ARCADE OS

Project Documentation
Computer Organization & Assembly Language
Fall-2023

Submitted To:
Ms. Fatima Aslam

Submitted By:

Ghulam Mustafa	Fa-2022/BSCS/188
Ammad Rasheed	Fa-2022/BSCS/199
Faizan Ali	Fa-2022/BSCS/187
Abubakar Ajmal	Fa-2022/BSCS/208
Ahsan Ilahi	Fa-2022/BSCS/210

Department of Computer Science,
Lahore Garrison University, Lahore, Main Campus.

[January 20th, 2024]

Abstract

Arcade OS is an operating system designed to provide a platform for retro gaming, developed in Assembly Language. It is designed to be bootable and run directly on the System using BIOS. Arcade OS offers a nostalgic journey with an old-school Command Line Interface paying homage to the MS-DOS era. The project focuses on three main aspects providing a collection of retro games, a classic Command Line Interface (CLI), and ensuring a smooth booting experience.

Introduction

Overview

Arcade OS, at its core, is an exploration into the technicalities of assembly language. It consists of a minimalistic Command Line Interface (CLI) paying tribute to MS-DOS. The system enables the execution of classic arcade games directly on bare metal, delivering a unique experience. Arcade OS is a practical endeavor with the ability to boot on the system and not confined to a specific medium. The flexibility ensures that the retro gaming experience is accessible on any system. A carefully crafted bootable platform, reliving the golden era of classic arcade gaming.

Motivation

Arcade OS is driven by the motivation of preserving the essence of classic arcade gaming and the simplicity of the Command Line Interface. In a world, dominated by technological advancements, Arcade OS takes a step back and helps its users relive the MS-DOS era. The Arcade OS is also was created to test and delve into the world of Assembly Language Programming embracing its power and facing the challenges it provides the developers with.

Project Scope:

The scope of our Arcade OS project is the development of a minimalistic Operating system purely dedicated to Retro-Gaming. It includes the designing of a bootable program written in x86 NASM Syntax Assembly Language, emphasizing the simplicity and nostalgia of Command Line Interface (CLI) and a system for direct execution of text-based retro games on bare-metal.

- **Inclusions:**

  Bootable OS, Command Line Interface, Classic Text- Based Games, Flexible Boot Support, Nostalgic MS-DOS Experience.

- **Exclusions:**

  The Project does not include: Real-time System Information or Custom Theme Setting due to time constraints. Features like Advanced Graphics and Animations, Sound Effects or Multiplayer Support was also not added.

Features

- **Command Line Interface (CLI)**

  Arcade OS features a minimalistic CLI, inspired by MS-DOS. The CLI serves as the point where the user interacts with the Arcade OS. It focuses on functionality and code over a graphical look.
- **Classic Arcade Games**

  The system offers several built-in games and users can add their games in the upcoming updates of the Arcade OS. The games run on bare metal without needing anything else to run the games.
  Games included in the Arcade OS beta are:
    - SNAKE
    - TETRIS
    - BRICKS
- **Bootable Capability**

  Arcade OS is designed to be bootable and flexible for users. It works from both floppy disks and disk images ensuring a nostalgic experience.
- **Assembly Language Powers**

  The project uses the powers of assembly language programming, allowing us developers to use assembly language to craft software that runs on bare-metal.
- **Nostalgic MS-DOS experience**

  Arcade OS remains lightweight and sticks to its minimalist CLI design focusing on speed and efficiency and providing a Nostalgic MS-DOS computing experience so that users can relive the joys and experiences of classic arcade-based gaming environments.

Architecture and System Requirements

Arcade OS is designed for x86 Processor Architecture and has very few system requirements making it compatible with running on various hardware.

The system requirements are:

- **Processor:** x86
- **Memory:** 512MB RAM
- **Storage:** 2MB
- **Boot Medium:** USB, Floppy Disks or Emulators
- **Display**: Standard VGA
- **Input:** Keyboard Only

Project Development

Code Documentation

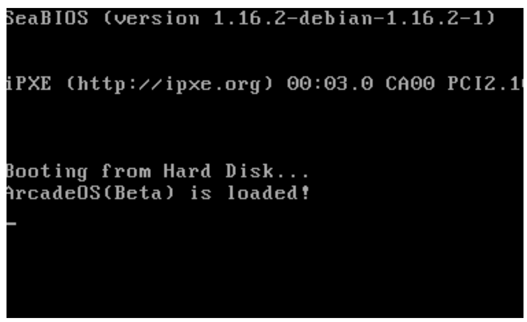
1. Bootloader (boot.asm):

Overview:

Initializes the system by setting up the environment and loads main components into the memory.

Memory Layout:

- Loaded at 0x0000_7c00.
- Files are loaded at 0x0000_7e00.
- The shell is loaded at 0x0000_8000.



(Bootloader Successfully being loaded)

Initialization:

- Set up the segment registers (DS, ES, SS).
- Initializes and sets up the stack pointers.
- Sets the video mode to 80x25 text mode and changes the color scheme.

Loading Files:

- Reads file names from sector 2on the USB flash drive.
- Loads the shell from sector 3.

User Interaction:

- Displays a Welcome Greeting.
- Waits for a key press before proceeding to jump to shell/home screen main menu.
- Video Mode and Colors: Sets background color to red.
- **BIOS** interrupts to manipulate video mode and colors.

(Arcade OS boot.asm main screen)

Procedure for Printing Strings:

- A procedure **print_string** to display strings.
- BIOS interrupts to print characters.

Reading Sectors:

- A procedure **read_sector** to read a single sector from the drive.
- Uses BIOS **interrupt 0x13** for I/O operations.

Error Handling:

- Displays an error message if it fails to read a sector.

2. List (list. asm):

Overview:

Lists available games and provides functionality to execute them.

Memory Layout:

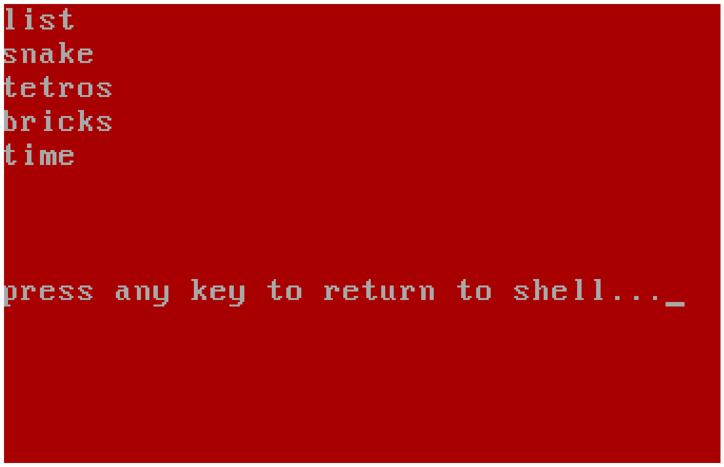
- Loaded at 0x0000_7e00.

Initialization:

- Sets up the environment similar to the bootloader.

Printing Available Games:

- A procedure **print_files** to display available games and other games.
- Retrieves file names from the file list.



(List Command Fetching available games/commands)

User Interaction:

- A message to press any key for the menu.
- Waits for a key press than proceeds returning to the **shell**.

Video Mode and Colors:

- Sets background color to red.

3. Shell (shell.asm):

Overview:

A command-line interface for user interaction. It also searches and executes entered games.

Memory Layout:

- Loaded at 0x0000_8000.

Initialization:

- Same environment setup as the bootloader.

User Interface Loop:

- Continuously **prompts** the user for **input**.
- Allows users to enter **commands**.
- **Typing** and **Deleting** Characters from the Prompt

(Shell.asm main page)

Command Processing:

- Processes the **input** and **searches** for game names or the command.
- Executes the selected game or the game.

File Search:

- A procedure **search_file** for comparing the input with available file names.
- **Executes** the corresponding game/command if a match is found.

(Handling User Input)

Game Execution:

- Utilizes BIOS disk I/O interrupts to load and execute games.

Procedure for Printing Strings:

- Reuses the **print_string** procedure.

Error Handling:

- Displays an **error message** if no file is found.

(File Not Found)

4. Time (time.asm):

Overview:

Lists available games and provides functionality to execute them.

Memory Layout:

- Loaded at 0x0000_8400.

Initialization:

- Changes the color of the shell and shows current time (Not available in beta version because of time constraint).

(Time Command Execution)

User Interaction:

- A message to press any key for the menu.
- Waits for a key press than proceeds returning to the **shell**.

Video Mode and Colors:

- Sets background color to yellow.

5. Files (files.asm):

Overview:

Contains the list of available games and commands. The List Command uses this file to read and display the names on the screen.

Memory Layout:

- Loaded at 0x0000_8400.

Memory Map

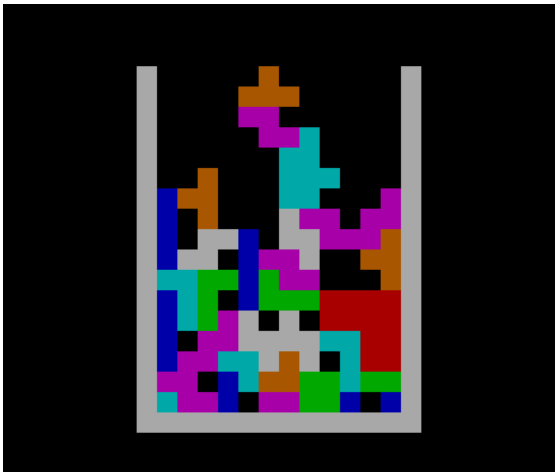
Start	End	Description	Size
0×0000-0000	0×0000-01FF	Boot Loader	512 bytes
0×0000-0200	0×0000-03FF	File Loader	512 bytes
0×0000-0400	0×0000-05FF	Shell	512 bytes
0×0000-0600	0×0000-07FF	List Command	512 bytes
0×0000-0800	0×0000-09FF	Snake Game	512 bytes
0×0000-0A00	0×0000-0BFF	Tetros	512 bytes
0×0000-0C00	0×0000-0DFF	Bricks	512 bytes
0×0000-0CE0	0×0000-0FFF	Time	512 bytes

(This Memory Map Visualizes the files and the addresses they are present at)

GAMES

1. Tetris:

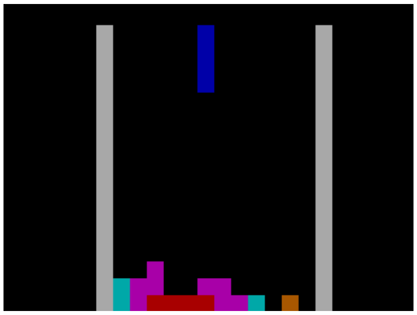
Tetris is a classic block game that challenges players to manipulate shapes falling from top to bottom these shapes are known as bricks too. The Tetris game in Arcade OS contributes a lot to the overall gaming and nostalgic experience within the operating system.



Game Logic

New Brick Generation:

- Randomly select a brick.
- Starting position at row 4 and column 38.

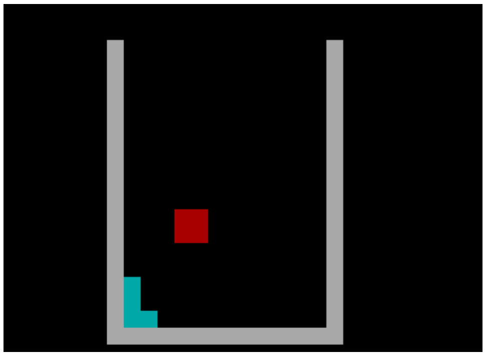


Collision Detection:

- Collisions with other bricks or boundaries.
- Ends the game if a collision is detected with the boundaries.

User Input Handling:

- Waits for user input.
- Handles Arrow keys for brick movement and rotation.



Row Clearing:

- Clears rows when they are filled.

Macros

- **sleep:** Sleeps for the given number of microseconds.
- **select_brick:** Chooses a brick at random.
- **clear_screen:** Sets video mode and hides the cursor.

Features

Implemented:

1. Different Brick Colors.
2. Hidden Cursor.
3. Movement: Left/Right Arrows.

4. Brick Rotation: Up Arrow.
5. Fast Drop: Down Arrow.
6. Random Brick Selection.
7. Clean Playing Field after filled row.

Missing (Due to Size):

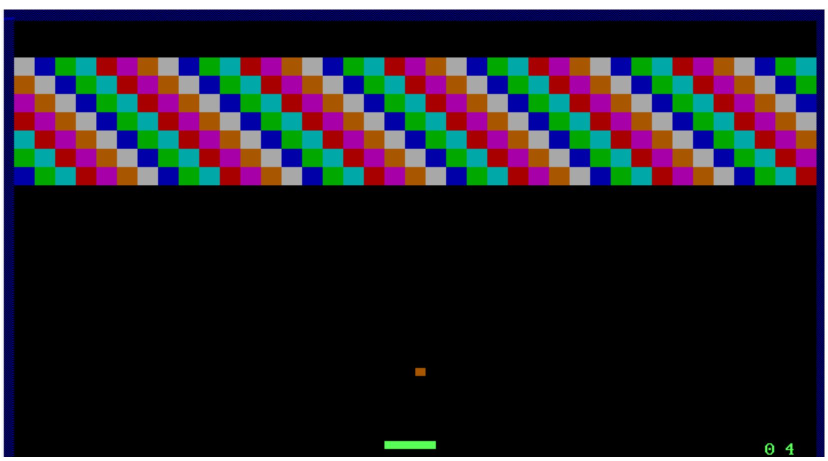
1. Scores & Highscores.
2. Introductory Animation.
3. Game Over Message & Restart.
4. Next Brick Preview.
5. Speed Increase Mechanism.

Controls

- **Arrow Keys Left/Right:** Move Left Right
- **Up Key:** Rotate the shape
- **Down Key**: Fast Drop

2. Bricks

Bricks is a game involving bricks, which is a breakout-style game one of the most classic games from the Retro-Gaming era. The game features a **paddle**, a **ball**, and **bricks** on the screen. The objective is to break the bricks with the ball using the paddle provided.



Game Logic

Initialization:

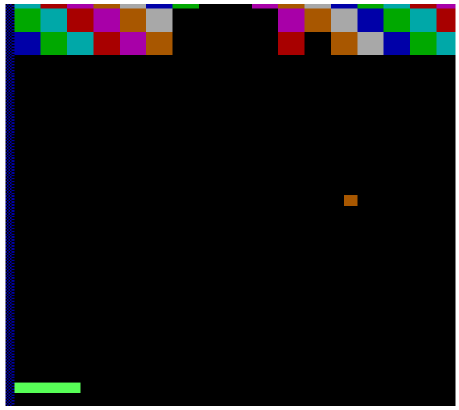
- Video mode of **80x25** with **16 colors**.
- Initializes the stack and global variables.

Level Setup:

- Draws borders and bricks for the new level.
- The new level is started after completing the last.

Ball Movement:

- Controls the ball movement based on input.
- Collision detection with the paddle, borders, and bricks.



Paddle Control:

- Allows the player to control the paddle.
- Handles left and right movement of the paddle.

Score and Lives:

- Updates the player's score.
- Manages the balls count.

Game Flow:

- The game loop continues until all balls are popped.

Macros:

- **wait_frame:** Pauses the game for a short time.
- **locate_ball:** Calculates the position of the ball.
- **update_score:** Updates and displays the score.

Controls

- **Left Shift:** Start Game.
- **Left CTRL:** Move Paddle Left
- **Left ALT:** Move Paddle Right

Limitations

- Graphics and Animation (due to 512 bytes limit)
- Sound Effects (due to 512 bytes limit)
- No Increase in Speed (due to 512 bytes limit)
- No Game Over Screen (due to 512 bytes limit)

3. Snake:

Snake is a classic arcade game and without Snake the Arcade OS is incomplete. The game challenges players to control a snake, consume food, and grow longer. As the snake grows, the game becomes challenging, requiring players to traverse the snake without colliding with its own body or the game boundaries.

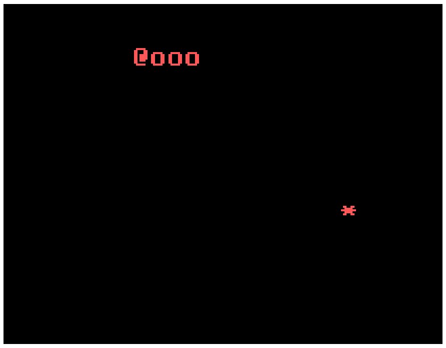
Features

Snake Movement:

- The snake moves continuously in a single direction.
- Players can control the snake using arrow keys.

Food Consumption:

- Food items appear on the screen.
- The snake eats food, it grows longer.



(Snake Size After consumption of 3 fruits)

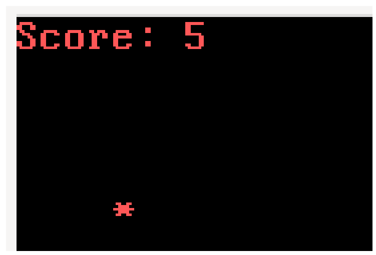
Collision Detection:

- The game detects collisions with the snake's own body and walls.
- Colliding results in the end of the game.

(if snake hits the wall)

Score Tracking:

- Players earn points for each food item.
- Score increases with each food consumed.



Game Over Handling:

- Displays game-over message when the snake collides.
- Option to restart the game.

(if snake hits the itself)

Limitations

- Graphics and Animation (due to 512 bytes limit)
- Sound Effects (due to 512 bytes limit)
- Game State Transitions (due to 512 bytes limit)
- No Increase in Speed (due to 512 bytes limit)
- No Walls Graphics (due to 512 bytes limit)

Controls

- **Up Key:** Move Snake Up.
- **Down Key:** Move Snake Down.
- **Left Key:** Move Snake Left.
- **Right Key:** Move Snake Right.
- **Q:** Quit Game/Return to Shell

Techniques and Concepts Used in Arcade OS

Boot Sector:

The x86 boot sector is the entry point during system boot.

Implementation:

- Used in **boot.asm**.
- Directives **[bits 16] and [org 0x7c00]** define **16-bit assembly**.
- BIOS **interrupts** (int 0x10, int 0x13) video/storage access.

CHS (Cylinder-Head-Sector) :

Legacy addressing for hard drive access using cylinder, head, and sector.

Implementation:

- Used in **boot. asm**.
- **mov dl, 0x80** specifies drive.
- **int 0x13** BIOS interrupt for **reading sectors.**

Video Mode Setting:

Configuring the display modes for output/information.

Implementation:

- Used in **boot.asm**.
- **mov ah, 0x00** and **int 0x10** BIOS **interrupts** set video **mode**.

Memory Segmentation:

Organizes memory into segments for efficient use.

Implementation:

- Used in **boot.asm**.
- **mov ax, 0, mov ds, ax, mov es, ax, mov ss, ax** initializing the segment registers.

Command-Line Interface (CLI) Implementation:

Provides a text-based interface for interaction.

Implementation:

- Used in **shell.asm**.
- User input using **BIOS** interrupts **(int 0x10, int 0x16, int 0x13).**
- **File search** by comparing input with available names.

BIOS Interrupts Usage:

Software interrupts for BIOS functions.

Implementation:

- Used in **boot**.asm, **files**.asm, and **shell**.asm.
- **int 0x10, int 0x16, int 0x13** used for video mode, keyboard input, and storage access.

Development Tools and Technologies Used in Arcade OS

Assembly Language:

The primary language used for building the Arcade OS, getting low-level control over the system.

Variation: x86 Assembly (NASM Syntax)

NASM (Netwide Assembler):

NASM served as the assembler to compile the assembly code into machine-readable format.

Usage: NASM was integral to the build process, translating assembly code into executable binaries.

BIOS (Basic Input/Output System):

The basic firmware interface for input and output operations in the system.

Usage: BIOS **interrupts**, such as int **0x10** and int **0x13**, were used for **video mode** setting, **storage** device access, and other important functions. Interacting with BIOS was crucial for initializing the system and file handling.

QEMU (Quick Emulator):

QEMU, enabling testing of the operating system without physical hardware.

Usage: QEMU played an important role in the testing phase. The Arcade OS image file was loaded into QEMU to simulate its execution without the PC. This allowed our group to debug system behavior in a controlled environment. Commands such as **qemu-system-x86_64** were used to launch the emulator.

GitHub:

GitHub served as our version control system, source code management, and group collaboration.

Usage: Git was used for collaborative development, allowing our group to work on the code at the same time.

Testing and Evaluation:

To ensure the quality and reliability of our Arcade OS, testing and evaluation was performed primarily through QEMU. The traditional testing methods (Unit testing, System Testing) were not used due to the project constraints, but the QEMU evaluation still served effectively for simulating our code in a virtual environment.

Results and Achievements

Key Outcomes:

The development of Arcade OS yielded a lot of outcomes, both achievements and challenges:

- **Successful Booting**
- **CLI Implementation**
- **Games Execution and Functionality**
- **Assembly Language Proficiency**

Challenges Faced:

A notable number of challenges were faced by our group, the key difficulties we faced include:

- **Size Limitation** due to **Boot Sector 512-byte Constraint.**
- Limited Colors and graphics options.
- Features Exclusions due to Project **Deadline.**
- **Limited Testing** on bare-metal hardware.

Future Enhancements

While Arcade OS has achieved the primary goals, there is still room for future enhancements and refinements to provide a better user experience and more functionality. Some areas of improvement that our group wanted to work on but was not able to due to the time constraint included:

- Expanded Game Library
- Graphical Enhancements
- Custom Theme Settings
- Real-Time System Information
- Multiplayer Support

Conclusion

In conclusion, Arcade OS provides the users with a nostalgic journey into the era of classic gaming and the MS-DOS Command line interface (CLI) experience. The project delivers a minimalistic still functional operating System with a CLI. The desire for preserving the essence of retro gaming and embracing the development challenges that are provided by the Assembly language made Arcade OS a unique project workable on nearly any system without any additional requirements or dependency. bootable platform ensures flexibility.

While some features were missing the project still achieves all its core goals of being simple and efficient. The implemented games – Snake, Tetris, Bricks – shows the power of the our system providing users with engaging gaming experience.

The development involved overcoming challenges boot sector initialization, memory map, game logics. Techniques like CHS addressing, x86 Real Mode and BIOS interrupts were used.

References

- OS Wiki - CHS Addresses: https://en.wikipedia.org/wiki/Cylinder-head-sector
- x86 Real Mode: https://wiki.osdev.org/Real_Mode
- x86 Real Mode Memory Segmentation: https://wiki.osdev.org/Segmentation
- x86 Memory Map: https://wiki.osdev.org/Memory_Map_(x86)
- x86 BIOS Interrupts: http://www.ablmcc.edu.hk/~scy/CIT/8086_bios_and_dos_interrupts.htm
- x86 Assembly Registers: https://www.assemblylanguagetuts.com/x86-assembly-registers-explained/
- x86 Assembly Instructions: https://www.aldeid.com/wiki/X86-assembly/Instructions
- NASM (Netwide Assembler): https://nasm.us/
- QEMU (Quick Emulator): https://www.qemu.org/

Appendix

Complete Code

Boot.asm:

```
[bits 16]
[org 0x7c00]
%define BOOTSECTOR_ADDR 0x7c00
%define FILES_ADDR 0x0000_7E00
%define SHELL_ADDR 0x800
%define THEME_ADDR 0x0000_8400
mov si, success_message
call print_string
mov ah, 0x00
int 0x16
mov ah, 0x00
mov al, 0x03
int 0x10
mov ah, 0x0b
mov bh, 0
mov bl, 0x04
mov cx, 0
mov dx, 0
int 0x10
mov ax, 0
mov ds, ax
mov es, ax
mov ss, ax
mov bp, BOOTSECTOR_ADDR
mov sp, bp
mov si, welcome_message
call print_string
mov si, press_any_key_message
call print_string
mov ah, 0x00
int 0x16
mov bx, FILES_ADDR
mov cl, 2
call read_sector
mov ax, SHELL_ADDR
mov es, ax
mov bx, 0
mov cl, 3
call read_sector
jmp SHELL_ADDR:0x0000
print_string:
   cld
   mov ah, 0x0e
   mov bh, 0
   mov bl, 0x04
   .next_char:
      lodsb
      cmp al, 0
      je .return
      int 0x10
      jmp .next_char
   .return: ret
read_sector:
   mov ah, 0x02
   mov al, 1
mov ch, 0
   mov dh, 0
   mov dl, 0x80
   int 0x13
   jc .error
   ret
   .error:
      mov si, error_message
      call print_string
      jmp $
success_message db 'ArcadeOS(Beta)
is loaded!', 10, 13, 0
welcome_message db '    ___    ____
__________    _____  _______   _____
______', 10, 13,'  /   |  / __ \/ ____/   |
/ __ \/ ____/  / __ \/ ___/', 10, 13,'  /
/| | / /_/ / /   / /| | / / / / __/   / / /
/\__ \ ', 10, 13,' / ___ |/ _, _/ /___/
___ |/ /_/ / /___  / /_/ /___/ / ', 10,
13,'/_/  |_/_/ |_|\____/_/
|_/_____/_____/  \____//____/', 10,
13, 0
press_any_key_message db
10,13,'Press any key for menu...', 10,
13, 0
error_message db 'Failed to read
sector from USB!', 10, 13, 0
times 510 - ($ - $$) db 0
dw 0xaa55
```

Files.asm:

```
[bits 16]

; list of available games

db 'list', 0, 0, 0, 0

db 'snake', 0, 0, 0

db 'tetros', 0, 0

db 'bricks', 0, 0
```

```
db 'time', 0, 0

times 512 - ($ - $$) db 0
```

List.asm:

```
[bits 16]
[org 0x7c00]

%define OFFSET 8
%define FILES_ADDR 0x7e00
%define SHELL_SEGMENT 0x800

int 0x10
mov ax, 0
mov ds, ax
mov es, ax
mov ss, ax
mov bp, 0x7c00
mov sp, bp
mov ah, 0x00
mov al, 0x03
int 0x10

mov ah, 0x0b
mov bh, 0
mov bl, 0x04
mov cx, 0
mov dx, 0
int 0x10

call print_files
mov si, press_any_key
call print_string
mov ah, 0x00
int 0x16
jmp SHELL_SEGMENT:0x0000
```

```
print_files:
  cld
  mov bx, 0

  .next_file:
    mov ax, [file_list + bx]
    cmp ax, no_file
    je .return
    mov si, ax
    call print_string
    mov si, new_line
    call print_string
    add bx, 2
    jmp .next_file

  .return: ret

print_string:
  cld
  mov ah, 0x0e
  .next_char:
    lodsb
    cmp al, 0
    je .return
    int 0x10
    jmp .next_char
  .return: ret
press_any_key db 10, 13, 'press any key to return to
shell...', 0
new_line db 10, 13
no_file dw 0
file_list dw FILES_ADDR, FILES_ADDR + OFFSET,
FILES_ADDR + 2 * OFFSET, FILES_ADDR + 3 * OFFSET,
FILES_ADDR + 4 * OFFSET, FILES_ADDR + 5 * OFFSET,
FILES_ADDR + 6 * OFFSET, no_file

times 512 - ($ - $$) db 0
```

Shell.asm:

```
[bits 16]
[org 0x8000]

%define BOOTSECTOR_SEGMENT 0x7c0
%define BOOTSECTOR_ADDR 0x7c00
%define FILES_ADDR 0x7e00
%define OFFSET 8
```

```
int 0x10
mov ah, 0x0e
mov al, 0
int 0x10
mov ah, 0x0e
mov al, 8
int 0x10
```

```
cmp al, 0
je .return_true
jmp .next_byte

.return_true:
  mov cl, 1
  ret
```

```
%define ENTER_KEY 0x1c
%define BACKSPACE_KEY 0x0e

mov ax, 0
mov ds, ax
mov es, ax
mov ss, ax
mov bp, 0x7c00
mov sp, bp
mov ah, 0x00
mov al, 0x03
int 0x10

mov si, intro
call print_string

; main OS loop
shell_loop:
   mov si, user_prompt
   call print_string
   mov di, user_input
   mov al, 0
   times 20 stosb
   mov di, user_input

   .next_byte:
      mov ah, 0x00
      int 0x16
      cmp ah, ENTER_KEY
      je .search
      cmp ah, BACKSPACE_KEY
      je .erase_char
      stosb
      mov ah, 0x0e
      int 0x10
      jmp .next_byte

   .erase_char:
      mov ah, 0x03
      int 0x10
      cmp dl, 3
      je .next_byte
      mov ah, 0x0e
      mov al, 8

intro db
'|==============================|',10,13,'|-
------ARCADE OS (BETA)--------|',10,13,'|
|',10,13,'| Type "list" to list the Games |', 10,
13,'|==============================|', 0
user_prompt db 10, 13, ' $ ', 0
user_input times 20 db 0
new_line db 10, 13
```

```
      mov al, 0
      dec di
      stosb
      dec di
      jmp .next_byte

    .search:
      call search_file

   jmp shell_loop

search_file:
   cmp byte [user_input], 0
   je .return

   mov bx, 0
   mov dl, 3

   .next_game:
      mov ax, [file_list + bx]
      cmp ax, no_file
      je .no_file_found
      add bx, 2
      inc dl
      call compare_strings
      cmp cl, 1
      je execute
      jmp .next_game

   .no_file_found:
      mov si, error_no_file
      call print_string
      ret
   .return: ret

compare_strings:
   cld
   mov di, user_input
   mov si, ax

   .next_byte:
      lodsb
      scasb
      jne .return_false
      jmp $

error_message db 'Failed to
read sector from USB!', 10,
13, 0
error_no_file db 10, 13, 'No
file found!', 0
```

```
      .return_false:
         mov cl, 0
         ret

execute:
   mov ax, BOOTSECTOR_SEGMENT
   mov es, ax
   mov bx, 0
   mov cl, dl
   call read_sector
   jmp
BOOTSECTOR_SEGMENT:0x0000

print_string:
   cld
   mov ah, 0x0e
   mov bh, 0
   mov bl, 0x0F

.next_char:
   lodsb
   cmp al, 0
   je .return
   int 0x10
   jmp .next_char

.return:
   ret

read_sector:
   mov ah, 0x02
   mov al, 1
   mov ch, 0
   mov dh, 0
   mov dl, 0x80
   int 0x13
   jc .error
   ret

   .error:
      mov si, error_message
      call print_string
```

```
no_file dw 0
file_list dw FILES_ADDR, FILES_ADDR + OFFSET,
FILES_ADDR + 2 * OFFSET, FILES_ADDR + 3 *
OFFSET, FILES_ADDR + 4 * OFFSET, FILES_ADDR + 5
* OFFSET, FILES_ADDR + 6 * OFFSET, no_file

times 512 - ($ - $$) db 0
```

Time.asm:

```
mov ax, 0                      ; set ACCUMULATOR REGISTER to
0
mov ds, ax                     ; set DATA SEGMENT to 0
mov es, ax                     ; set EXTRA SEGMENT to 0
mov ss, ax                     ; set STACK SEGMENT to 0
mov bp, 0x7c00                 ; set STACK BASE to
0x0000_7c00
mov sp, bp                     ; set STACK POINTER to
0x0000_7c00
; Set video mode to 80x25 text mode (change the value of
al)
mov ah, 0x0e   ; BIOS teletype output
mov al, '>'
int 0x10
mov al, ' '
int 0x10
mov al, '-'
; Set background color to white and text color to black
mov ah, 0x0b
mov bh, 0      ; Page number
mov bl, 0x0C   ; Text color: black (lower 4 bits),
Background color: white (higher 4 bits)
mov cx, 0      ; Starting column
mov dx, 0      ; Ending column
int 0x10
; Wait for a keypress
mov ah, 0
int 0x16
; Clear the screen
mov ah, 0x06
mov al, 0
int 0x10
mov al, '-'
int 0x10
mov al, ':'
int 0x10
mov al, '-'
int 0x10
mov al, '-'
int 0x10
mov al, ' '
int 0x10
mov al, 'A'
int 0x10
mov al, 'M'
int 0x10
mov bh, 0
mov cx, 0
mov dh, 24
int 0x10
message db 'Time Feature Not Availible Yet in Beta
Version', 0
; Jump back to the shell
jmp 0x0000:0x7e00
```

Bricks.asm:

```
%ifdef com_file
  org 0x0100
%else
  org 0x7c00
%endif

old_time:       equ 16
        mov al,4
        push ax
        mov bp,sp
another_level:
        mov word [bp+bricks],273
        xor di,di
        mov ax,0x01b1
        stosw
        inc ah
        cmp ah,0x08
        jne .4
        mov ah,0x01
.4:
        loop .3
```

```
ball_x:         equ 14
ball_y:         equ 12
ball_xs: equ 10
ball_ys: equ 8
beep:           equ 6
bricks:         equ 4
balls:    equ 2
score:     equ 0

start:
        mov ax,0x0002
        int 0x10
        mov ax,0xb800
        mov ds,ax
        mov es,ax
        sub sp,32
        xor ax,ax
        push ax
        int 0x16
        test al,0x04

je .1
        mov byte [di+6],0
        mov byte [di+8],0
        sub di,byte 4
        cmp di,0x0f02
        ja .1
        mov di,0x0f02
.1:
        test al,0x08
        je .2
        xor ax,ax
        stosw
        stosw
        stosw
        stosw
        stosw
        pop di
        mov bx,[bp+ball_x]
        mov ax,[bp+ball_y]
        call locate_ball
        test byte [bp+ball_y],0x80
        mov ah,0x60
        je .12
        mov ah,0x06
.12:    mov al,0xdc
        mov [bx],ax
        push bx
        pop si
.14:
        mov bx,[bp+ball_x]
        mov ax,[bp+ball_y]
        add bx,[bp+ball_xs]
        mov cx,80
        cld
        rep stosw
        mov cl,24
.1:
        stosw
        mov ax,0x20
        push cx
        cmp cl,23
        jnb .2
        sub cl,15
        jbe .2
        mov al,0xdb
        mov ah,cl
.2:
        mov cl,39
.3:
        stosw
test ah,ah
        jnz .9
        neg word [bp+ball_ys]
.9:     jmp .14
.3:
        cmp al,0xdf
        jne .4
        sub bx,di
        sub bx,byte 4
        mov cl,6
        shl bx,cl
        mov [bp+ball_xs],bx
        mov word [bp+ball_ys],0xff80
        mov cx,2711
        call speaker
        pop bx
        pop ax
        jmp .14
.4:
        cmp al,0xdb
        jne .5
        mov cx,1355
        call speaker
        test bl,2
        jne .10
        dec bx
        dec bx
.10:    xor ax,ax
        mov [bx],ax
        mov [bx+2],ax
        inc word [bp+score]
        neg word [bp+ball_ys]
        pop bx
        pop ax
        dec word [bp+bricks]
        pop cx
        mov ax,0x01b1
        stosw
        loop .1
        mov di,0x0f4a
another_ball:
        mov byte [bp+ball_x+1],0x28
        mov byte [bp+ball_y+1],0x14
        xor ax,ax
        mov [bp+ball_xs],ax
        mov [bp+ball_ys],ax
        mov byte [bp+beep],0x01
        mov si,0x0ffe
game_loop:
        call wait_frame
        mov word [si],0x0000
        call update_score
        mov ah,0x02
1:      call wait_frame.2
        int 0x20
wait_frame:
.0:
        mov ah,0x00
        int 0x1a
        cmp dx,[bp+old_time]
        je .0
        mov [bp+old_time],dx
        dec byte [bp+beep]
        jne .1
.2:
        in al,0x61
        and al,0xfc
        out 0x61,al
.1:
        ret
speaker:
        mov al,0xb6
        out 0x43,al
        mov al,cl
        out 0x42,al
        mov al,ch
        out 0x42,al
        in al,0x61
        or al,0x03
        out 0x61,al
        mov byte [bp+beep],3
        ret
locate_ball:
        mov al,0xa0
        mul ah
        mov bl,bh
        mov bh,0
        shl bx,1
```

```
        add ax,[bp+ball_ys]
        push ax
        push bx
        call locate_ball
        mov al,[bx]
        cmp al,0xb1
        jne .3
        mov cx,5423
        call speaker
        pop bx
        pop ax
        cmp bh,0x4f
        je .8
        test bh,bh
        jne .7
.8:
        neg word [bp+ball_xs]
.7:

.3:     mov [bx],ax
        dec bx
        dec bx
        jne .14
        jmp another_level
.5:
        pop bx
        pop ax
.6:
        mov [bp+ball_x],bx
        mov [bp+ball_y],ax
        cmp ah,0x19
        je ball_lost
        jmp game_loop
ball_lost:
        mov cx,10846
        call speaker
        mov word [si],0
        dec byte [bp+balls]
        js .1
        jmp another_ball
        ret
%ifdef com_file
%else%endif
        add bx,ax
        ret
update_score:
        mov bx,0x0f98
        mov ax,[bp+score]
        call .1
        mov al,[bp+balls]
.1:
        xor cx,cx
.2:     inc cx
        sub ax,10
        jnc .2
        add ax,0x0a3a
        call .3
        xchg ax,cx
        dec ax
        jnz .1

        times 510-($-$$) db 0x4f
        db 0x55,0xaa
```

Tetros.asm:

```
%endif

%macro sleep 1
        pusha
        xor cx, cx
        mov dx, %1
        mov ah, 0x86
        int 0x15
        popa
%endmacro

%macro select_brick 0
        mov ah, 2
        int 0x1a
        mov al, byte [seed_value]
        xor ax, dx
        mov bl, 31
        mul bl
        inc ax
        mov byte [seed_value], al
        xor dx, dx
        mov bx, 7
        div bx
        shl dl, 3
        xchg ax, dx
%endmacro

%macro clear_screen 0
: push cx
        inc dh
        mov dl, field_left_col
        mov cx, field_width
        mov bx, 0x78
        call set_and_write
        cmp dh, 21
        je ib
        inc dx
        mov cx, inner_width
        xor bx, bx
        call set_and_write
ib: pop cx
        loop ia
%endmacro

delay:      equ 0x7f00
seed_value: equ 0x7f02

section .text

start_tetris:
        xor ax, ax
        mov ds, ax
        init_screen
new_brick:
        mov byte [delay], 100
        select_brick
        pop ax
        jz no_key
        call clear_brick
        cmp ch, 0x4b
        je left_arrow
        cmp ch, 0x48
        je up_arrow
        cmp ch, 0x4d
        je right_arrow

        mov byte [delay], 10
        jmp clear_keys
left_arrow:
        dec dx
        call check_collision
        je clear_keys
        inc dx
        jmp clear_keys
right_arrow:
        inc dx
        call check_collision
        je clear_keys
        dec dx
        jmp clear_keys
up_arrow:
        mov bl, al
        inc ax
        inc ax
```

```
	xor ax, ax
	int 0x10
	mov ah, 1
	mov cx, 0x2607
	int 0x10
%endmacro

field_left_col:  equ 13
field_width:    equ 14
inner_width:    equ 12
inner_first_col: equ 14
start_row_col:   equ 0x0412

%macro init_screen 0
	clear_screen
	mov dh, 3
	mov cx, 18
	inc dh
	call check_collision
	je lp
	dec dh
	call print_brick
	call check_filled
	jmp new_brick

set_and_write:
	mov ah, 2
	int 0x10
	mov ax, 0x0920
ia
	int 0x10
	ret

set_and_read:
	mov ah, 2
	int 0x10
	mov ah, 8
	int 0x10
	ret

replace_current_row 0
	pusha
	mov dl, inner_first_col
	mov cx, inner_width
cf_aa:
	push cx
	dec dh
	call set_and_read
	inc dh
	mov bl, ah
	mov cl, 1
	call set_and_write
	inc dx
	mov dx, start_row_col
lp:
	call check_collision
	jne $
	call print_brick

wait_or_keyboard:
	xor cx, cx
	mov cl, byte [delay]
wait_a:
	push cx
	sleep 3000

	push ax
	mov ah, 1
	int 0x16
	mov cx, ax
cf_loop:
	call set_and_read
	shr ah, 4
	jz cf_is_zero
	inc bx
	inc dx
cf_is_zero:
	loop cf_loop
	cmp bl, inner_width
	jne next_row
replace_next_row:
	replace_current_row
	dec dh
	jnz replace_next_row
	call check_filled
cf_done:
	popa
	ret

clear_brick:
	xor bx, bx
	jmp print_brick_no_color
print_brick:
	mov bl, al
	shr bl, 3
	inc bx
	shl bl, 4
print_brick_no_color:
	inc bx
	mov di, bx
	jmp check_collision_main
	; BL = color of brick
	; DX = position (DH = row), AL
= brick offset
	; return: flag
check_collision:
	test al, 00000111b
	jnz nf
	sub al, 8
nf: call check_collision
	je clear_keys
	mov al, bl
clear_keys:
	call print_brick
	push ax
	xor ah, ah
	int 0x16
	pop ax
no_key:
	pop cx
	loop wait_a

	call clear_brick
	mov cx, 1
	call set_and_write
	popa
	jmp is_zero_a
ee:
	call set_and_read
	shr ah, 4
	jz is_zero_a
	inc bx
is_zero_a:
	pop ax

is_zero:
	shl ax, 1
	inc dx
	loop zz
	sub dl, 4
	inc dh
	pop cx
	loop cc
	or bl, bl
	popa
	ret

bricks:
	db 01000100b, 01000100b,
00000000b, 11110000b
	db 01000100b, 01000100b,
00000000b, 11110000b
	db 01100000b, 00100010b,
00000000b, 11100010b
	db 01000000b, 01100100b,
00000000b, 10001110b
	db 01100000b, 01000100b,
00000000b, 00101110b
```

```
        pop cx
        loop cf_aa
        popa
%endmacro

check_filled:
        pusha
        mov dh, 21
next_row:
        dec dh
        jz cf_done
        xor bx, bx
        mov cx, inner_width
        mov dl, inner_first_col
        pusha
        mov bx, di
        xor al, al

%ifndef DEBUG
times 446-($-$$) db 0
        db 0x80
        mov di, 0
check_collision_main:
        pusha
        xor bx, bx
        mov bl, al
        mov ax, word [bricks + bx]
        xor bx, bx
        mov cx, 4
cc:
        push cx
        mov cl, 4
zz:
        test ah, 10000000b
        jz is_zero
        push ax
        or di, di
        jz ee
  db 0x00, 0x01, 0x00

%endif
  db 0x17
  db 0x00, 0x02, 0x00
  db 0x00, 0x00, 0x00, 0x00
  db 0x02, 0x00, 0x00, 0x00
        db 00100000b, 01100010b, 00000000b, 11101000b
        db 00000000b, 01100110b, 00000000b, 01100110b
        db 00000000b, 01100110b, 00000000b, 01100110b
        db 00000000b, 11000110b, 01000000b, 00100110b
        db 00000000b, 11000110b, 01000000b, 00100110b
        db 00000000b, 01001110b, 01000000b, 01001100b
        db 00000000b, 11100100b, 10000000b, 10001100b
        db 00000000b, 01101100b, 01000000b, 10001100b
        db 00000000b, 01101100b, 01000000b, 10001100b
times 510-($-$$) db 0
        db 0x55
        db 0xaa
```

Note:

The code updates will be available on the GitHub repository: *https://github.com/Musxeto/ArcadeOS.git*