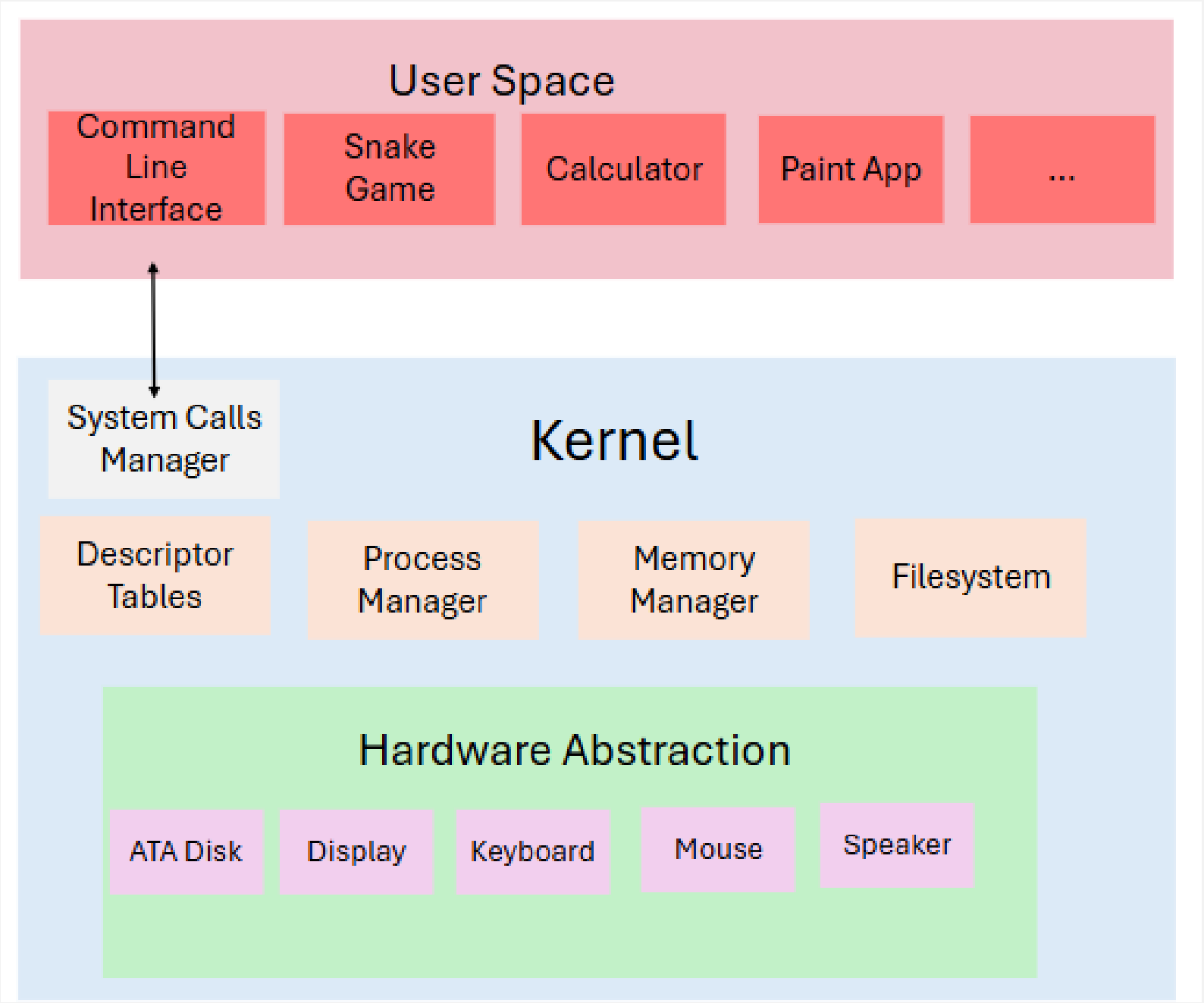
MazOS - Technical Details

This document highlights the technical details involved in designing and building MazOS

Contents

1. Design



High level overview of MazOS

1. Design - Kernel

The kernel is the core of the OS, and it is responsible for managing critical resources, like memory and processes, and initialises drivers.

The kernel allows for the communication between user space applications and hardware.

Memory is handled around a physical memory map, which tracks available and allocated blocks. Memory frames are allocated contiguously using a **first-fit to minimise external fragmentation**.

The scheduling protocol follows a **first-come first-served algorithm**.

1. Design – The Driver System

The driver sub-system's main responsibility is enforcing a consistent interface that allows application to interact with the hardware. The subsystem was inspired by the **Common Driver Model** where the drivers implemented in a consistent and standardised way (at least I tried).

The following hardware drivers were implemented: simple text-mode **display driver**, **PS/2 Keyboard** and **Mouse**, **ATA disk driver**, **Sound-Blaster 16** sound driver, **VGA graphics mode**.

1. Design – The filesystem

MazOS implements a FAT32 filesystem. The choice of FAT32 reflects its wide adoption, compatibility and documentation. MazOS allows for a wide range of file operations, including: file reads, file writes, file creation and removals, directory creation and removal.

1. Design – System Calls

MazOS implements the following POSIX compliant system calls:

• **sys_write**(fd, buf, count): Writes data from a buffer to a file descriptor. The implementation restricts writing to stdout (fd = 1), ensuring that user applications can output text through the console. The function converts the data into a UEFI-compatible format before invoking OutputString().
• **sys_read**(fd, buf, count): Reads user input from stdin (fd = 0) by interacting with the UEFI console input buffer. The function ensures that characters are correctly processed and returned to the user application.
• **sys_sbrk**(increment): Implements dynamic memory allocation by adjusting the program's heap. The kernel keeps track of the heap's end address and expands it as needed.
• **sys_exit**(status): Terminates a process and prevents further execution by entering an infinite loop. This syscall ensures that processes cleanly exit without affecting system stability.
• **sys_execve**(path, argv, envp): Loads and executes a new program from a given path. The implementation leverages UEFI services to load and start an executable, ensuring program execution is managed at the kernel level.
• **sys_fork**(): Creates a new process by duplicating the memory space of the parent process. The OS maintains a process table to track active processes and their memory allocations.
• **sys_open**(pathname, flags, mode) / sys_close(fd): Provide basic file management, allowing programs to access and release file resources. The kernel maintains a file descriptor table, ensuring that opened files are properly tracked.
• **sys_kill**(pid, sig): Terminates a process by marking it as inactive in the process table, preventing further execution.

2. Architecture

MazOS is built around a classic monolithic kernel architecture, where all critical components operate in privileged mode. This ensures that performance is prioritised as memory access is direct and context switching is minimised.

The kernel architecture tries to implement a hub-and-spoke model where the core kernel acts as a central mediator for all inter-component communication.

2. Architecture – Control Flows

- Interrupt-Driven Flow: events associated with hardware trigger interrupts that are captured and distributes them to the appropriate handler registered in the IRQ table.
- System-Call Flow: User space programs request services through the POSIX compliant interface that the kernel deals with.
- Kernel-Initiated Flow: The kernel manages resources such as memory and process scheduling.

3. Tools Used

- **C++**: as the main programming language
- **x86** assembly language: for interacting directly with the hardware
- **Makefile**: for automating the build process
- **Git**: used for version control
- **GCC** and **G++**: were used as the main compilers for translating C++ code into machine code
- **NASM**: used to assemble x86 code into binary
- **QEMU**: used for emulating hardware and testing the OS
- **Bochs**: another alternative for emulating
- **Linux** (Ubuntu): it provides a wide set of tools for cross-compilation, debugging and testing