COMPUTER ARCHITECTURE LAB

PROJECT REPORT

MEMBER LIST

Hà Việt Khánh	20225979	Khanh.HV225979@sis.hust.edu.vn
Nguyễn Hải Đăng	20226019	Dang.NH6019@sis.hust.edu.vn

Teacher: Lê Bá Vui

Hà Nội — 2024

1 RAID 5 SIMULATION

1.1 Problem Description

The RAID5 drive system requires at least 3 hard disks, in which parity data will be stored on 3 drives as shown below. Write a program to simulate the operation of RAID 5 with 3 drives, assuming that each data block has 4 characters. The interface is as shown in the example below. Limit the length of the input string to a multiple of 8.

The task involves:

- Reading a string from the user.
- Determine whether the block stores data from string or the output of 2 given blocks.
- Divide string into block.
- Xor the two given block to have the third one.

1.2 Input and Output

Input: The input consists of a string having size divisible by 8. The user is prompted to enter a valid string, and the program will ask the user to enter again if that string is not valid.

Output: The program outputs the following:

- Three disks.
- The block of each disk.

1.3 Algorithm Workflow

The algorithm for processing the infix expression involves two main stages:

Stage 1: Input Handling

In this stage, our program checks the input whether its size is divisible by 8 or not. If not, It will ask users to input again.

1. Ask the user to input the string.
2. Count the length of the input.
3. Check the condition.

Stage 2: Divide input

1. Because the string's length is divisible by 8, we will divide it into chunk(s) each having 8 characters.
2. Each chunk is then divided into 2 blocks, each block having 4 characters.
3. Compute the other block by xor the two we have in the previous step.
4. Check the block to know whether it is the answer or the input from string to print [[]] or | | as in the question demonstration.
5. Repeat the previous step until all the string have been computed.

1.4 Conclusion

Overall, this simulation demonstrates the core principles of RAID 5, stimulate the operation of RAID 5 with 3 drives, assuming each data block has 4 characters.

2 READ THE BMP FILE AND DISPLAY ON THE BITMAP DISPLAY

This report describes the development of a RARS (RISC-V Assembler) program designed to read and display a BMP (Bitmap) image file on a Bitmap Display. The program lets the user specify the path of a BMP file, reads its content, processes the image data, and displays it on the screen. The maximum image resolution supported by the program is 512x512.

The report covers the key system calls used for file handling, the structure of the BMP file format, and the various operations involved in reading the image file and displaying it. By understanding these components, we gain insights into how low-level system programming interacts with hardware components such as the CPU, memory, and I/O devices.

2.1 Research on System Calls for File Operations

System calls are the fundamental interface between user applications and the operating system kernel. In this program, several system calls are used for file operations:

- `open`: This system call opens a file and returns a file descriptor that can be used in subsequent file operations.
- `read`: This system call reads data from a file and places it into a buffer in memory.
- `write`: Writes data to the screen or console.
- `lseek`: Moves the file pointer to a specified offset within the file.
- `close`: Closes an open file descriptor after the program has finished processing the file.

These system calls allow the program to interact with the operating system to open files, read data into memory, and write processed data to a display.

2.2 Research on the BMP File Format

The BMP (Bitmap) file format is a widely used image format that stores pixel data in a rasterized grid. It consists of a header and pixel data. The header provides metadata about the image, such as its dimensions, color depth, and compression format.

The BMP file consists of the following key sections:

- **File Header (14 bytes)**: This header contains information about the file type, the size of the file, and where the image data begins in the file.
- **DIB Header (40 bytes)**: Contains information about the image, including its width, height, and color depth.
- **Pixel Data**: After the headers, the pixel data follows, storing the color information for each pixel. The pixel data is typically organized in rows, with each row padded to a multiple of 4 bytes.

For this program, we focus on reading the header to ensure that the file is a valid BMP file and extracting the pixel data for display.

2.3 Program Overview

The RARS code follows these key steps:

1. The program prompts the user for the path of a BMP file.
2. It reads the BMP file, validates the file header to ensure it is in BMP format, and verifies that the image dimensions are within the acceptable range (up to 512x512 pixels).
3. The program reads the pixel data from the BMP file and stores it in memory.
4. It then displays the image on the Bitmap Display, where each pixel's color value is extracted and rendered.
5. Error handling is performed to ensure the program gracefully handles issues such as invalid file format, file access errors, or image size exceeding the maximum resolution.

2.4 Key Computer Architecture Concepts

The RARS program illustrates several important principles of computer architecture, including:

- **Memory Management:** Buffers are allocated in memory to hold the BMP file path, header, and pixel data.
- **System Calls:** System calls are used to perform file operations, interact with the operating system, and handle I/O.
- **Register Usage:** Registers are used to store file descriptors, image dimensions, and pixel data, allowing the program to interact with memory.
- **Error Handling:** The program checks for errors, such as invalid file formats or out-of-bounds image dimensions, and outputs appropriate error messages.

2.5 System Calls and I/O Operations

The program relies on the following system calls for file operations:

- `open`: Opens a file in read-only mode.
- `read`: Reads a specified number of bytes from the file into memory.
- `write`: Writes data to the console or screen memory.
- `lseek`: Moves the file pointer to a specific offset within the file for reading specific sections.
- `close`: Closes the file when done processing.

2.5.1 Example: Opening a File

The program opens the BMP file in read-only mode using the following RARS code:

```
# Open the file
open_file:
    li a7, 1024         # syscall: open
    la a0, buffer       # File path
    li a1, 0            # Read-only mode
    ecall
    mv t0, a0           # Save file descriptor in t0
    blt t0, zero, file_open_error  # If t0 < 0, file open failed
```

Here, the open system call is invoked with the file path stored in buffer. The file descriptor is stored in register t0, which is used for subsequent file operations.

2.6 Memory Management

Memory buffers are allocated for reading and processing the BMP file. The .space directive is used to allocate space for storing the file path, header, and pixel data:

```
buffer:         .space   256          # Buffer to hold the input file
   path
big_buffer:     .space   1100000      # Large buffer for header and
   pixels
```

These buffers store the file path, BMP header, and pixel data for processing.

2.6.1 Reading and Processing the BMP Header

The program reads the first 54 bytes of the BMP file, which contains the header. The header includes the image size, dimensions, and color format. The program validates the file by checking for the "BM" signature at the beginning of the file:

```
# Validate BMP file type
    la t2, big_buffer   # Address of the header
    lbu t3, 0(t2)       # First byte
    lbu t4, 1(t2)       # Second byte
    li t5, 'B'
    bne t3, t5, type_error
    li t5, 'M'
    bne t4, t5, type_error
```

The program compares the first two bytes with the ASCII values for 'B' and 'M'. If the bytes don't match, the program jumps to the type_error label to handle the error.

2.7 Pixel Data Processing

After validating the header, the program reads the pixel data. The pixel data is stored in the big_buffer and processed by the program. Each pixel consists of 3 bytes (BGR format), and the program extracts these values and displays them on the Bitmap Display.

```
# Read pixel data
    li s10, 3           # Each pixel is 3 bytes (BGR)
    mul s7, t4, s10     # rowSize = width * 3
    mul s8, s7, t5        # dataSize = rowSize * height
    la a1, big_buffer     # Address of big_buffer
    mv a0, t0             # File descriptor
    mv a2, s8             # Byte count
    li a7, 63             # syscall: read
    ecall
```

The program processes each pixel and displays it on the screen.

2.8 Error Handling

The program includes error handling routines for various potential errors, such as invalid file type or errors during file operations. For example:

```
file_open_error:
    li a7, 4
    la a0, error_open
    ecall
    j generic_error
```

In the case of an error (such as an inability to open the file), the program prints an error message and exits gracefully.

2.9 Conclusion

This RARS program provides a detailed example of how low-level system programming interacts with computer architecture. It demonstrates key concepts such as memory management, system calls, and file I/O operations. The program handles a BMP file by reading its header, validating the file format, processing pixel data, and displaying the image on a Bitmap Display.

Through the manipulation of memory buffers, registers, and system calls, the program offers a practical understanding of how a CPU communicates with memory and peripheral devices. It also emphasizes the importance of error handling, ensuring the program runs smoothly under different scenarios.

By working with this program, we can gain valuable insights into the underlying architecture of modern computers, including the critical role of memory management, file systems, and system-level programming.