
  }

}


void Greenblock(int x , int y)
  {
  x = x*10;
  y = y*10;
  int sqaure_size = 10;

  for (int i = x; i < x + sqaure_size; i++) {
     for (int j = y; j < y + sqaure_size; j++) {
        //light sides
        if(i - x == 0){
           putpixel(i, j, 0x31);
        }
        else if(j - y == 0){
           putpixel(i, j, 0x31);
        }
        //dark sides
        else if(i - x == 9){
           putpixel(i, j, 2);
        }
        else if(j - y == 9){
           putpixel(i, j, 2);
        }
        //fill
        else{
           putpixel(i, j, 10);
        }

     }
  }

}
```

```
void Redblock(int x , int y)
   {
   x = x*10;
   y = y*10;
   int sqaure_size = 10;

   for (int i = x; i < x + sqaure_size; i++) {
      for (int j = y; j < y + sqaure_size; j++) {
         //light sides
         if(i - x == 0){
            putpixel(i, j, 0x40);
         }
         else if(j - y == 0){
            putpixel(i, j, 0x40);
         }
         //dark sides
         else if(i - x == 9){
            putpixel(i, j, 4);
         }
         else if(j - y == 9){
            putpixel(i, j, 4);
         }
         //fill
         else{
            putpixel(i, j, 12);
         }

      }
   }

}
```

keyboard.c:

```
#include <stdint.h>

/**
 * Read a byte from the specified port
 */
unsigned char port_byte_in(uint16_t port) {
```

```
    unsigned char result;
    /* Inline assembler syntax
     * !! Notice how the source and destination registers are switched from NASM !!
     *
     * '"=a" (result)'; set '=' the C variable '(result)' to the value of register e'a'x
     * '"d" (port)': map the C variable '(port)' into e'd'x register
     *
     * Inputs and outputs are separated by colons
     */
    asm("in %%dx, %%al" : "=a" (result) : "d" (port));
    return result;
}

void port_byte_out(uint16_t port, uint8_t data) {
    /* Notice how here both registers are mapped to C variables and
     * nothing is returned, thus, no equals '=' in the asm syntax
     * However we see a comma since there are two variables in the input area
     * and none in the 'return' area
     */
    asm("out %%al, %%dx" : : "a" (data), "d" (port));
}
```

kernel folder

util.h:

```
#pragma once
#include <stdint.h>

#define low_16(address) (uint16_t)((address) & 0xFFFF)
#define high_16(address) (uint16_t)(((address) >> 16) & 0xFFFF)
```

kernel.c:

```
#include "../cpu/Interrupt_Descriptor_Table.h"
#include "../cpu/Interrupt_Service_Routine.h"
#include "util.h"

#define TETRIMINO_AMOUNT 7
#define TETRIMINO_SIZE 16
```

```
#define BOARD_WIDTH 10
#define BOARD_HEIGHT 20

#define TETRIMINO_ROWS 4
#define TETRIMINO_COLS 4

int xStart = 3;
int yindex = 0;
int count = 0;
int placed = 0;
int TetriminoType;
unsigned int seed = 1134212;
int loop = 0;
int timer = 0;

int random_number(unsigned int *seed) {
   *seed = *seed * 11035245 + 13424;
   return (*seed >> 16) % 7;
}


char board[BOARD_HEIGHT * BOARD_WIDTH];

void init_board() {
for (int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
   board[i] = '.';
}}



char tetrimino[TETRIMINO_AMOUNT][TETRIMINO_SIZE] = {
   {'.', '.', 'X', '.',
    '.', '.', 'X', '.',
    '.', '.', 'X', '.',
    '.', '.', 'X', '.'    }, // I tetromino


   {'.', '.', 'X', '.',
    '.', 'X', 'X', '.',
    '.', 'X', '.', '.',
    '.', '.', '.', '.'    }, // Z tetromino


   {'.', '.', '.', '.',
    '.', 'X', 'X', '.',
    '.', 'X', 'X', '.',
```

```
    '.', '.', '.', '.'      }, // O tetromino


   {'.', 'X', '.', '.',
    '.', 'X', 'X', '.',
    '.', '.', 'X', '.',
    '.', '.', '.', '.'      }, // S tetromino


   {'.', '.', 'X', '.',
    '.', 'X', 'X', '.',
    '.', '.', 'X', '.',
    '.', '.', '.', '.'      }, // T tetromino


   {'.', '.', '.', '.',
    '.', 'X', 'X', '.',
    '.', 'X', '.', '.',
    '.', 'X', '.', '.'      }, // J tetromino


   {'.', '.', '.', '.',
    '.', 'X', 'X', '.',
    '.', '.', 'X', '.',
    '.', '.', 'X', '.'      }  // L tetromino
};


void placeTetrimino(int tetriminoType, int xPos, int yPos) {
   // Get the 1D array representing the tetrimino
   char* tetriminoArray = tetrimino[tetriminoType];

   // Calculate the starting index in the board 1D array
   int boardIndex = (yPos * BOARD_WIDTH) + xPos;

   // Loop through each row and column of the tetrimino and update the board
   for (int row = 0; row < TETRIMINO_ROWS; row++) {
      for (int col = 0; col < TETRIMINO_COLS; col++) {
         // Calculate the index of the current cell in the tetrimino array
         int tetriminoIndex = (row * TETRIMINO_COLS) + col;

         // Calculate the index of the current cell in the board array
         int boardOffset = (row * BOARD_WIDTH) + col;

         // If the current cell in the tetrimino is not empty, update the board
         if (tetriminoArray[tetriminoIndex] == 'X') {
            board[boardIndex + boardOffset] = 'X';
```

```
        }
      }
   }
}
// An array to store the new rotated tetrimino shapes
char Newtetrimino[TETRIMINO_AMOUNT][TETRIMINO_SIZE];

// The number of times the current tetrimino has been rotated
int rotations = 0;

// The current and new X positions of the tetrimino
int oldX = 0;
int newX = 0;

// A function to rotate the current tetrimino shape
void Rotate(){

   // Loop through the rows and columns of the tetrimino shape
   for(int y = 0; y < 4; y++){
      for(int x = 0; x < 4; x++){

         // Store the new rotated tetrimino shape in the Newtetrimino array
         Newtetrimino[TetriminoType][oldX] = tetrimino[TetriminoType][12 + y - (x * 4)];
         oldX++;
      }
   }

   // Reset the number of rotations and old X position of the tetrimino
   rotations = 0;
   oldX = 0;

   // Copy the new rotated tetrimino shape to the current tetrimino shape
   for(int i = 0; i < 16; i ++){
      tetrimino[TetriminoType][i] = Newtetrimino[TetriminoType][I];
   }
}

// A function to draw the current tetrimino on the game board
void draw_tetrimino() {
   int checkX = 0;
   int checkY = 0;

   // Loop through all the cells on the game board
   for(int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++){

      // Calculate the X and Y positions of the current cell
      checkX = i % 10;
```

```
    checkY = i / 10;

    // If the current cell is part of the tetrimino, draw it in the appropriate color
    if(board[i] == 'X'){
      if(TetriminoType == 0){
        Cyanblock(checkX + 11, checkY);
      }
      else if(TetriminoType == 1){
        Greenblock(checkX + 11, checkY);
      }
      else if(TetriminoType == 2){
        Yellowblock(checkX + 11, checkY);
      }
      else if(TetriminoType == 3){
        Redblock(checkX + 11, checkY);
      }
      else if(TetriminoType == 4){
        Pinkblock(checkX + 11, checkY);
      }
      else if(TetriminoType == 5){
        Blueblock(checkX + 11, checkY);
      }
      else if(TetriminoType == 6){
        Orangeblock(checkX + 11, checkY);
      }
    }
    // Otherwise, if the current cell is part of a completed line, draw it in the appropriate
color
    else if(board[i] == '1'){
      Cyanblock(checkX + 11, checkY);
    }
    else if(board[i] == '2'){
      Greenblock(checkX + 11, checkY);
    }
    else if(board[i] == '3'){
      Yellowblock(checkX + 11, checkY);
    }
    else if(board[i] == '4'){
      Redblock(checkX + 11, checkY);
    }
    else if(board[i] == '5'){
      Pinkblock(checkX + 11, checkY);
    }

    else if(board[i] == '6'){
      Blueblock(checkX + 11, checkY);
    }
```

```
        else if(board[i] == '7'){
            Orangeblock(checkX + 11, checkY);
        }

        else if(board[i] == '.'){
            BlackBlock(checkX + 11, checkY);
        }
    }

}

void start_kernel() {
    clear_screen();
    isr_install();
    asm volatile("sti");
    init_keyboard();


    for(int i = 0; i < 20; i++){
    Grayblock(21,i);
    }

    for(int i = 0; i < 20; i++){
    Grayblock(10,i);
    }

    init_board();


while (loop == 0){
TetriminoType = random_number(&seed);

while (placed == 0){

placeTetrimino(TetriminoType, xStart, yindex);

clear_line();

draw_tetrimino();

CheckIfTetriminoPlaced();
timer++;
if(timer % 800 == 0){
    shiftdown();
}

}
```

```
placed = 0;
GameEnd();

}

  }

void shiftleft(){
  // loop through the leftmost column of the board to count the number of blocks present in it
  for(int z = 0; z < BOARD_HEIGHT*BOARD_WIDTH; z = z + 10){
    if(board[z] == 'X'){
      count ++;
    }
  }
  // loop through each block of the current tetrimino to check if there is a block to the left of it on the board
  for(int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
    if(board[i] == 'X'){
      if(board[i-1] == '1' || board[i-1] == '2' || board[i-1] == '3' || board[i-1] == '4' || board[i-1] == '5' || board[i-1] == '6' || board[i-1] == '7'){
        count++;
      }
    }
  }
  // if there are no blocks in the leftmost column of the board and there are no blocks to the left of the current tetrimino on the board
  if(count == 0){
    // if the current tetrimino is not already at the left edge of the board
    if(board[0] != 'X'){
      // loop through each block on the board and move the current tetrimino one block to the left
      for (int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
        if(board[i] == 'X'){
          board[i] = '.';
          board[i-1] = 'X';
        }
      }
      // decrease the xStart variable by one to reflect the new position of the tetrimino
      xStart--;
    }
  }
  // reset the count variable to 0
  count = 0;
}
```

```
void shiftright(){
   // Loop through every 10th element in the board starting from index 9
   for(int z = 9; z < BOARD_HEIGHT*BOARD_WIDTH; z = z + 10){
      // If the element is 'X', increment the count
      if(board[z] == 'X'){
         count ++;
      }
   }
   // Loop through every element in the board
   for(int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
      // If the element is 'X'
      if(board[i] == 'X'){
         // Check the adjacent element to the right
         if(board[i+1] == '1' || board[i+1] == '2' || board[i+1] == '3' || board[i+1] == '4' ||
board[i+1] == '5' || board[i+1] == '6' || board[i+1] == '7'){
            // If the adjacent element to the right is a digit from 1 to 7, increment the count
            count++;
         }
      }
   }
   // If there are no 'X' elements that need to be shifted and the rightmost element is not
already an 'X'
   if(count == 0){
      if(board[19] != 'X'){
         // Loop through every element in the board
         for (int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
            // Replace any 'X' elements with a '.'
            if(board[i] == 'X'){
               board[i] = '.';
            }
         }
         // Increment the starting position of the 'X' elements
         xStart++;
      }
   }
   // Reset the count
   count = 0;
}


// Shift all elements down by one row
void shiftdown(){
   // Iterate over all elements on the board
   for (int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
      // If an element is 'X', replace it with '.'
```

```
      if(board[i] == 'X'){
        board[i] = '.';
      }
    }
    // Increase the index of the current row by one
    yindex ++;
}

// Shift all elements up by one row
void shiftup(){
    // Iterate over all elements on the board
    for (int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){
      // If an element is 'X', replace it with '.'
      if(board[i] == 'X'){
        board[i] = '.';
      }
    }
    // Decrease the index of the current row by one
    yindex --;
}



void CheckIfTetriminoPlaced() {
    // Loop through the last row of the board
    for (int i = 190; i < 200; i++) {
      if (board[i] == 'X') {
        // Mark all blocks on the board with '1' to indicate they are fixed
        for (int j = 0; j < BOARD_WIDTH * BOARD_HEIGHT; j++) {
          if (board[j] == 'X') {
            board[j] = TetriminoType + 49;
          }
        }

        // Set the 'placed' flag to indicate the current tetrimino is no longer being controlled
        placed = 1;

        // Set the starting position of the next tetrimino
        xStart = 3;
        yindex = 0;
      }
    }

    for(int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++){
      if(board[i] == 'X'){
        if(board[i + 10] == '1' || board[i + 10] == '2' || board[i + 10] == '3' || board[i + 10] ==
'4' || board[i + 10] == '5' || board[i + 10] == '6' || board[i + 10] == '7'){
```

```
        for(int j = 0; j < BOARD_HEIGHT * BOARD_WIDTH; j++){
            if(board[j] == 'X'){
                board[j] = TetriminoType + 49;
            }
        }
        placed = 1;

        // Set the starting position of the next tetrimino
        xStart = 3;
        yindex = 0;
        }
    }
  }
}

void clear_line() {
  int lines_cleared = 0;
  for (int row = BOARD_HEIGHT - 1; row >= 0; row--) {
    int block_count = 0;
    for (int col = 0; col < BOARD_WIDTH; col++) {
      if (board[row * BOARD_WIDTH + col] == '1' || board[row * BOARD_WIDTH + col] ==
'2' || board[row * BOARD_WIDTH + col] == '3' || board[row * BOARD_WIDTH + col] == '4'
|| board[row * BOARD_WIDTH + col] == '5' || board[row * BOARD_WIDTH + col] == '6' ||
board[row * BOARD_WIDTH + col] == '7') {
        block_count++;
      }
    }
    if (block_count == BOARD_WIDTH) {
      // Shift all the rows above this row down by one
      for (int i = row; i > 0; i--) {
        for (int j = 0; j < BOARD_WIDTH; j++) {
          board[i * BOARD_WIDTH + j] = board[(i - 1) * BOARD_WIDTH + j];
        }
      }
      // Clear the top row
      for (int j = 0; j < BOARD_WIDTH; j++) {
        board[j] = '.';
      }
      lines_cleared++;
      row++;
    }
  }

}
// This function calculates the number of empty spaces that the blocks can fall through
void fall() {
  int falldistance = 9999; // Initialize falldistance to a large value
```

```
   for (int i = 0; i < BOARD_HEIGHT * BOARD_WIDTH; i++) { // Iterate over all the blocks on
the board
      if (board[i] == 'X') { // If a block is found
         int hasfallen = 0; // Initialize the flag to check whether the block has fallen
         int findfalldistance = 0; // Initialize the distance that the block can fall
         int z = i; // Save the index of the block
         while (hasfallen == 0 && z < BOARD_HEIGHT * BOARD_WIDTH - 10) { // Loop while
the block hasn't fallen and there are still blocks below it
            if (board[z + 10] == '.' || board[z + 10] == 'X') { // Check if the space below the block
is empty or has a block
               findfalldistance++; // Increment the distance that the block can fall
               z += 10; // Move to the space below the block
            } else {
               hasfallen++; // Set the flag to indicate that the block has fallen
            }
         }
         if (falldistance > findfalldistance) { // Update the falldistance if the current block can
fall farther than previous blocks
            falldistance = findfalldistance;
         }
      }
   }
   // Move the blocks down by the calculated distance
   for(int i = 0; i < falldistance; i++){
      for (int i = 0; i < BOARD_HEIGHT*BOARD_WIDTH; i++){ // Loop over all the blocks on the
board
         if(board[i] == 'X'){ // If a block is found
            board[i] = '.'; // Remove the block from its current position
         }
      }
      yindex ++ ; // Move the blocks down one row
   }
}

// This function checks whether the game has ended
void GameEnd(){
   if(board[5] == 'X' || board[5] == '.'){ // Check if there is a block in the top row of the
center column or if it's empty
      // The game hasn't ended yet
   }
   else{
      placed++; // Increment the number of blocks that have been placed
      loop++; // Increment the loop counter
   }
}
```

MAKEFILE

```
C_SOURCES = $(wildcard kernel/*.c drivers/*.c cpu/*.c)
HEADERS = $(wildcard kernel/*.h  drivers/*.h cpu/*.h)
OBJ_FILES = ${C_SOURCES:.c=.o cpu/interrupt.o}


kernel.bin: boot/kernel_entry.o ${OBJ_FILES}
	x86_64-elf-ld -m elf_i386 -o $@ -Ttext 0x1000 $^ --oformat binary

os-image.bin: boot/mbr.bin kernel.bin
	cat $^ > $@

run: os-image.bin
	qemu-system-i386 -fda $<

echo: os-image.bin
	xxd $<


%.o: %.c ${HEADERS}
	x86_64-elf-gcc -g -m32 -ffreestanding -c $< -o $@ # -g for debugging

%.o: %.asm
	nasm $< -f elf -o $@

%.bin: %.asm
	nasm $< -f bin -o $@

%.dis: %.bin
	ndisasm -b 32 $< > $@

all: run
```

Testing plans

Bootloader tests

Test ID	Description	Input/Output data	Expected results
1	Setting up environment	Running assembly through QEMU in terminal	QEMU screen loads.

2	Setting up environment	“Hello World!” after running code through terminal	QEMU loads with message outputted from assembly code.
3	Verify bootloader loads disk correctly	Bootloader binary and disk image	Disk successfully loads
4	Verify bootloader handles errors	Disk read errors	Error message displayed
5	Verify null segment selector is set up correctly	GDT with null segment selector, with a message displaying its done	System successfully enters protected mode
6	Verify Code segment is setup correctly	GDT with code segment, with a message displaying its done	System successfully enters protected mode
7	Verify data segment is setup correctly	GDT with data segment, with a message displaying its done	System successfully enters protected mode
8	Verify GDT descriptor is setup correctly	GDT descriptor with correct size and address, with a message displaying its done	System successfully enters protected mode
9	Disable interrupts	-	Interupts are disabled
10	Load the GDT descriptor	gdt_descriptor	GDT descriptor is loaded
11	Set 32-bit mode	Cr0 register	32-bit mode set in cr0
12	Far jump by using a different segment	CODE_SEG:init_32bit	Processor enters 32-bit mode
13	Verify program does not exit after switch to 32-bit mode	-	Program stays running without exiting
14	Verify switch to 32-bit mode is successful	Code segment is switched	Message saying in 32 bit mode
15	Verify video mode is set correctly	AH = 0x00, AL = 0x13	Video mode is set to 13, with resolution 320x200 pixels.
16	Bootloader jumps to kernel and can be coded in C	Test a subroutine written in C and implement it in the main kernel	The bootloader successfully loads the kernel and jumps to it, allowing

		function, in this case plotting a green pixel in the middle of the screen	for C code to be executed in the kernel.

Interrupt tests

Test ID	Description	Input/Output data	Expected results
17	Setup I/O ports	Switch to Text mode temporarily and make sure Ports coded correctly	Allow interfacing with I/O ports
18	Test ISR installation function	-	Ensure that all ISR gates are set correctly and are mapped to their corresponding interrupt handlers.
19	Test IRQ installation function	-	Ensure that all IRQ gates are set correctly and are mapped to their corresponding interrupt handlers.
20	Test PIC remapping function	-	Ensure that the PIC has been remapped correctly to avoid conflicts with the interrupts.
21	Test exception handler	Trigger an interrupt exception	A blue screen will appear showing there is a invalid input
22	Test division by zero exception	Divide a used variable by zero	A blue screen will appear showing there is a invalid input
23	Test debug exception	Uses the int3 instruction in your code to trigger a debug exception and examine the state of the program using a debugger.	A blue screen will appear showing there is a invalid input
24	Test non-maskable interrupt exception	simulate a hardware malfunction like disconnecting the keyboard	A blue screen will appear showing there is a invalid input

25	Test breakpoint exception	when the processor reaches a instruction, it will halt program execution.	A blue screen will appear showing there is a invalid input
26	Test overflow exception	adding two large positive numbers when using a signed integer data type	A blue screen will appear showing there is a invalid input
27	Test out of bounds exception	Attempt to access an array out of its bounds	A blue screen will appear showing there is a invalid input
28	Test invalid opcode exception	Execute an invalid opcode instruction	A blue screen will appear showing there is a invalid input
29	Test coprocessor not available exception	Attempt to use a floating-point instruction when there is no floating-point coprocessor available	A blue screen will appear showing there is a invalid input
30	Test double fault exception	Simulate a double fault exception	A blue screen will appear showing there is a invalid input
31	Test coprocessor segment overrun exception	execute a FPU instruction with a bad or invalid address in the FPU segment.	A blue screen will appear showing there is a invalid input
32	Test bad TSS exception	Attempt to access a segment that is not present in memory	A blue screen will appear showing there is a invalid input
33	Test stack fault exception	Attempt to access an invalid memory address while pushing or popping from the stack	A blue screen will appear showing there is a invalid input
34	Test General Protection Fault Exception	Trigger a general protection fault exception	A blue screen will appear showing there is a invalid input
35	Test Page Fault Exception	attempt to access an invalid memory address	A blue screen will appear showing there is a invalid input
36	Test Unknown Interrupt Exception	Trigger an unknown interrupt	A blue screen will appear showing

			there is a invalid input
37	Test Coprocessor Fault Exception	Trigger a coprocessor fault exception	A blue screen will appear showing there is a invalid input
38	Test Alignment Check Exception	Trigger an alignment check exception	A blue screen will appear showing there is a invalid input
39	Test Machine Check Exception	Trigger a machine check exception	A blue screen will appear showing there is a invalid input

Tetris Tests

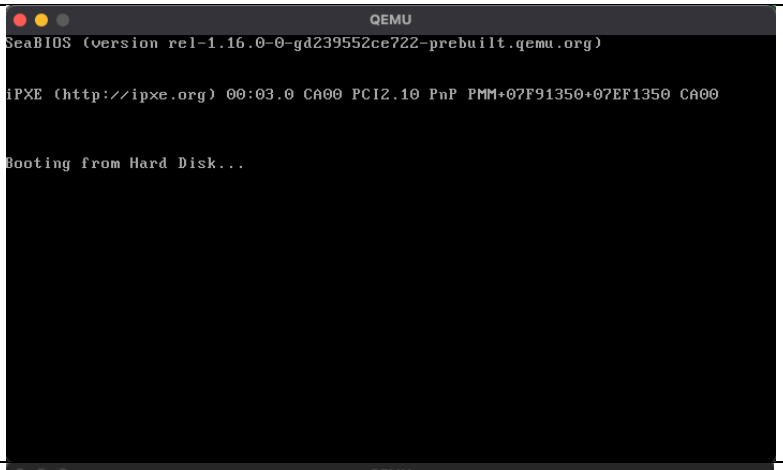
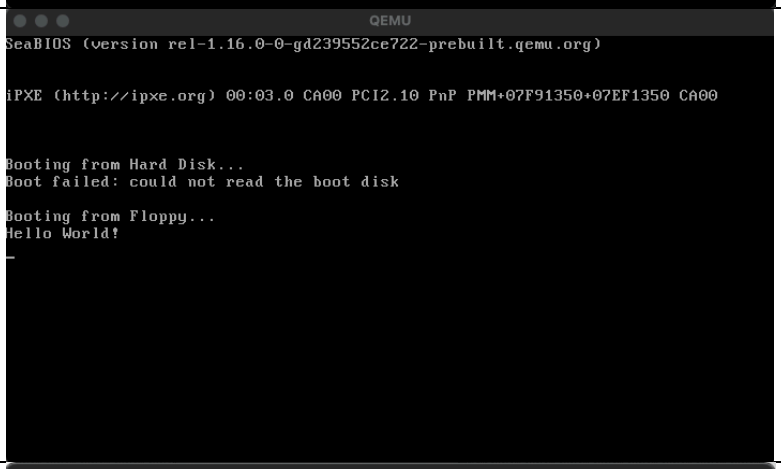
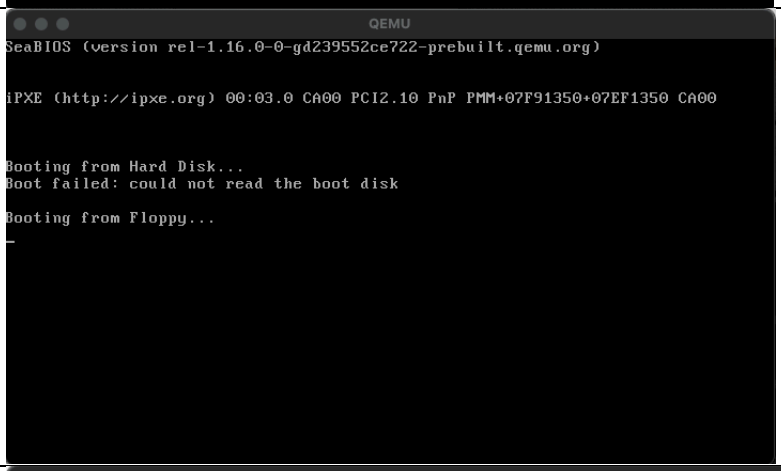
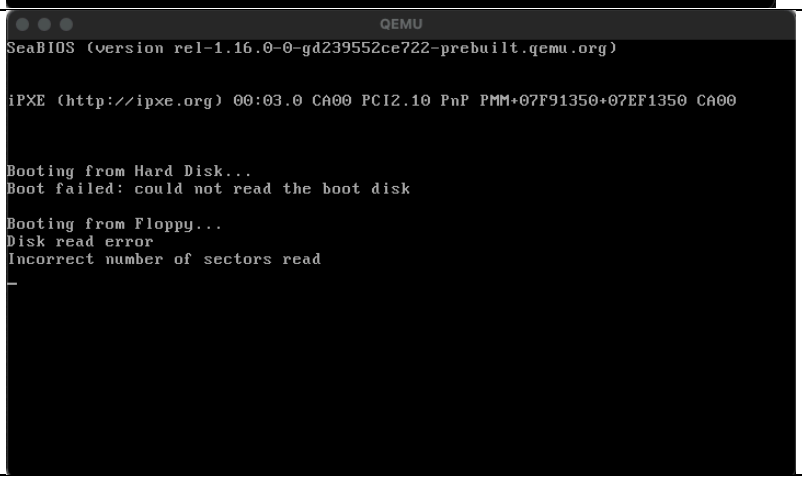
Test ID	Description	Input/Output Data	Expected Results
40	Make a 10x10 pixel block of each colour required for tetris on the screen	-	Output all blocks necessary including all blocks for tetris and the border
41	Design the borders for tetris	-	Outputs the borders for there tetris board
42	Test all tetrominos have the correct geometrical shape	-	Outputs the tetrominoes and they should be the correct shape for their colour
43	Test if tetrominoes represented correctly in the game board	-	Tetrominoes displayed correctly within the borders of the board
44	Test if the game initialises on an empty board	-	Tetromino is spawned on the board when game started
45	Test wether the tetromino can be moved left with the left arrow key input	Left arrow key	Tetromino should move one square to the left

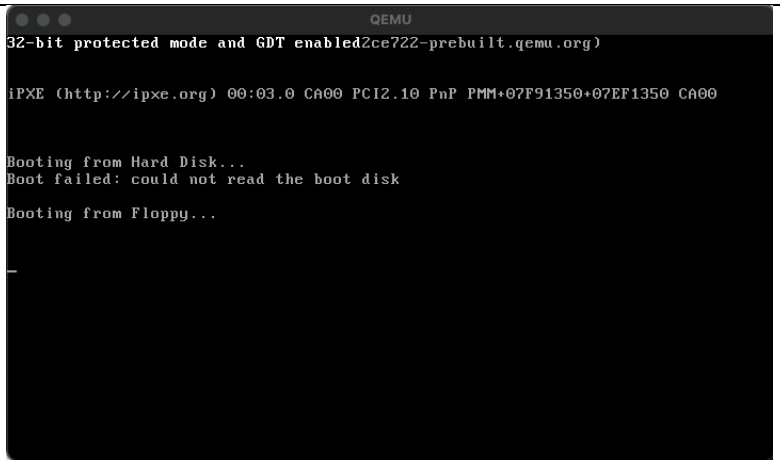
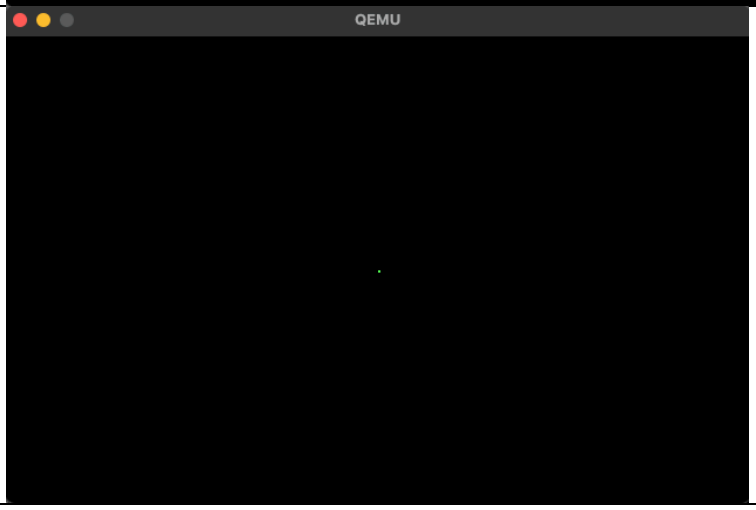
46	Test wether the tetromino can be moved right with the right arrow key input	Right arrow key	Tetromino should move one square to the right
47	Test wether the tetromino can be moved down with the down arrow key input	Down arrow key	Tetromino should move one square down
48	Test wether the tetromino can be rotated clockwise by 90 degrees with the up arrow key input	Up arrow key	Tetromino should rotate 90 degrees clockwise
49	Test wether the tetromino can be dropped to the bottom	Space bar key	Tetromino should fall to the lowest position possible in that column
50	Test wether the row is cleared when its filled with tetrominoes	-	The row should be cleared and any tetrominoes above should move down one row.
51	Test wether the game ends when a tetromino is placed on the top of the game board	-	The game should stop not spawning any new tetrominoes and user should not be able to control any tetrominoes
52	Test wether a tetromino is spawned when the previous one is placed	-	Tetromino spawned in correct spawn location when previous tetromino placed
53	Test wether the new tetromino spawned is random	-	Tetrominoes should spawn randomly
54	Test if tetrominoes cant move through other tetrominoes	-	Tetrominoes should stack ontop of placed tetrominoes or stack ontop of the bottom of the board not passing through any already placed tetrominoes

55	Test wether tetrominoes cannot exit the game board	-	Tetrominoes should stay within the bounds of the board and not pass the gray border
56	Test wether the tetrominno falls at a regular interval	-	Tetromino should fall at a regular interval
57	Test wether a tetromino can rotate fully 360 degrees correctly	-	Tetromino should rotate 90 degrees correctly once rotated 360 degrees.

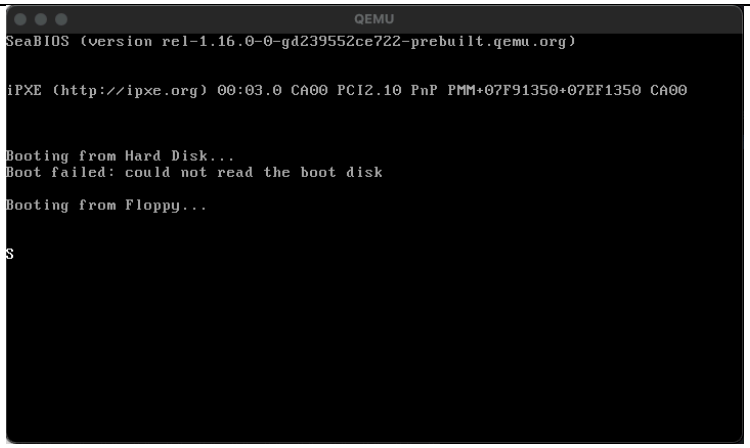
Bootloader Results

Test ID	Result	Description	Reflection

1		Environemtn set QEMU working properly	Task successful
2		QEMU loads with message outputted from assembly code.	Task successful
3		No “Disk read error” displayed	Task successful
4		Changed boot code so that disk loads incorrectly	Task successful

5-14		Edited mbr.asm so a message will display once entered 32 bit protected mode with GDT implemented. Message displayed in white text at top of the screen. QEMU doesn't close stays running in 32-bit mode.	Task successful
15		Kernel leaves standard text mode, goes to 320x200 resolution with no text	Task successful.
16		Kernel executing a C-written function, plotting a green pixel in the middle of the screen	Task successful

Interrupts Results

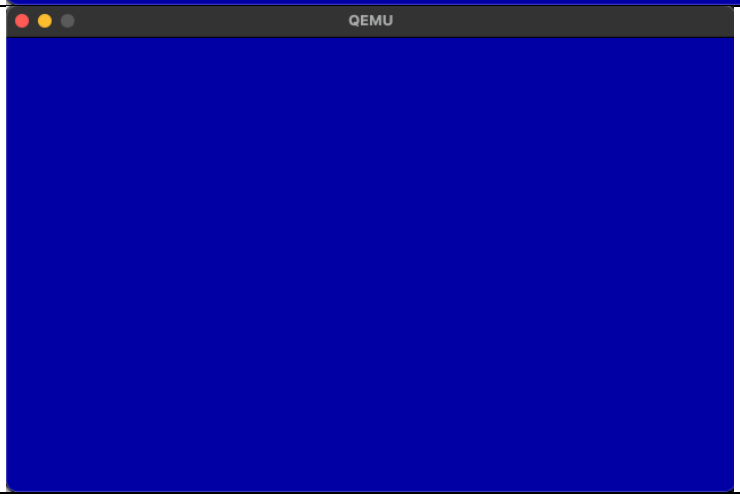
Test ID	Result	Description	Reflection
17		uses I/O ports to communicate with the VGA controller and get the current cursor position. It then calculates the offset into the VGA text buffer for the cursor position and writes a character to that position, in this case the characeter "S”	Task successful
18-20		Call the function to install the ISR and trigger a interrupt by any interrupt exception. First picture shows QEMU after kernel loaded second image shows what happens after a key is pressed calling a subroutine tat includes	Task successful

		division by zero	
21-22		Triggers a key that will include a division by zero to a variable	Task successful
23		Uses the int3 instruction in your code to trigger a debug exception and examine the state of the program using a debugger. Resulting in a interrupt error screen appearing	Task successful

24		Disconnects hardware so a interrupt exception is triggered resulting in a blue screen halting the system	Task successful
25		when the processor reaches a instruction, it will halt program execution, resulting in a blue screen	Task successful
26		2 large signed integers added resulting in a blue screen	Task successful

27		Out of bounds array attempted resulting in a blue error screen	Task successful
28		Executed an invalid opcode instruction, resulting in a blue error screen	Task successful
29		Attempts to use a floating-point instruction when there is no floating-point coprocessor available, resulting in a blue error screen	Task successful

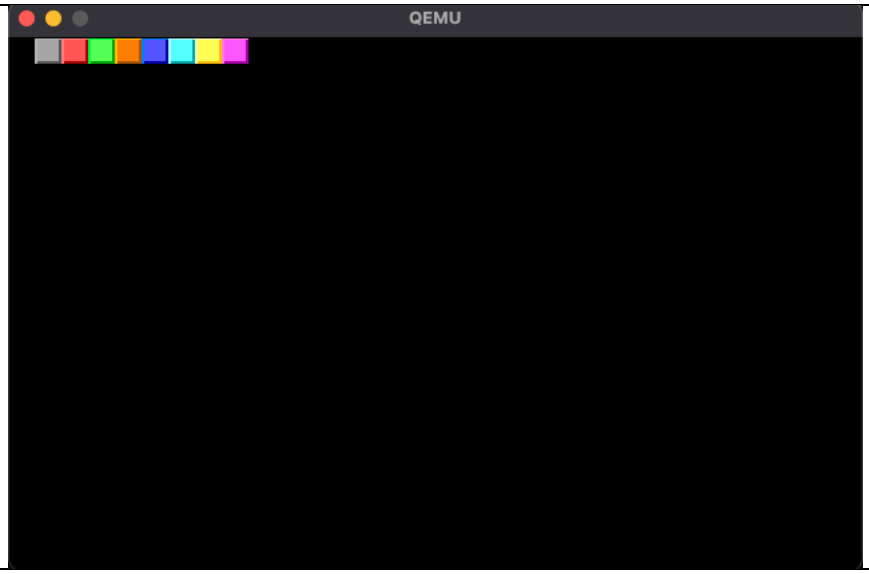
30		Simulates a double fault exception, resulting in a a blue error screen	Task successful
31		executes a FPU instruction with a bad or invalid address in the FPU segment, resulting in a blue error screen	Task successful
32		Attempts to access a segment that is not present in memory, resulting in a blue error screen	Task successful

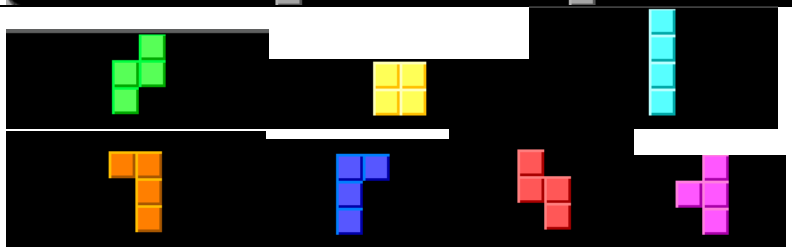
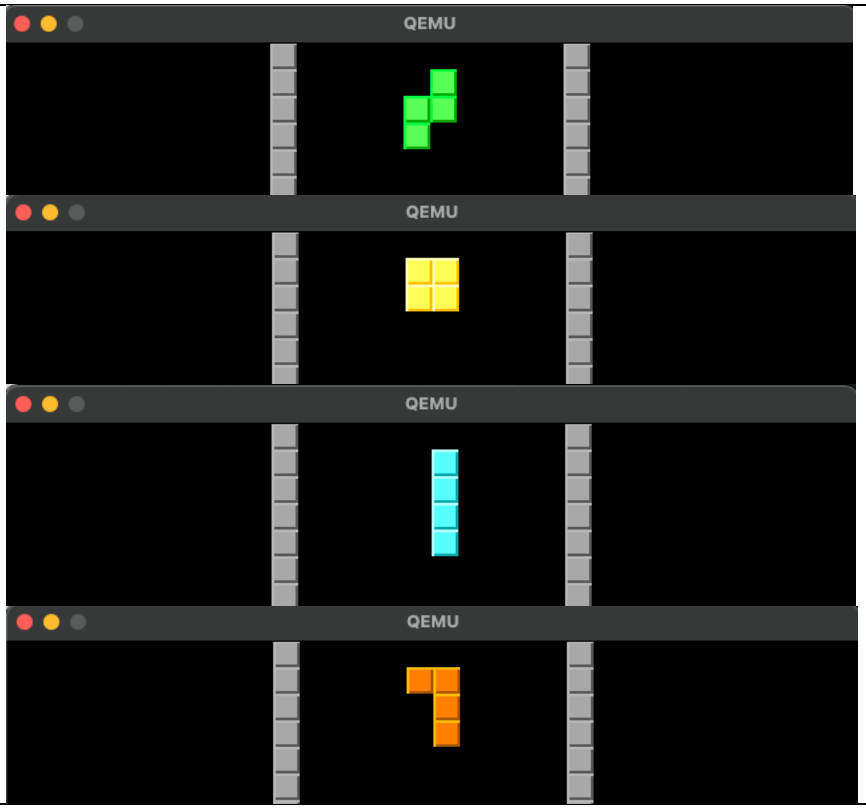
33		accesses an invalid memory address while pushing or popping from the stack, resulting in a blue error screen	Task successful
34		Triggered a general protection fault exception, resulting in a blue error screen	Task successful
35		accesses an invalid memory address, resulting in a blue error screen	Task successful

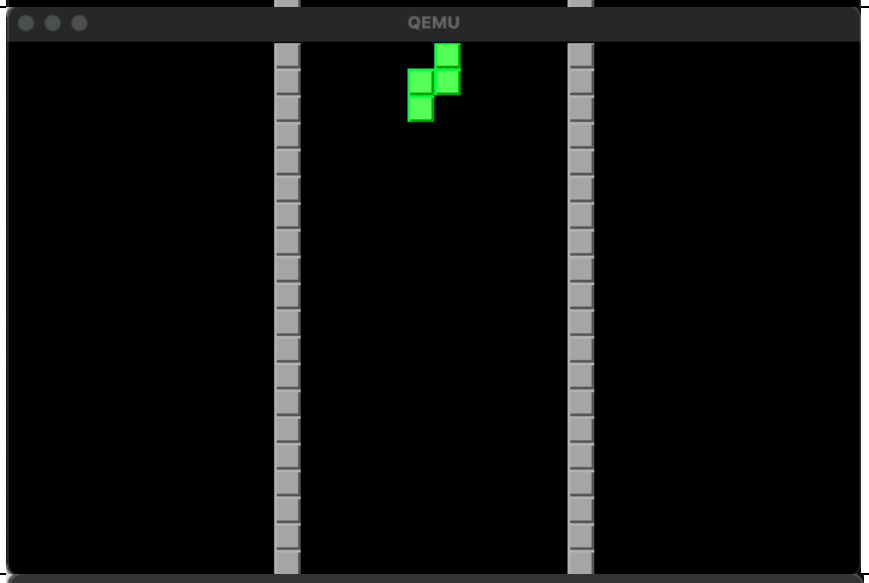
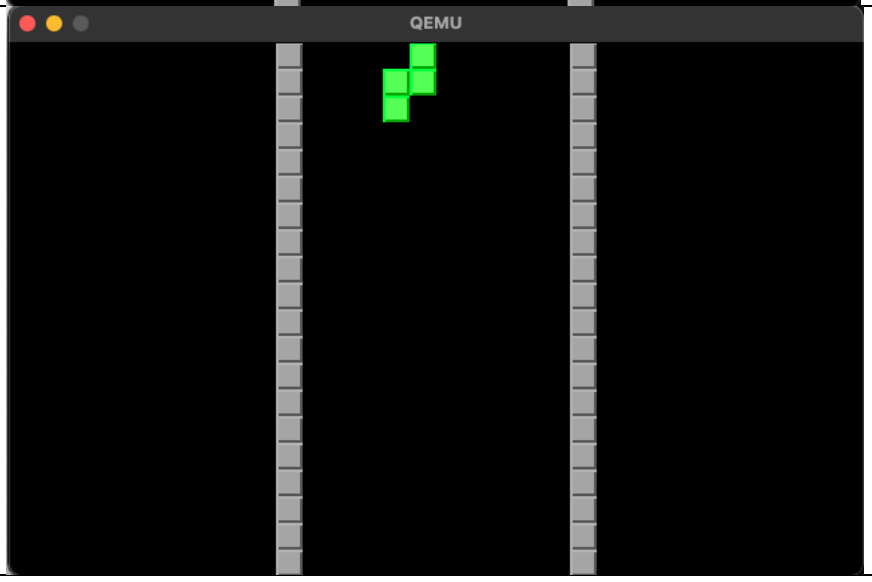
36		Triggered a unkown interrupt, resulting in a blue screen	Task successful
37		Triggered a coprocessor fault exception, resulting in a blue error screen	Task successful
38		Triggered an alignment check exception, resulting in a blue error screen	Task successful

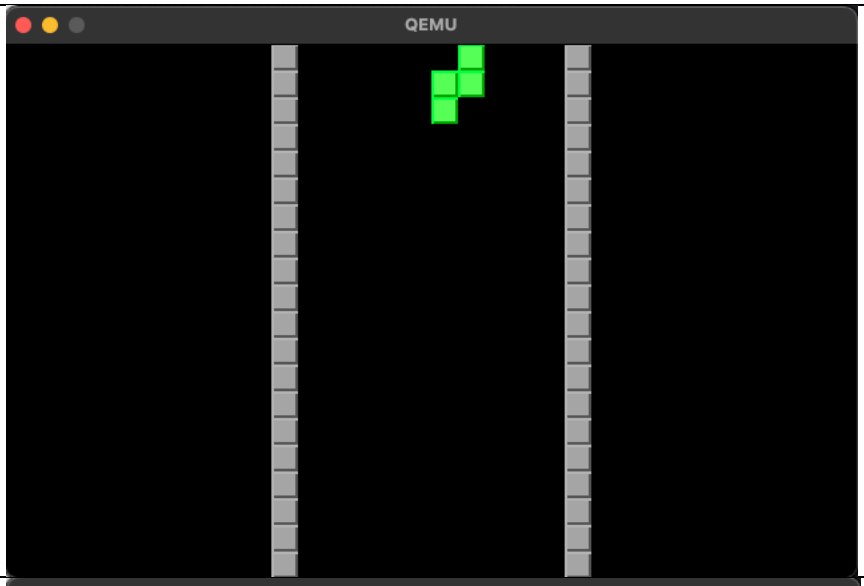
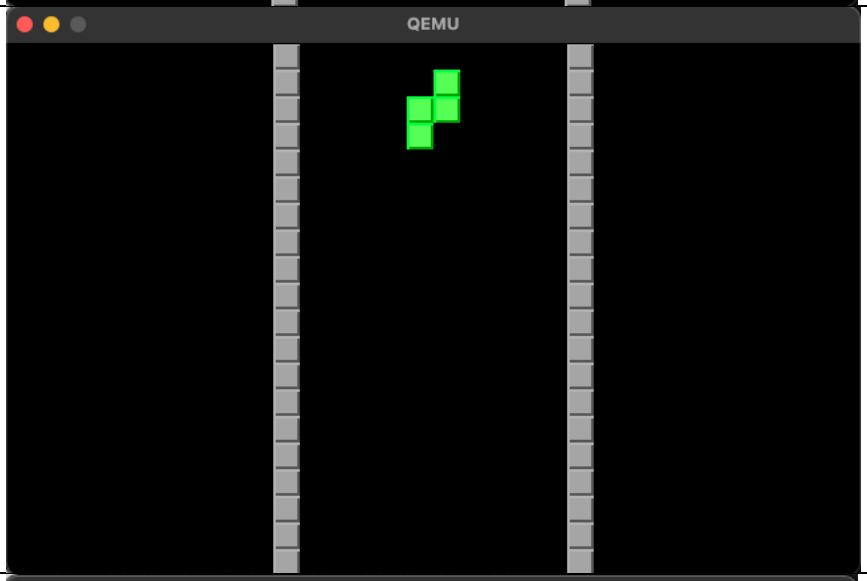
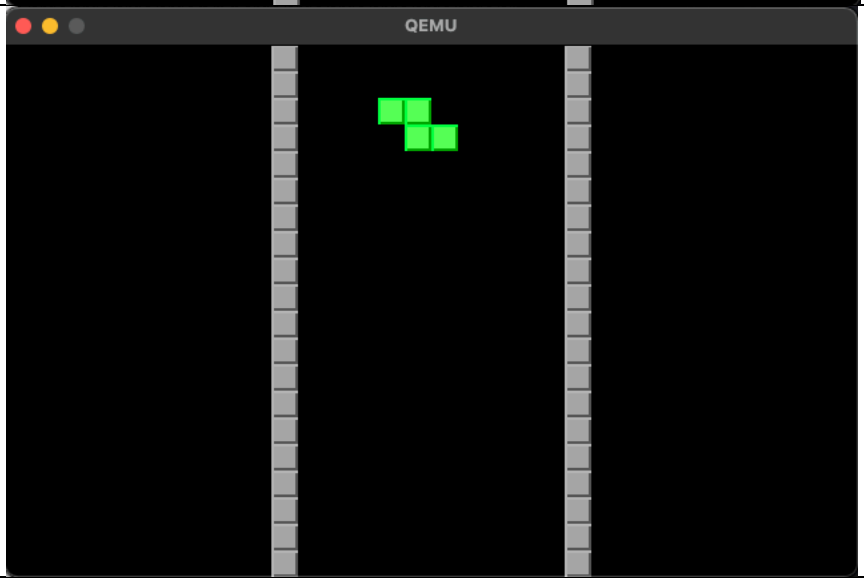
39		Triggered a machine check exception, resulting in a blue error screen	Task successful

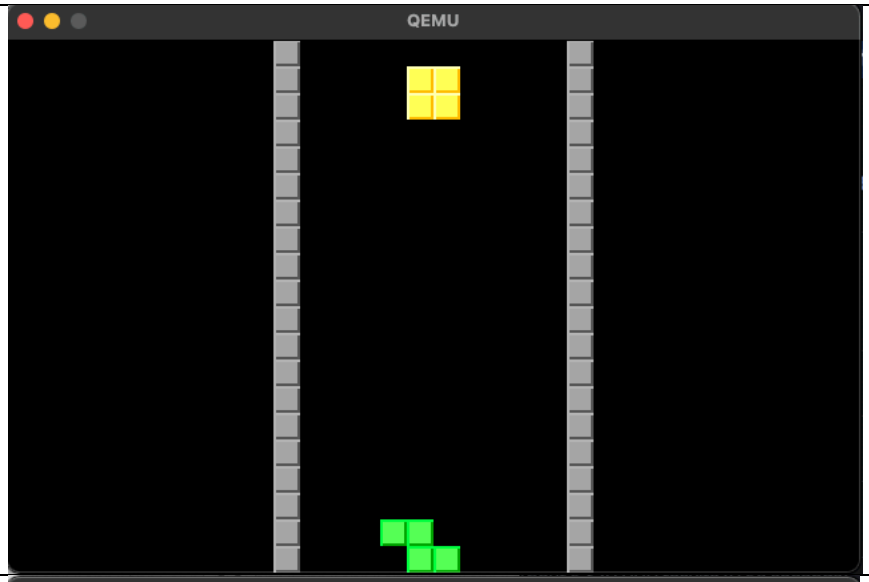
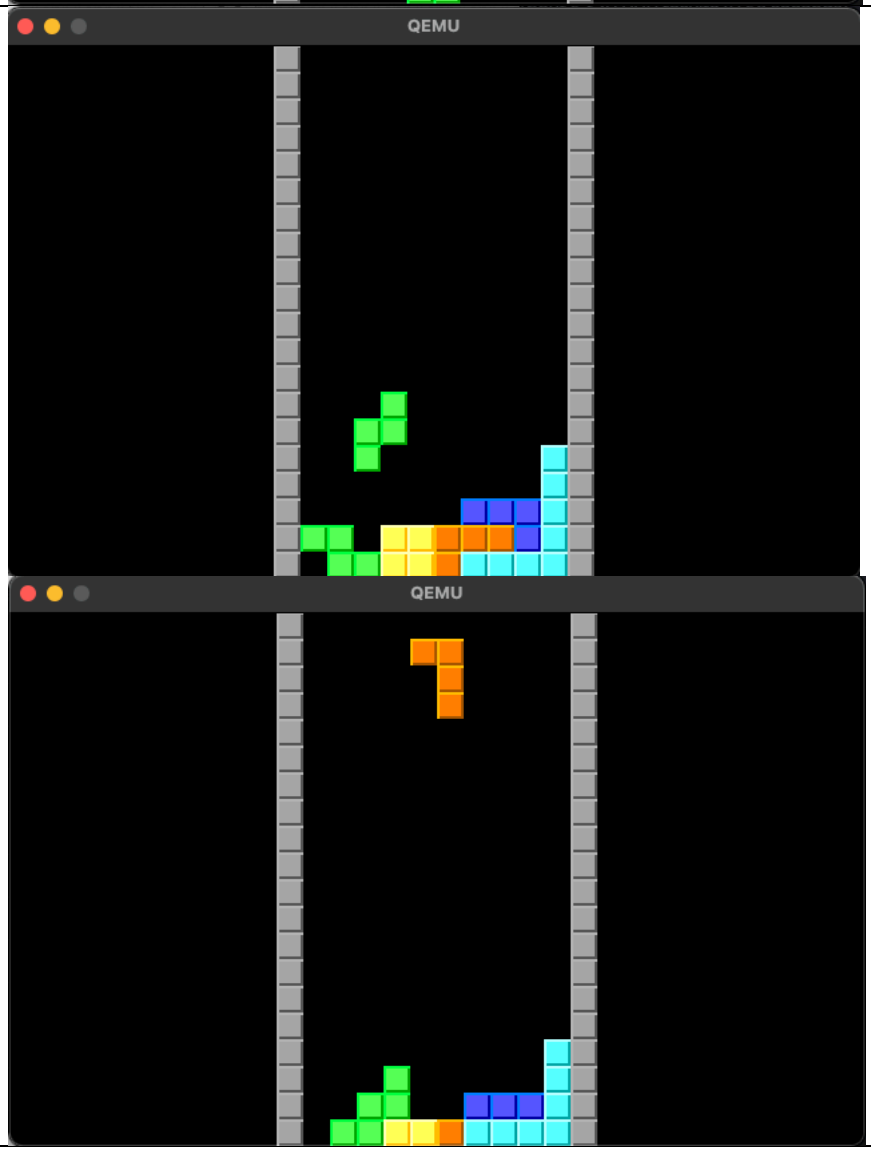
Tetris Results

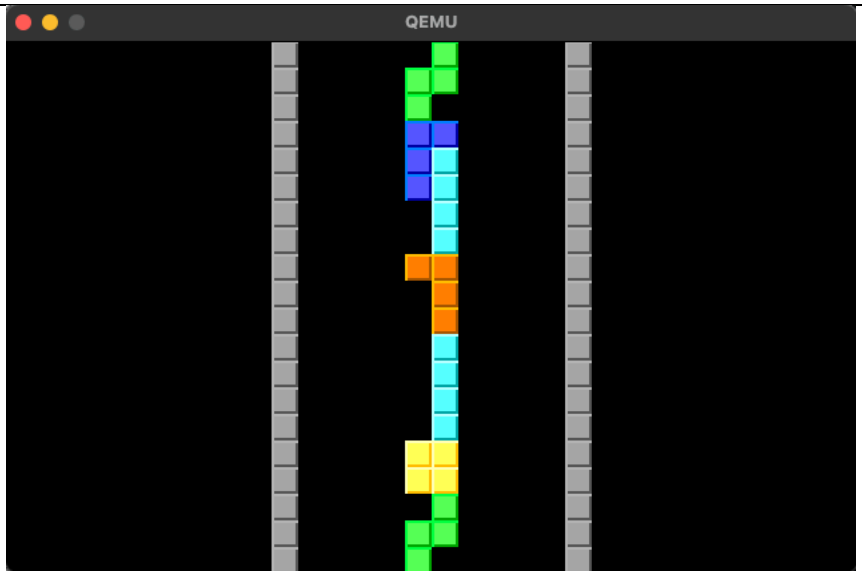
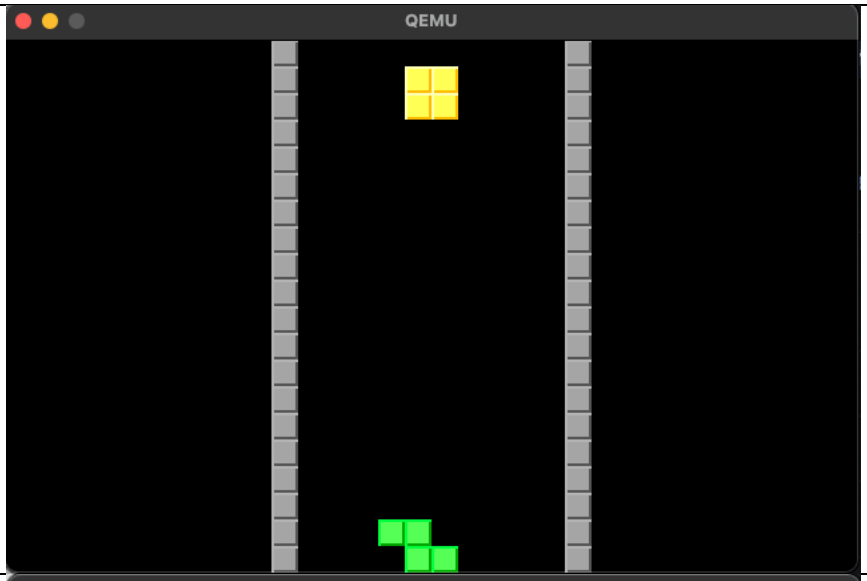
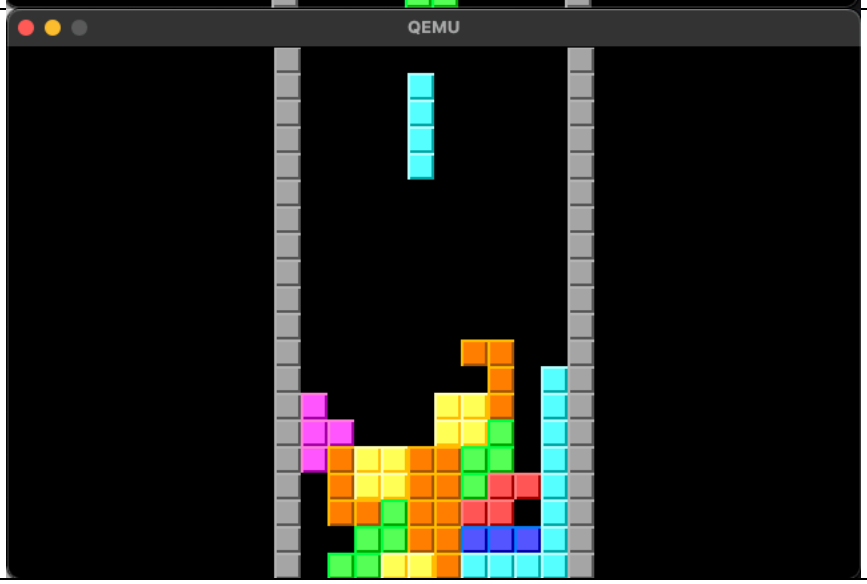
Test ID	Result	Description	Reflection
40		Outputs all coloured blocks necessary including blocks for tetris and the border	Task successful

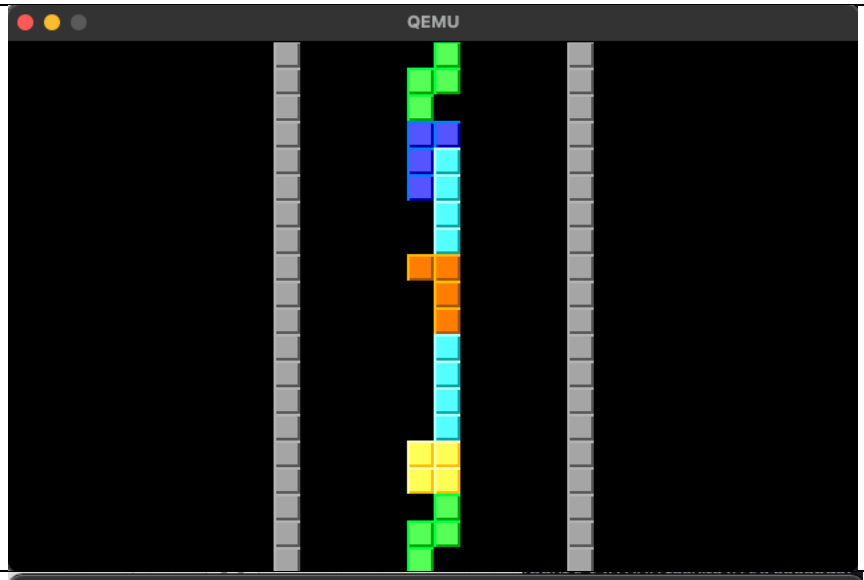
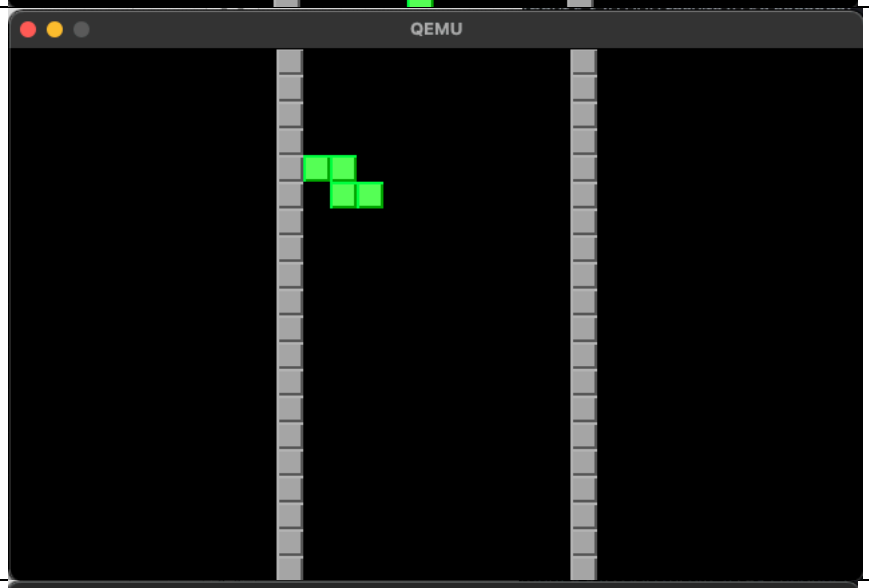
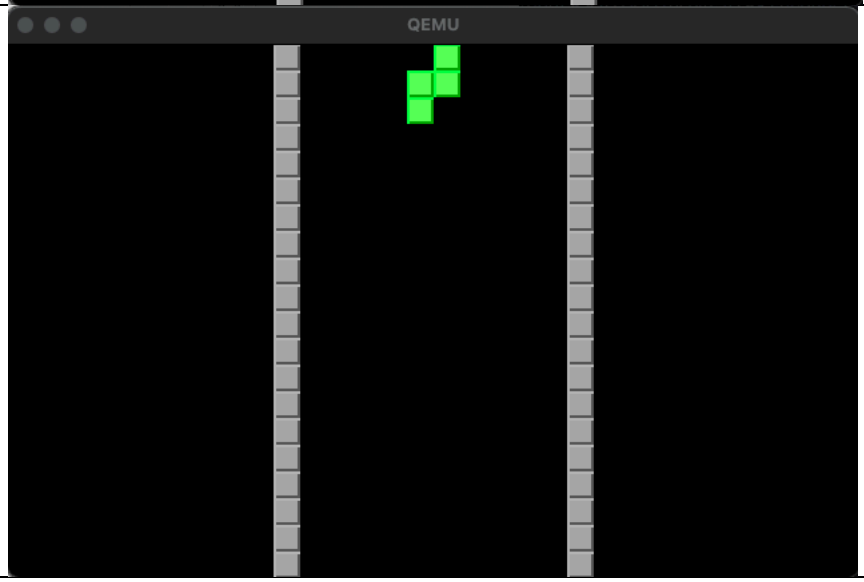
41		Outputted borders for tetris board of each 20 blocks high	Task successful
42		All tetrominoes outputted have the correct shape and colour	Task successful
43		All tetrominoes outputted within the borders have the correct shape and colour	Task successful

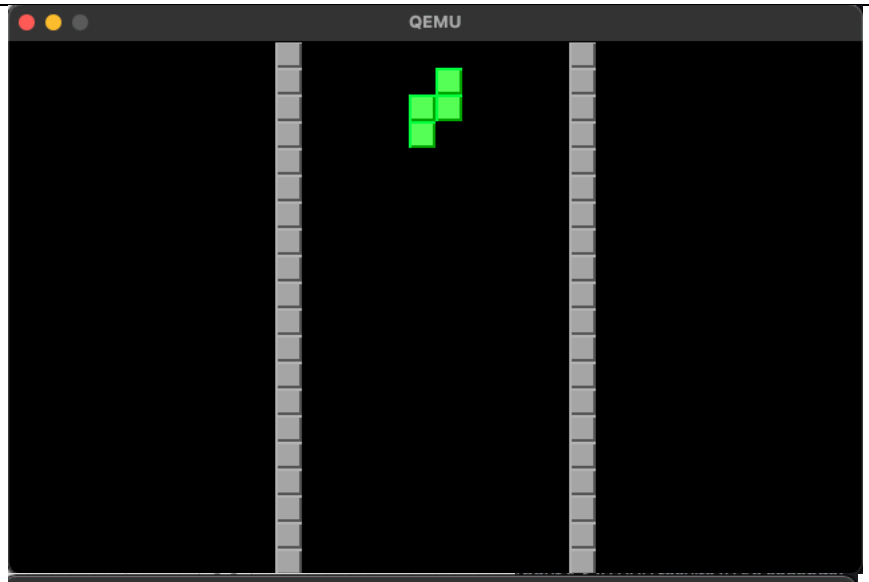
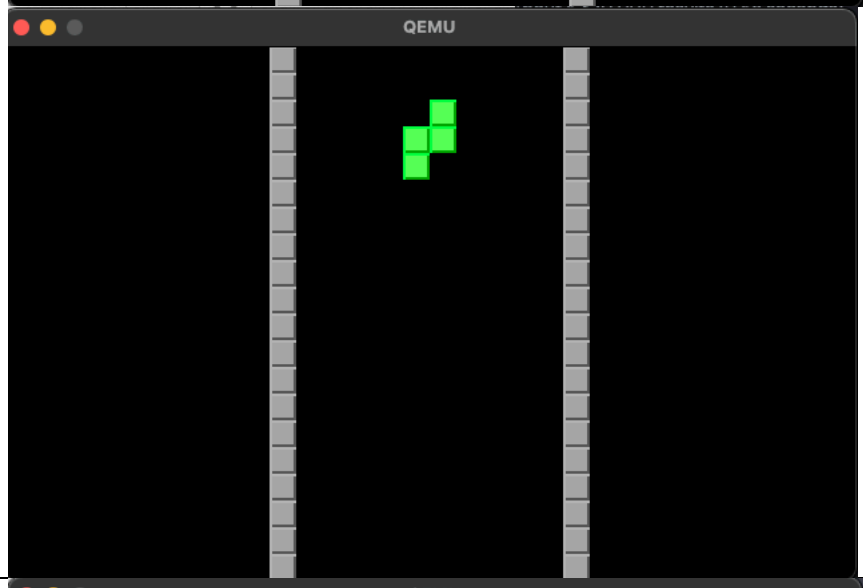
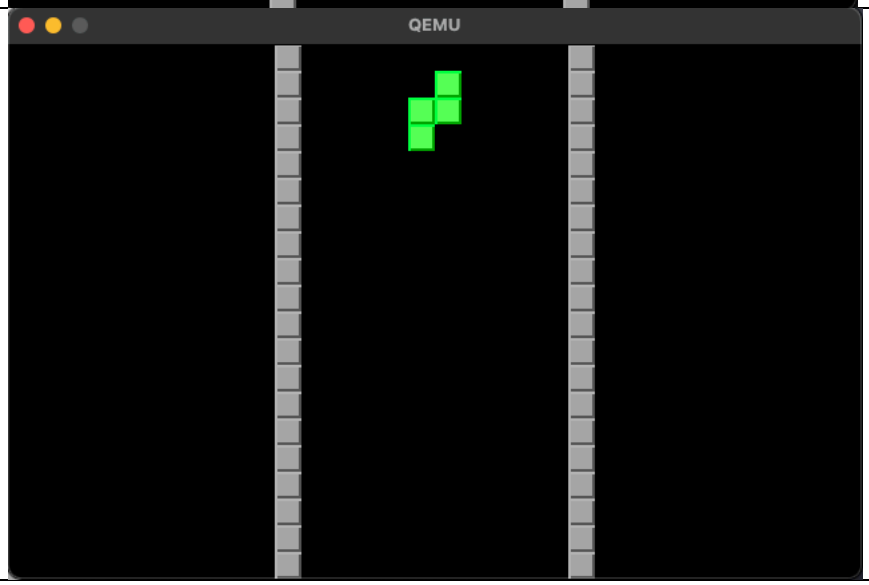
			
T44		Tetromino is spawned on the board when game started, no blank board	Task successful
45		When left arrow key pressed tetromino moves one square to the left	Task successful

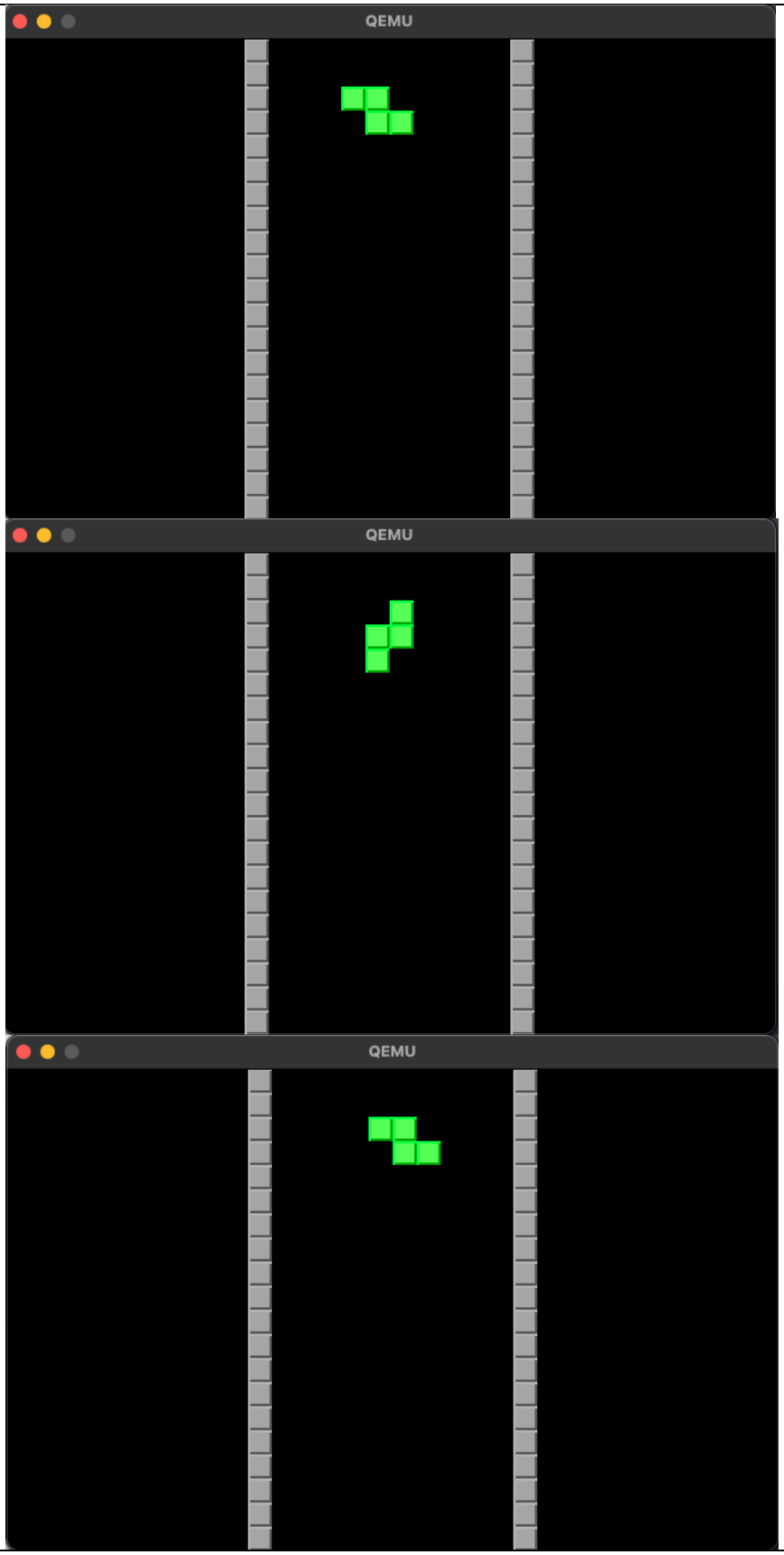
46		When right arrow key pressed tetromino moves one square to the right	Task successful
47		When down arrow key pressed tetromino moves one square down	Task successful
48		When up arrow key pressed tetromino rotates 90 degrees clockwise	Task successful

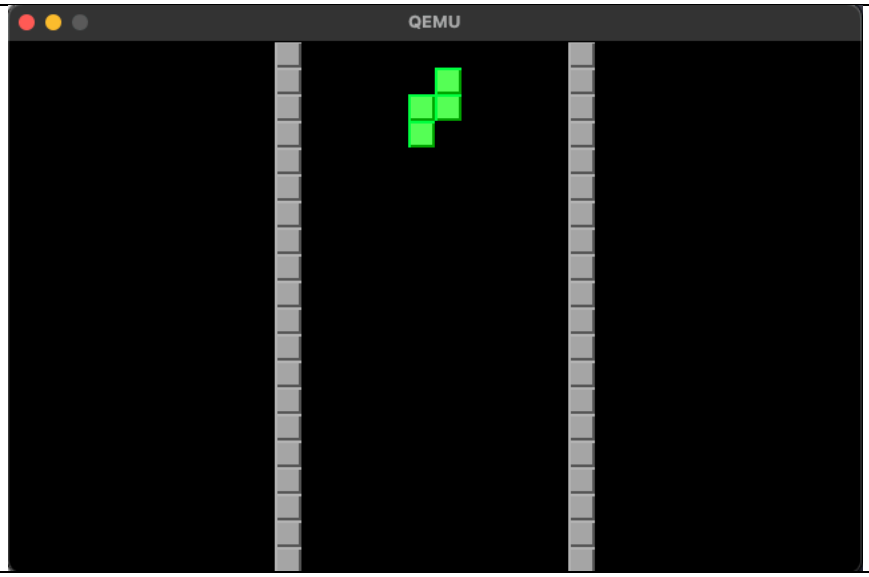
49		When space bar key pressed tetromino falls to the lowest possible place in that row	Task successful
50		First image shows the tetromino falling into a place where will fill out a row, the second image shows that the row was cleared and the rows above moved down by one square	Task successful

51		Once a tetromino is in the location of a spawning tetromino the game stops and the user cannot move any tetrominioes and keyboard inputs do not affect the board	Task successful
52		Once a tetromino is places a new tetromino is spawned at the starting position	Task successful
53		Tetrominoes spawn at random	Task successful

54		Tetrominoes stack on top of each other and a tetromino will fall to the lowest location on the screen if no other tetrominoes are placed underneath it	Task successful
55		The tetromino cannot leave either side of the border and will always stay in between them	Task successful
56		The tetromino falls at regular interval and will eventually be placed	Task successful

	 		
57		Tetromino rotates correctly 360 degrees with 4 rotations of 90 degrees clockwise	Task successful


QEMU
QEMU
QEMU


QEMU

Evaluation

Introduction

Writing a kernel for Tetris was a challenging and rewarding experience for me. Overall, I am happy with the game's mechanics, how the kernel functions, and how I used my objectives to design an operating kernel that runs Tetris, with objectives created to a great standard. Still, there are some areas where I could have done better to please the end user further and provide a better educational perspective.

While developing the kernel, I focused heavily on ensuring that the game mechanics were implemented correctly and that the kernel functioned smoothly. However, I realise there is more to creating a successful game than just the technical aspect. In retrospect, I could have placed more emphasis on the user experience, such as adding a scoring system, sound effects, and a main menu, and also provided some knowledge on interrupts if an interrupt exception occurs instead of just crashing the system. These features would not only enhance the user's enjoyment of the game but also provide a more comprehensive educational experience and gain attraction to low-level programming around the software engineering community.

How objectives were met

1)
a) Read from disk, which involves reading the bootloader from the disk into memory and transferring control to it. To achieve this objective, I wrote code that would read the bootloader data from the storage device, validate its integrity, and store it in memory in a location accessible to the CPU. This ensured that the bootloader would be properly loaded into memory and ready for execution. I also designed error handling and recovery techniques to ensure that any faults or corruption in the bootloader data would be found and dealt with properly. This enhanced the kernel's overall robustness and dependability by ensuring that the boot process would proceed despite potential problems.

b) Set up a GDT to allow the system to run in protected mode.
I specified the structure of the memory segments utilised by the CPU to build up the GDT. I ensured that the GDT accurately reflected the various memory segments used by the system. This enabled the CPU to control memory access and implement protection levels effectively and securely. After initialising the GDT, I made it available to the CPU and ensured it was appropriately configured to control memory access and enforce protection levels. To initialise the GDT and ensure it was operating as planned, this included writing and running the relevant code.

c) Switch to 32-bit protected mode.

To accomplish this objective, I first disabled interrupts to ensure a smooth transition to protected mode. I then loaded a flat memory model to prepare the system securely, which is necessary for memory management and protection. Next, I set the protected mode bit in the Control Register 0 (CR0) to signal the CPU to switch to protected mode. Finally, I jumped to protected mode, enabling the system to operate in a secure method of operation.

d) Write the Master Boot Record.
To write the Master Boot Record, I developed code to locate the operating system, load it into memory, store the boot drive information, set up the stack and switch to 32-bit protected mode. This ensured the system was properly configured and ready to run the Tetris game.

e) Create a kernel entry point
I established a well-known starting point for the kernel to be executed and prepare for C-written code.

2)
a) Set up the IDT (Interrupt Descriptor Table) to handle keyboard interrupts.

By setting up the IDT (Interrupt Descriptor Table) to handle keyboard interrupts, I have ensured that the user's keyboard inputs are properly received and processed. The IDT is set up in memory with sufficient space allocated to accommodate all of the interrupt vectors and is initialised with default values as necessary. The structure of an IDT entry is defined with fields for the routine handler address, segment selector, flags, and other essential information. The keyboard interrupt handler routine is registered in the IDT, mapping the interrupt vector for the keyboard to the routine handler address. Finally, the Programmable Interrupt Controller (PIC) is configured to send keyboard interrupts to the processor, ensuring they are properly handled.

This implementation of the IDT ensures that the system accurately captures and processes the user's keyboard inputs. By testing the IDT, I have confirmed that it correctly handles keyboard interrupts and routes them to the appropriate handler routine. This guarantees that keyboard inputs are processed and dealt with correctly, allowing the user to control the game with their keyboard inputs.

b) Write the ISR (Interrupt Service Routine) for the keyboard interrupt to process the incoming scan codes.

By defining the keyboard interrupt vector in the Interrupt Descriptor Table (IDT), creating the Interrupt Service Routine (ISR) for the keyboard interrupt, and extracting the scan codes from the keyboard interrupt signal, I have ensured that the system accurately captures and processes the user's keyboard inputs. Implementing logic to interpret the extracted scan codes and translate them into meaningful inputs allows the user's inputs to control the Tetris pieces. The system state is then updated based on the processed keyboard inputs,

allowing the Tetris game to function as intended. Finally, the previous context is restored after the ISR is complete, allowing the processor to continue its normal operation.

This implementation of the ISR and scan code processing ensures that the system accurately captures and processes the user's keyboard inputs. In addition, this guarantees that keyboard inputs are processed and dealt with correctly, allowing the user to control the game with their keyboard inputs.

c) Map the scan codes to the corresponding keyboard keys using an IRQ (Interrupt Request) handler.

By determining the IRQ handler structure, implementing the IRQ handler to process scan codes and map to keyboard keys, verifying the IRQ handler correctness, and integrating the IRQ handler with the rest of the keyboard input handling system, I have ensured that the user's keyboard inputs are accurately captured and processed by the system. Furthermore, implementing an IRQ handler allows the scan codes to be properly mapped to their corresponding keyboard keys, enabling the user's inputs to control the Tetris pieces as intended.

This implementation of the IRQ handler ensures that the system accurately captures and processes the user's keyboard inputs. In addition, this guarantees that keyboard inputs are processed and dealt with correctly, allowing the user to control the game with their keyboard inputs.

d) Implement a technique to take keyboard inputs from the buffer and process them according to the game's rules.

By implementing a technique to take keyboard inputs from the buffer and process them according to the game's rules, I have ensured that the user's keyboard inputs are properly received and processed by the system. This implementation ensures that the user's inputs control the Tetris pieces as intended and that the game functions properly.

This implementation of the keyboard input buffer ensures that the user's keyboard inputs are accurately captured and processed by the system. In addition, this guarantees that keyboard inputs are processed and dealt with correctly, allowing the user to control the game with their keyboard inputs.

3)
a) Determine the design requirements, such as essential design elements and colour palettes.

By researching best practices and trends in user interface design and exploring how design elements and colour palettes have been used in other Tetris games, I determined what design elements and colour schemes would be most appealing to users. This helped me to create a visually appealing user interface that would attract and retain users. In addition, by evaluating how design elements and colour palettes could enhance the game mechanics, I created a user interface that improved the gameplay experience for users. For example, by

optimising the design of blocks to be easily distinguishable from one another, users could more easily plan and execute their moves.

b) Design the user interface layout and create visually appealing interactive elements.

By compiling data on user needs and expectations, I established the essential components and design of the user interface. This helped me to create a user interface that was intuitive and easy to use, which made it more appealing to users. Also, by creating preliminary sketches of the interface layout and making drawings of the interface, I could visualise how the user interface would look and feel. This helped me to create a user interface that was aesthetically pleasing and easy to navigate. I made sure to pick the right colour scheme and font, and I created a user interface that was easy to read and use. This made it more appealing to users and helped them to stay engaged with the game.

4)

a) Define the game mechanics

To ensure the game mechanics were accurate and adhered to the original Tetris gameplay, I conducted thorough research on the rules and mechanics of the game. I studied how blocks fall, rotate, and clear lines and incorporated all of these elements into my kernel. This objective was important for the end user because it ensured the game mechanics were consistent with their expectations and experiences with Tetris.

I also used correct algorithms to ensure that the pieces rotated correctly. This involved researching and implementing the appropriate mathematical operations to make the pieces rotate smoothly and accurately. This objective was important for the end user because it ensured that the gameplay was smooth and satisfying.

b) Develop the game for the kernel

To design the kernel architecture, I carefully planned out the structure of the kernel and decided on the necessary modules and functions needed to implement the game mechanics. This objective was important for the end user because it ensured that the kernel was efficient and organised, contributing to a smooth and stable gameplay experience.

I then wrote the kernel code for the modules and functions outlined in the kernel architecture. This involved a lot of careful coding and testing to ensure that everything worked properly. This objective was important for the end user because it ensured the game was functional and reliable.

To test and debug the kernel, I conducted extensive testing to ensure that it implemented the Tetris game mechanics correctly and to identify and fix any errors. This objective was important for the end user because it ensured the game was playable and free of glitches or bugs.

c) Test and refine the game

To conduct usability testing, I tested the game with real users to identify any usability issues, user preferences, and areas for improvement. This objective was important for the end user because it ensured the game was intuitive and easy to use.

I also collected user feedback through surveys, feedback forms, and customer support channels to gain insight into user needs and expectations. This objective was important for the end user because it ensured that their feedback and preferences were taken into account and used to improve the game.
Finally, I used insights gained from testing, feedback, and analytics to refine and optimise the game mechanics and user interface. This objective was important for the end user because it ensured that the game was constantly being improved to meet their needs and preferences.

End-user feedback and analysis

The following is the feedback I received from the end user software engineer Emily Smith.

“Your kernel that runs Tetris is responsive, with smooth gameplay that provides a satisfying experience for the player. The game mechanics are well-implemented and work seamlessly, providing a sense of challenge and excitement as the player progresses through the levels. Additionally, your kernel handles interrupts well, ensuring the game continues without disruption.
The game also has a visually pleasing and easy-to-use interface. The colours and graphics used in the game are well-chosen, and the controls are intuitive, making it easy for players to pick up and play the game without any difficulty. This ensures that the game appeals to a wide range of users, including those new to gaming or Tetris.

If there were anything to include in your Tetris kernel, it would be a scoring system. A scoring system would add an extra level of engagement for the player, allowing them to compete against others and keep track of their progress in the game. By incorporating a scoring system, you could increase the game's replay value and make it more addictive, encouraging players to return for more. Additionally, a scoring system would allow players to compare their performance with others, creating a sense of community and competition among the game's users.”

After receiving feedback on the kernel, I took some time to analyse it and realised that I had overlooked an essential aspect of the game - the competitive aspect. I had been so focused on developing the game mechanics and ensuring that the kernel was responsive and handled interrupts well that I had yet to consider including a scoring system. I now understand that a scoring system is an important feature that helps increase users' competitiveness and engagement. It allows users to track their progress and compete against one another for the highest score, adding an extra layer of excitement and motivation to the game.

Possible improvements

Several possible improvements could be made to the Tetris kernel that would enhance its functionality and user experience. One possible improvement is to adapt the kernel to compile on different operating systems, such as Windows or Linux. This would allow a wider range of users to access and play the game, regardless of their preferred operating system. By making the kernel more widely available, we can reach a larger audience and increase the game's popularity.

Another potential improvement is to include a scoring system in the game. While the current kernel already offers a satisfying gameplay experience, adding a scoring system would provide users with a layer of competitiveness and challenge. It would allow players to track their progress and compare their scores with others, encouraging them to play the game more frequently and improve their skills. By including a scoring system, we can make the game more engaging and increase its replay value.

In addition to a scoring system, adding sound effects to the game could enhance the user experience. Sound effects such as music and sound effects for actions like rotating and dropping blocks can make the game more engaging and immersive. It can also provide feedback for the player, such as indicating when a line is cleared, or a block is locked in place. This can help users to understand the game mechanics better and enjoy the gameplay experience more thoroughly.

Another possible improvement would be adding a main menu to the game, allowing users to access different game modes or settings easily. This would provide a more user-friendly experience and make it easier for users to navigate the game's features. Additionally, allowing players to pause the game would provide greater convenience and flexibility, enabling them to take breaks or attend to other matters without losing their progress in the game. Finally, expanding the kernel to include more games would offer users more variety and entertainment options within the same system. This would increase the overall value of the kernel and make it more attractive to users.