Create OS

CSYE 6230

Seattle

Northeastern University

Team Members: Mengyun Xie, Xinyu Qiu

Instructor: Prof. Ahmed Banafa

Date: 04/16/2024

1. Purpose of the Operating System (OS)

The purpose of an operating system (OS) is to act as an intermediary between hardware and software, providing a platform for applications to run efficiently and managing computer resources. It provides an interface for users to interact with the computer and ensures that hardware resources are utilized effectively by software programs.

1.1 Functions of the Operating System

- Process Management: The OS manages processes, including process creation, scheduling, and termination, to ensure efficient use of CPU time.
- Memory Management: It manages system memory, allocating memory to processes and ensuring memory protection to prevent one process from accessing another's memory.
- File System Management: The OS provides file management functions, including:
    - Create
    - Update
    - Copy
    - Delete
    - List
- Device Management: It manages devices such as printers, disks, and networks, handling device communication and ensuring efficient device utilization.
- Security: The OS enforces security policies, controls access to system resources, and protects against unauthorized access and malware.

1.2 Number of Code Lines

The provided C++ program contains approximately 134 lines of code. It demonstrates basic file management operations (create, update, copy, delete, list) and includes a simulation of process scheduling. While it provides a simplified example, a full-fledged operating system would require millions of lines of code to implement all the necessary functionality and manage complex interactions between hardware and software.

In addition, there are 70 lines of code for the various configuration files.

2. Setting Up Development Environment

Install necessary tools and libraries:

```
sudo apt-get install build-essential nasm genisoimage qemu
```

- build-essential: contains the GNU compiler collection including gcc, g++, make, etc.
- nasm: the Netwide Assembler, a popular assembler for x86 architecture.
- genisoimage: for creating ISO files for CD-ROMs.
- qemu: a generic and open-source machine emulator and virtualizer, useful for running and testing.

3. Compiling the Operating System and Linking the Kernel

3.1 Create loader.s file

Implement bootloader functionality, preparing for kernel execution. This assembly file will serve as the bootloader, setting up essential registers and the protected mode before jumping to the kernel main function.

Code | Blame | 17 lines (14 loc) · 910 Bytes | Code 55% faster with GitHub Copilot

```
1        global loader                   ; the entry symbol for ELF
2
3        MAGIC_NUMBER equ 0x1BADB002     ; define the magic number constant
4        FLAGS        equ 0x0            ; multiboot flags
5        CHECKSUM     equ -MAGIC_NUMBER  ; calculate the checksum
6                                        ; (magic number + checksum + flags should equal 0)
7
8        section .text:                  ; start of the text (code) section
9        align 4                         ; the code must be 4 byte aligned
10           dd MAGIC_NUMBER             ; write the magic number to the machine code,
11           dd FLAGS                    ; the flags,
12           dd CHECKSUM                 ; and the checksum
13
14       loader:                         ; the loader label (defined as entry point in linker script)
15           mov eax, 0xCAFEBABE         ; place the number 0xCAFEBABE in the register eax
16       .loop:
17           jmp .loop                   ; loop forever
```

3.2 Create link.ld file

Define memory layout and sections for the kernel. The linker script that specifies the memory addresses where the kernel sections (text, data, bss) should be placed.

Code Blame 26 lines (21 loc) · 744 Bytes Code 55% faster with GitHub Copilot

```
ENTRY(loader)                /* the name of the entry label */

SECTIONS {
    . = 0x00100000;          /* the code should be loaded at 1 MB */

    .text ALIGN (0x1000) :   /* align at 4 KB */
    {
        *(.text)             /* all text sections from all files */
    }

    .rodata ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.rodata*)          /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) :   /* align at 4 KB */
    {
        *(.data)             /* all data sections from all files */
    }

    .bss ALIGN (0x1000) :    /* align at 4 KB */
    {
        *(COMMON)            /* all COMMON sections from all files */
        *(.bss)              /* all bss sections from all files */
    }
}
```

3.3 Create kernel.elf

```
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ ls
link.ld  loader.s
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$  nasm -f elf32 loader.s
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ ls
link.ld  loader.o  loader.s
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$  ld -T link.ld -melf_i386 loader.o -o kernel.elf
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ ls
kernel.elf  link.ld  loader.o  loader.s
```

The final executable will be called kernel.elf, which is the entry point for the operating system.

4. Building an ISO Image

4.1 Create Directory

```
mkdir -p iso/boot/grub              # create the folder structure
cp stage2_eltorito iso/boot/grub/   # copy the bootloader
cp kernel.elf iso/boot/             # copy the kernel
```

4.2 Create menu.lst Configuration for GRUB

Configure GRUB with necessary boot parameters and ensure it points to kernel.elf.

Code | Blame | 5 lines (4 loc) · 54 Bytes | Code 55% faster with GitHub Copilot

```
1  default=0
2  timeout=0
3
4  title os
5  kernel /boot/kernel.elf
```

```
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ ls
grub-0.97         iso         link.ld   loader.s
grub-0.97.tar.gz  kernel.elf  loader.o  stage2_eltorito
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ cd iso/boot/grub
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os/iso/boot/grub$ ls
stage2_eltorito
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os/iso/boot/grub$ sudo nano menu.lst
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os/iso/boot/grub$ ls
menu.lst  stage2_eltorito
```

4.3 Generate ISO Image

```
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ genisoimage -R
                -b boot/grub/stage2_eltorito    \
                -no-emul-boot                   \
                -boot-load-size 4               \
                -A os                           \
                -input-charset utf8             \
                -quiet                          \
                -boot-info-table                \
                -o os.iso                       \
                iso
ubuntu@ip-172-31-92-26:~/os/LittleOperatingSystems/os$ ls
grub-0.97  grub-0.97.tar.gz  iso  kernel.elf  link.ld  loader.o  loader.s  os.iso  stage2_eltorito
```

The ISO image os.iso now contains the kernel executable, the GRUB bootloader and the configuration file.

5. Implementing the Shell

5.1 Write Makefile

Makefile

```
CC = g++
CFLAGS = -std=c++11

SRCS = main.cpp
OBJS = $(SRCS:.cpp=.o)
TARGET = LittleOperatingSystems

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

.cpp.o:
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(OBJS) $(TARGET)
```

5.2 Write Shell Program

Develop a C++ program for the shell, parsing user commands for file operations.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <filesystem>

using namespace std;
namespace fs = std::filesystem;

struct Process {
    string name;
    int priority;
};

// Function to delete a file
void deleteFile(const string& file_name) {
    if (fs::remove(file_name)) {
        cout << "File '" << file_name << "' has been deleted." << endl;
    } else {
        cerr << "Error: Unable to delete file." << endl;
    }
}

// Function to copy a file
void copyFile(const string& source_file, const string& dest_file) {
    if (fs::copy_file(source_file, dest_file, fs::copy_options::overwrite_existing)) {
        cout << "File '" << source_file << "' copied to '" << dest_file << "'." << endl;
    } else {
        cerr << "Error: Unable to copy file." << endl;
    }
}

// Function to update content of a file
void updateFile(const string& file_name, const string& new_content) {
    ofstream file(file_name); // Open the file in truncate mode
    if (file.is_open()) {
        file << new_content;
        cout << "File content updated." << endl;
        file.close();
    } else {
        cerr << "Error: Unable to open file for updating content." << endl;
    }
}

// Function to list files in a directory
void listFiles(const string& directory) {
    for (const auto& entry : fs::directory_iterator(directory)) {
        cout << entry.path() << endl;
    }
}

int main() {
    std::cout << "\n- - - - - -" << std::endl;

    // Define a vector of processes
    vector<Process> processes = {
        {"Process A", 2},
        {"Process B", 1},
        {"Process C", 3}
    };

    // Simulate process scheduling
    sort(processes.begin(), processes.end(), [](const Process& a, const Process& b) {
        return a.priority < b.priority;
    });

    // Print the scheduled processes
    cout << "Process Scheduling:" << endl;
    for (const Process& process : processes) {
        cout << "Running " << process.name << " (Priority " << process.priority << ")" << endl;
    }

    std::cout << "\n- - - Create/Add File - - -" << std::endl;
    // Create and manipulate files
    string file_name = "input.txt";
    ofstream file(file_name);
    if (file.is_open()) {
        // Write content to the file
        file << "This is a simple file created by a process.";
        file.close();
    }

    // Check if the file exists
    if (fs::exists(file_name)) {
        cout << "File '" << file_name << "' exists." << endl;

        // Read the file
        ifstream file_in(file_name);
        string content;
        getline(file_in, content);
        cout << "File Content: " << content << endl;

        std::cout << "\n- - - Updated File - - -" << std::endl;
        // Update content of the file
        updateFile(file_name, "Updated content.");

        std::cout << "\n- - - Copy File - - -" << std::endl;
        // Copy the file
        copyFile(file_name, "input_copy.txt");

        std::cout << "\n- - - List File - - -" << std::endl;
        // List files in the directory
        cout << "Files in current directory:" << endl;
        listFiles(".");
    }

    std::cout << "\n- - - Delete File - - -" << std::endl;
    // Clean up by deleting the file
    deleteFile(file_name);

    std::cout << "\n- - - - - -" << std::endl;
    // Introduce the concept of system calls
    cout << "System Calls:" << endl;
    cout << "1. Process Creation: The operating system creates and manages processes." << endl;
    cout << "2. File Operations: OS provides APIs for file I/O." << endl;
    cout << "3. File Deletion: OS allows processes to delete files." << endl;
    cout << "4. System Calls: Processes make requests to the OS using system calls." << endl;

    std::cout << "\n- - - - - -" << std::endl;
    // Explain how the OS abstracts hardware
    cout << "Operating System Abstraction:" << endl;
    cout << "The OS abstracts hardware details, providing a uniform interface to processes." << endl;
    cout << "Processes interact with the OS through system calls, which manage resources." << endl;

    std::cout << "\n- - - - - -" << std::endl;
    // Conclude by summarizing the role of an OS
    cout << "In summary, an operating system:" << endl;
    cout << "- Manages processes and scheduling." << endl;
    cout << "- Provides file management and I/O operations." << endl;
    cout << "- Abstracts hardware details for processes." << endl;
    cout << "- Ensures system stability and security." << endl;

    return 0;
}
```

5.3 Compile the Shell Program

```
ubuntu@ip-172-31-92-26:~/LittleOperatingSystems$ make
g++ -std=c++17 -c main.cpp -o main.o
g++ -std=c++17 -o LittleOperatingSystems main.o
```

5.4 Running the Shell

```
ubuntu@ip-172-31-92-26:~/LittleOperatingSystems$ ./LittleOperatingSystems

- - - - - -
Process Scheduling:
Running Process B (Priority 1)
Running Process A (Priority 2)
Running Process C (Priority 3)

- - - Create/Add File - - -
File 'input.txt' exists.
File Content: This is a simple file created by a process.

- - - Updated File - - -
File content updated.

- - - Copy File - - -
File 'input.txt' copied to 'input_copy.txt'.

- - - List File - - -
Files in current directory:
"./input_copy.txt"
"./sample_copy.txt"
"./main.o"
"./input.txt"
"./link.ld"
"./.git"
"./doc"
"./LittleOperatingSystems"
"./loader.o"
"./Makefile"
"./README.md"
"./loader.s"
"./main.cpp"
"./os.iso"
"./iso"
"./.gitignore"
"./kernel.elf"

- - - Delete File - - -
File 'input.txt' has been deleted.

- - - - - -
System Calls:
1. Process Creation: The operating system creates and manages processes.
2. File Operations: OS provides APIs for file I/O.
3. File Deletion: OS allows processes to delete files.
4. System Calls: Processes make requests to the OS using system calls.

- - - - - -
Operating System Abstraction:
The OS abstracts hardware details, providing a uniform interface to processes.
Processes interact with the OS through system calls, which manage resources.

- - - - - -
In summary, an operating system:
- Manages processes and scheduling.
- Provides file management and I/O operations.
- Abstracts hardware details for processes.
- Ensures system stability and security.
```