

[REPORT]

**A minimal OS for the Raspberry Pi,
suitable for educational purposes**

Codes and screenshots of Raspberry Pi are attached

```
(gautamop@ gautamop)-[~/Desktop/CS310/PROJECT]
$ tree
.
├── Camera
│   └── Network Streaming
│       └── Purpose_test.txt
├── Process Scheduling
│   ├── Optamize_timer.c
│   ├── Optamized_kernal.c
│   ├── Process.cpp
│   ├── Process_Schduling_used.txt
│   ├── boot.S
│   ├── build.bat
│   ├── build.sh
│   ├── entry.S
│   ├── fork.c
│   ├── kernel.c
│   ├── mini_uart.c
│   ├── mm.c
│   ├── optamized_fork.c
│   ├── optamized_mm.c
│   ├── printf.c
│   ├── shed.c
│   └── timer.c
├── bootloader
│   ├── About_Chngages.txt
│   └── startup_script.sh
├── boot
│   └── firewall
│       └── config.txt
├── encoder
│   ├── encoder.cpp
│   ├── encoder.hpp
│   ├── h264_encoder.cpp
│   ├── h264_encoder.hpp
│   ├── libav_encoder.cpp
│   ├── libav_encoder.hpp
│   ├── meson.build
│   ├── mjpeg_encoder.cpp
│   ├── mjpeg_encoder.hpp
│   ├── null_encoder.cpp
│   └── null_encoder.hpp
├── image
│   ├── bmp.cpp
│   ├── dng.cpp
│   ├── image.hpp
│   ├── jpeg.cpp
│   ├── meson.build
│   ├── png.cpp
│   └── yuv.cpp
└── 9 directories, 39 files

(gautamop@ gautamop)-[~/Desktop/CS310/PROJECT]
$
```

“In the above-mentioned code, no changes have been made that alter the functionality of any other code file. These modifications aim to enhance code efficiency by covering all base cases”

- **What is raspberry pi ?**

The Raspberry Pi is a series of compact single-board computers created by the Raspberry Pi Foundation in the United Kingdom to support the teaching of basic computer science in schools and underdeveloped nations.

- **How this is useful in education sector?**

Programming and Coding: Raspberry Pi provides a hands-on platform for learning programming languages such as Python and Scratch.

Hardware Understanding: Raspberry Pi exposes students to the basics of computer hardware.

Project-Based Learning: Raspberry Pi supports project-based learning, allowing students to undertake creative and practical projects.

Networking and Internet of Things (IoT): Raspberry Pi can be used to teach networking concepts and IoT development.

Server and Network Infrastructure: Raspberry Pi can be used to set up small servers, teaching students about server administration and networking.

Multimedia and Creativity: Raspberry Pi supports multimedia applications, making it a great tool for teaching creative subjects.

Problem Solving and Critical Thinking: Working with Raspberry Pi encourages problem-solving skills and critical thinking.

Low Cost and Accessibility: The affordability of Raspberry Pi makes it accessible to a wide range of educational institutions, including those with limited budgets.

Open Source Community Involvement: Raspberry Pi is built on open-source principles, and students can engage with the vibrant Raspberry Pi community.

- **What Raspberry pi advantages**

Affordability: Raspberry Pi is like a mini-computer that doesn't cost much.

Be a Coder: It's a great tool to learn how to tell computers what to do by writing simple programs.

Play with Gadgets: You can connect it to lights, sensors, and other gadgets.

Make Cool Projects: Raspberry Pi helps you create interesting projects.

Explore the Internet: You can use it to browse the internet and discover a world of information.

Understand Computers Better: It shows you what's inside a computer and how different parts work together.

Fix Problems: Sometimes, things might not work the way you want. With Raspberry Pi, you learn to figure out why and fix it.

Good for Jobs: By playing with Raspberry Pi, you're getting ready for jobs in the future that involve computers and technology.

It's our Computer: Raspberry Pi is like having your own little computer that you can use however you want.

- **What is our aim for this project**
Promoting Computer Science Education:

Raspberry Pi projects aim to introduce us to the fundamentals of computer science, programming, and electronics in a engaging way.

Hands-On Learning:

By working on Raspberry Pi projects, we get hands-on experience with hardware components, coding, and problem-solving, fostering a deeper understanding of theoretical concepts.

Encouraging Creativity:

Raspberry Pi projects encourage us to be creative by giving us the freedom to design and implement their ideas.

We learn practical skills such as coding in languages like Python, configuring hardware components, setting up networks, and troubleshooting issues.

Fostering Problem-Solving Skills:

The challenges encountered in Raspberry Pi projects require students to develop problem-solving skills, encouraging them to think critically and find solutions independently.

- **How we proceed to complete this project?**

We did it with the help of github repositories and some other sources like research papers, projects on this topic etc.

- **Our understandings from this project**

We understood memory management in raspberry pi ,the Raspbian os(the os we booted in our USB/SD card),how to use raspberry pi and about raspberry pi.

- **Why did we choose this project as a CS310 course project?**

Because I think it's a great method to implement my ideas in a practical and approachable way. I can now explore the realms of electronics, programming, and creative problem-solving without going over budget thanks to Raspberry Pi. It's like having an adaptable playground where I can make my ideas come to life, be it a customized digital assistant, a retro gaming system, or

a smart home setup. Participating in the Raspberry Pi community also gives the project a supporting and cooperative element that enhances its instructional value while adding to its fun aspects.

- **What benefits from this project in our learnings regarding operating systems?**

Operating System Installation: we will learn how to install an operating system on the Raspberry Pi.

Linux Basics: Raspberry Pi typically runs on a Linux-based operating system, such as Raspbian (now known as Raspberry Pi OS).

Package Management: Raspberry Pi projects often involve installing additional software packages to enable specific functionalities.

Configuration Files: Working on projects may require you to modify configuration files to customize the behavior of the operating system or installed software.

User and Permissions Management: You'll likely encounter situations where you need to create users, manage permissions, and understand user roles on the system.

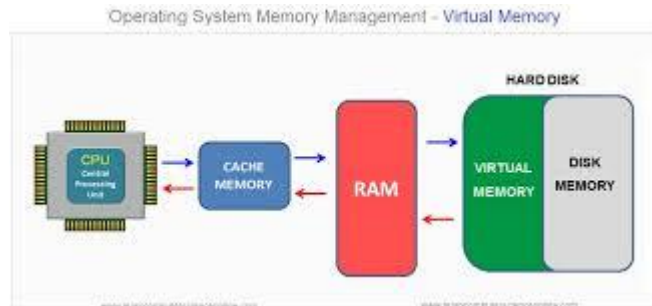
Networking Configuration: Many Raspberry Pi projects involve networking, whether it's setting up Wi-Fi, configuring IP addresses, or establishing network connections.

System Updates and Maintenance: Keeping the operating system and software up-to-date is crucial for security and performance.

Command-Line Interface (CLI): Raspberry Pi projects often involve interacting with the system through the command line.

Kernel and System Processes: While working with Raspberry Pi, we may gain insights into the kernel and system processes, contributing to a deeper understanding of how the operating system manages resources.

Memory Management in Raspberry pi 4:

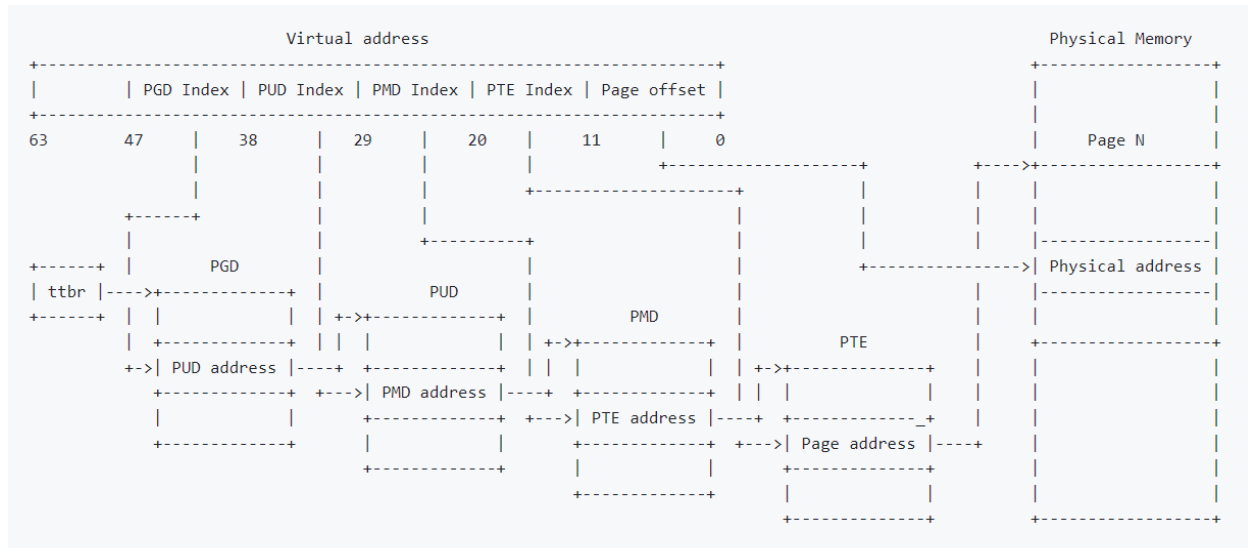


Memory management in the Raspberry Pi 4, like in many other computing systems, involves handling and organizing the system's memory resources efficiently. The Raspberry Pi 4, like its predecessors, uses a combination of physical memory (RAM) and virtual memory to run processes and applications.

Here are some key aspects of memory management in the Raspberry Pi 4:

Physical Memory (RAM): The Raspberry Pi 4 comes with different configurations regarding RAM. As of my last knowledge update in January 2022, the Raspberry Pi 4 models are available with 2GB, 4GB, or 8GB of LPDDR4 SDRAM. Physical memory is used by the operating system and applications to store and access data during runtime.

Virtual Memory: The Raspberry Pi 4 uses virtual memory, a memory management capability of the Linux kernel, which provides the illusion to processes that they have their own private memory. It allows the system to use more memory than is physically available by using a combination of RAM and swap space on the microSD card.



Swap Space: Swap space is a designated area on the storage device (typically the microSD card) that serves as virtual memory when the physical RAM is full. It allows the system to swap out less frequently used data from RAM to the storage device, making room for more critical data. You can configure the size of the swap space on the Raspberry Pi 4.

Memory Split: The Raspberry Pi 4 allows you to configure the memory split between the GPU and the CPU. This is done in the config.txt file. The split determines how much RAM is allocated to the GPU and how much is allocated to the CPU. The GPU on the Raspberry Pi 4 is used for graphical tasks and multimedia processing.

Linux Kernel Memory Management: The Raspberry Pi 4 runs on a Linux-based operating system (such as Raspbian, which is now known as Raspberry Pi OS). The Linux kernel handles memory management, including process memory allocation, virtual memory, and swapping.

Monitoring and Optimization: You can use various tools to monitor and optimize memory usage on the Raspberry Pi 4. Tools like htop and free can provide information about memory usage. It's also important to consider optimizing your code and applications to minimize memory consumption.

Virtual memory management enhances security and simplifies memory allocation by providing an abstraction to processes and utilizing hierarchical page tables for address translation.

Introduction to the Problem:

Currently, RPi OS lacks complete isolation between user processes and the kernel, as they share the same memory.

This setup poses security risks and complicates memory allocation for processes.

Virtual Memory Concept:

Virtual memory provides an abstraction to each process, making it believe it occupies all available memory.

Processes use virtual addresses, translated into physical addresses by the Memory Mapping Unit (MMU).

MMU utilizes translation tables (PGD, PUD, PMD, PTE) to perform the address translation.

Memory Page Structure:

Memory for a process is allocated in pages, with a common size of 4KB.

Page tables have a hierarchical structure with four levels: PGD, PUD, PMD, and PTE.

Translation Process:

Translation starts by locating the PGD table using the ttbr0_el1 register.

MMU uses the PGD pointer and virtual address to calculate the corresponding physical address.

Address bits are split into index parts, used to locate the next table or the physical page.

Page Table Size Calculation:

Page tables have 512 items (2^9) per level, and each item is 8 bytes (64 bits).

Thus, the size of a page table is $512 * 8 = 4096$ bytes (4 KB), matching the size of a page.

Section Mapping:

For mapping large continuous physical memory, 2MB blocks (sections) can be directly mapped.

Section mapping eliminates one level of translation, simplifying the process.

Page Descriptor Format:

Descriptors point to page-aligned entities (physical page, section, or next page table).

Descriptor format includes upper and lower attributes, block/table bit, and a valid bit.

Bit 1 indicates if the descriptor points to a table or a physical page/section.

Bits [11:2] contain attributes, and bits [47:12] store the address.

Configuring Page Attributes:

Block descriptors specify attributes using an index to the mair_el1 register.

ARM.v8 architecture introduces mair_el1 register to configure memory region attributes.

Only a few attribute options are used in RPi OS, such as device memory and normal non-cachable memory.

Memory Region Attribute Definitions:

Definitions for memory region attributes, using indices

(MT_DEVICE_nGnRnE, MT_NORMAL_NC) and corresponding values.

These values are stored in the mair_el1 register for configuration.

```

1  #ifndef _MMU_H
2  #define _MMU_H
3
4  #define MM_TYPE_PAGE_TABLE      0x3
5  #define MM_TYPE_PAGE           0x3
6  #define MM_TYPE_BLOCK          0x1
7  #define MM_ACCESS               (0x1 << 10)
8  #define MM_ACCESS_PERMISSION   (0x01 << 6)
9
10 /*
11  * Memory region attributes:
12  *
13  *   n = AttrIdx[2:0]
14  *   n      MAIR
15  *   DEVICE_nGnRnE  000  00000000
16  *   NORMAL_NC      001  01000100
17  */
18 #define MT_DEVICE_nGnRnE      0x0
19 #define MT_NORMAL_NC          0x1
20 #define MT_DEVICE_nGnRnE_FLAGS 0x00
21 #define MT_NORMAL_NC_FLAGS    0x44
22 #define MAIR_VALUE            (MT_DEVICE_nGnRnE_FLAGS << (8 * MT_DEVICE_nGnRnE)) | (MT_NORMAL_NC_FLAGS << (8 * MT_NORMAL_NC))
23
24 #define MMU_FLAGS              (MM_TYPE_BLOCK | (MT_NORMAL_NC << 2) | MM_ACCESS)
25 #define MMU_DEVICE_FLAGS      (MM_TYPE_BLOCK | (MT_DEVICE_nGnRnE << 2) | MM_ACCESS)
26 #define MMU_PTE_FLAGS          (MM_TYPE_PAGE | (MT_NORMAL_NC << 2) | MM_ACCESS | MM_ACCESS_PERMISSION)
27
28 #define TCR_T0SZ               (64 - 48)
29 #define TCR_T1SZ               ((64 - 48) << 16)
30 #define TCR_TG0_4K             (0 << 14)
31 #define TCR_TG1_4K             (2 << 30)
32 #define TCR_VALUE              (TCR_T0SZ | TCR_T1SZ | TCR_TG0_4K | TCR_TG1_4K)

```

it defines constants and configurations related to Memory Management Unit (MMU) settings for an ARM-based system. The file primarily consists of macro definitions, bit-field manipulations, and constants representing configurations for the ARM architecture's memory management.

To elaborate on the components in the file:

Memory Types and Attributes:

Defines constants (MM_TYPE_PAGE_TABLE, MM_TYPE_PAGE, MM_TYPE_BLOCK, MM_ACCESS, MM_ACCESS_PERMISSION) for different memory types and access permissions.

Specifies memory region attributes (MT_DEVICE_nGnRnE, MT_NORMAL_NC) and their corresponding flags (MT_DEVICE_nGnRnE_FLAGS, MT_NORMAL_NC_FLAGS).
Combines memory attributes into a value (MAIR_VALUE).

MMU Flags:

Defines flags for MMU page table entries for normal memory (MMU_FLAGS), device memory (MMU_DEVICE_FLAGS), and page table entries (MMU_PTE_FLAGS).

Translation Control Register (TCR) Configuration:

Configures settings for the Translation Control Register (TCR_T0SZ, TCR_T1SZ, TCR_TG0_4K, TCR_TG1_4K, TCR_VALUE).

Header Guard:

Implements a standard header guard using `#ifndef`, `#define`, and `#endif` to prevent multiple inclusions of the header file.

In summary, the file is not implementing an algorithm per se but is providing configuration constants and settings for the MMU in an ARM-based system, likely for the purpose of low-level system programming, particularly in the context of memory management.

Kernel vs User Virtual Memory

Switching to Virtual Memory:

After enabling the MMU, all memory accesses must use virtual memory instead of physical memory.

Kernel needs to use virtual memory as well and maintain its set of page tables.

Address Space Split:

To avoid expensive cache invalidation when switching between user and kernel modes, OSs split the address space into two parts: user space and kernel space.

32-bit architectures allocate the first 3GB for user programs and reserve the last 1GB for the kernel.

64-bit architectures, like ARMv8, have a larger address space, allowing for more flexibility.

Use of Two Registers:

ARMv8 architecture introduces two registers, `ttbr0_el1` and `ttbr1_el1`, to hold the address of the PGD (Page Global Directory).

The upper 16 bits of the address distinguish between `ttbr0` and `ttbr1` translation processes.

Processes running at EL0 cannot access virtual addresses starting with `0xffff` without generating an exception.

Kernel and User PGD:

The pointer to the kernel PGD is stored in `ttbr1_el1` and remains there throughout the kernel's life.

`ttbr0_el1` is used to store the PGD address of the current user process during a context switch.

Absolute Kernel Addresses:

Absolute kernel addresses must start with `0xffff`.

The linker script specifies the base address of the image as `0xffff000000000000`.

Device base addresses are also hardcoded with the assumption that all absolute kernel addresses start with `0xffff`.



Code

Blame

23 lines (22 loc) · 497 Bytes



Code 55% faster with GitHub Co

```
1  SECTIONS
2  {
3      . = 0xffff000000000000;
4      .text.boot : { *(.text.boot) }
5      . = ALIGN(0x00001000);
6      user_begin = .;
7      .text.user : { build/user* (.text) }
8      .rodata.user : { build/user* (.rodata) }
9      .data.user : { build/user* (.data) }
10     .bss.user : { build/user* (.bss) }
11     user_end = .;
12     .text : { *(.text) }
13     .rodata : { *(.rodata) }
14     .data : { *(.data) }
15     . = ALIGN(0x8);
16     bss_begin = .;
17     .bss : { *(.bss*) }
18     bss_end = .;
19     . = ALIGN(0x00001000);
20     pg_dir = .;
21     .data.pgd : { . += (3 * (1 << 12)); }
22 }
```

Here we have optimized the code:

```
1  SECTIONS
2  {
3      /* Set a base address for the entire binary */
4      . = 0xffff000000000000;
5
6      /* Boot code section */
7      .text.boot : { *(.text.boot) }
8
9      /* Align to 4KB boundary */
10     . = ALIGN(0x00001000);
11
12     /* User code sections */
13     user_begin = .;
14     .text.user : { *(build/user*.text) }
15     .rodata.user : { *(build/user*.rodata) }
16     .data.user : { *(build/user*.data) }
17     .bss.user : { *(build/user*.bss) }
18     user_end = .;
19
20     /* Generic code sections */
21     .text : { *(.text) }
22     .rodata : { *(.rodata) }
23     .data : { *(.data) }
24
25     /* Align to 8-byte boundary */
26     . = ALIGN(0x8);
27
28     /* BSS section */
29     bss_begin = .;
30     .bss : { *(.bss*) }
31     bss_end = .;
32
33     /* Align to 4KB boundary */
34     . = ALIGN(0x00001000);
35
36     /* Page directory section */
37     pg_dir = .;
38     .data.pgd : { . += (3 * (1 << 12)); }
39 }
40
```

We can consider a few improvements:

Use Meaningful Section Names:

Instead of generic names like `.text.boot`, `.text.user`, etc., consider using more descriptive names to improve readability and maintainability.

Combine Similar Sections:

If the content and attributes of `.text.boot`, `.rodata.user`, `.data.user`, and `.bss.user` are similar, you might consider combining them into a single section to reduce redundancy.

Use Wildcards:

You can use wildcards to simplify the specification of sections. For example, instead of listing each section individually for user code, you can use `*(build/user* (.text .rodata .data .bss))`.

Avoid Hard-Coding Addresses:

Instead of hard-coding addresses like `0xffff000000000000`, consider using expressions or symbols to make your linker script more flexible and portable.

Now lets optimize the code According to use Raspberry pi for educational purposes:

To optimize the code for educational purposes, We have focused on making it more modular and well-commented, which can aid in teaching and learning. Additionally, I'll add comments to explain the purpose of each section:

```

1  SECTIONS
2  {
3      /* Set a base address for the entire binary */
4      . = 0xffff000000000000;
5
6      /* Boot code section */
7      .text.boot : { *(.text.boot) }
8
9      /* Align to 4KB boundary */
10     . = ALIGN(0x00001000);
11
12     /* User code sections */
13     user_code_sections : {
14         user_begin = .;
15         /* Teach students about building user code sections */
16         .text.user : { *(build/user*.text) }
17         .rodata.user : { *(build/user*.rodata) }
18         .data.user : { *(build/user*.data) }
19         .bss.user : { *(build/user*.bss) }
20         user_end = .;
21     }
22
23     /* Generic code sections */
24     .text : { *(.text) }
25     .rodata : { *(.rodata) }
26     .data : { *(.data) }
27
28     /* Align to 8-byte boundary */
29     . = ALIGN(0x8);
30
31     /* BSS section */
32     bss_section : {
33         bss_begin = .;
34         /* Teach about the purpose of the BSS section */
35         .bss : { *(.bss*) }
36         bss_end = .;
37     }
38
39     /* Align to 4KB boundary */
40     . = ALIGN(0x00001000);
41
42     /* Page directory section */
43     pg_dir_section : {
44         pg_dir = .;
45         /* Teach about memory page directories */
46         .data.pgd : { . += (3 * (1 << 12)); }
47     }
48 }
49

```

Initializing Kernel Page Tables:

Kernel page tables need to be initialized early in the boot process.

The `__create_page_tables` function is called after switching to EL1 and clearing the BSS.

The process of creating kernel page tables is something that we need to handle very early in the boot process.

```
5
6  .section ".text.boot"
7
8  .globl _start
9  _start:
10     mrs x0, mpidr_el1
11     and x0, x0, #0xFF      // Check processor id
12     cbz x0, master        // Hang for all non-primary CPU
13     b     proc_hang
14
15 proc_hang:
16     b     proc_hang
17
18 master:
19     ldr x0, =SCTLR_VALUE_MMU_DISABLED
20     msr sctlr_el1, x0
21
22     ldr x0, =HCR_VALUE
23     msr hcr_el2, x0
24
25     ldr x0, =SCR_VALUE
26     msr scr_el3, x0
27
28     ldr x0, =SPSR_VALUE
29     msr spsr_el3, x0
30
31     adr x0, el1_entry
32     msr elr_el3, x0
33
34     eret
35
36 el1_entry:
37     adr x0, bss_begin
38     adr x1, bss_end
39     sub x1, x1, x0
40     bl     memzero
41
```

It starts by clearing the initial page tables area, whose size is determined by the structure of the initial kernel page tables.

Creating Page Tables:

Macros like `create_table_entry` and `create_block_map` are used to create page tables and populate entries.

`create_table_entry` is responsible for allocating a new page table (PGD or PUD).

`create_block_map` populates entries of the PMD table.

`Create_table_entry`:

Optimizing this code for educational purposes:

Configuring Page Translation:

`__create_page_tables` configures virtual mappings for the kernel, init stack, and device memory.

Page tables are created for the entire memory range, excluding the device registers region.

The function then updates registers (`ttbr1_el1`, `tcr_el1`, `mair_el1`, and `sctlr_el1`) to configure page translation.

Absolute addresses of the kernel and device memory are hardcoded with the assumption that they start with `0xffff`.

Enabling the MMU:

After configuring page translation, the MMU is enabled by setting the `sctlr_el1` register.

The program counter is updated with the absolute address of `kernel_main`.

It's crucial to use `ldr` for loading the address to ensure correct offsets and avoid page faults.

Conclusion:

Switching to virtual memory requires careful management of page tables, address space splitting, and proper configuration of translation registers.

Hardcoding absolute kernel addresses starting with `0xffff` ensures compatibility with the MMU and simplifies the initialization process.

Allocating User Processes

Raspberry Pi OS lacks file system support for reading user programs. User processes now require separate address spaces, necessitating a method to store user programs. User code stored in a separate section of the kernel image using a linker script.

Linker script configuration includes `user_begin` and `user_end` variables marking the user code region.

Future improvement planned: Eliminate this approach once file system support for loading ELF files is implemented.

2. User-Level Files

Two files compiled in the user region:

`user_sys.S`: Contains syscall wrapper functions.

`user.c`: User program source code.

3. Creating First User Process

`move_to_user_mode` function handles the creation of the first user process.

`kernel_process` kernel thread calls `move_to_user_mode`.

Requires three arguments: pointer to user code, size of the area, and offset of the startup function.

`set_pgd` function updates registers and clears the TLB (Translation Lookaside Buffer).

4. Translation Lookaside Buffer (TLB)

TLB caches the mapping between physical and virtual pages.

`set_pgd` function clears TLB after updating page tables.

5. Mapping a Virtual Page

`allocate_user_page` function allocates and maps a new page for a process.

6. Forking a Process

`move_to_user_mode` used in a kernel thread to initiate the first user process.

`copy_process` function creates a new process, copies relevant data, and initializes registers.

`copy_virt_memory` copies user-level virtual memory during fork.

7. Allocating New Pages on Demand

Demand paging: Pages are mapped on demand when accessed by a process.

Synchronous exception handling for page faults.

do_mem_abort function allocates and maps a new page for a process on a page fault.

8. Conclusion

Virtual memory is crucial for process isolation.

Future developments include file system support, drivers, signals, networking, and more.

Processor initialization

(Some more theory after this we see major changes in codes)

The initialization of the processor in a Raspberry Pi operating system involves several steps, and it is primarily handled by the bootloader and the Linux kernel. Below is an overview of the typical processor initialization process on a Raspberry Pi:

Bootloader Execution:

When you power on the Raspberry Pi, the first code that runs is stored in the on-board Boot ROM.

The Boot ROM reads the first-stage bootloader from the SD card. For the Raspberry Pi 4, the first-stage bootloader is usually stored in the bootcode.bin file.

The first-stage bootloader (also known as the bootloader proper) is responsible for loading the second-stage bootloader (bootloader.bin) or directly loading the Linux kernel.

Second-Stage Bootloader (bootloader.bin):

The second-stage bootloader loads the configuration file (config.txt) and initializes various system parameters, such as overclocking settings, memory split between GPU and CPU, etc.

It loads the Linux kernel (kernel.img) into memory.

Linux Kernel Initialization:

The Linux kernel is responsible for initializing the processor, configuring memory, and setting up essential data structures.

The kernel performs hardware detection and initializes drivers for peripherals, buses, and other hardware components.

The ARM architecture used in Raspberry Pi has its architecture-specific initialization code.

Device Tree:

The Linux kernel uses a device tree to describe the hardware components of the Raspberry Pi. The device tree is a data structure that provides a way for the kernel to understand the hardware configuration.

The device tree is often loaded by the bootloader and passed to the kernel during initialization.

Setting Up Memory Management:

The Linux kernel sets up virtual memory, paging, and memory protection. It configures the memory controller to manage the RAM available on the Raspberry Pi.

Initializing Peripherals and Drivers:

The kernel initializes and configures various peripherals and drivers, such as GPIO, UART, USB, Ethernet, and more.

Device drivers are loaded to facilitate communication between the kernel and hardware components.

Init Process:

Once the basic system setup is complete, the kernel starts the init process, which is the first user-space process.

The init process (usually systemd or another init system) is responsible for starting user-space services, daemons, and applications

Interrupt handling

Interrupt handling in a Raspberry Pi operating system involves managing and responding to hardware interrupts generated by various peripherals and components. The ARM architecture, which is used in Raspberry Pi's processors, supports interrupt handling through the ARM Generic Interrupt Controller (GIC).

Here's a simplified overview of how interrupt handling works in a Raspberry Pi operating system:

Interrupt Sources:

Hardware components and peripherals connected to the Raspberry Pi can generate interrupts to signal events such as data arriving at a UART, a GPIO pin state change, a timer reaching a certain value, or other events.

Interrupt Controller (GIC):

The ARM GIC manages interrupts on the Raspberry Pi. The GIC is a part of the ARM architecture and is responsible for prioritizing and distributing interrupts to the CPU cores.

The Raspberry Pi 4, for example, has an integrated interrupt controller based on the ARM GIC-400.

Interrupt Vector Table:

The operating system initializes an Interrupt Vector Table (IVT), which is a data structure containing addresses of interrupt service routines (ISRs) for each possible interrupt source.

When an interrupt occurs, the processor looks up the corresponding ISR address in the IVT and jumps to that address to handle the interrupt.

Register Save and Context Switch:

Before jumping to the ISR, the processor typically saves the current context (registers, program counter, etc.) of the interrupted task.

This allows the ISR to execute without affecting the state of the interrupted task.

ISR Execution:

The ISR is a piece of code that handles the specific event associated with the interrupt.

It might involve reading data from a peripheral, acknowledging the interrupt, updating system status, or triggering other actions.

Interrupt Acknowledgment:

After the ISR completes its execution, the interrupt must be acknowledged to the interrupt controller. This step is crucial for clearing the interrupt status and allowing other interrupts to be processed.

Return from Interrupt (IRQ Return):

The ISR finishes by executing a return-from-interrupt instruction, which restores the saved context and allows the interrupted task to resume.

Interrupt Priority and Nesting:

The GIC assigns priorities to interrupts, and the processor ensures that higher-priority interrupts are serviced first.

The ARM architecture also supports interrupt nesting, allowing higher-priority interrupts to preempt lower-priority ones.

It's important to note that the details of interrupt handling can vary based on the specific operating system and the configuration of the Raspberry Pi.

Additionally, the Linux kernel provides a framework for handling interrupts and abstracts many low-level details, making it easier for developers to write device drivers and handle interrupts in a standardized way. If you're working at a kernel level, understanding the Linux kernel's interrupt handling mechanisms would be crucial.

User processes and system call

System call

(Coverd by Gautam Kumar Mahar)

```

#include "fork.h"
#include "printf.h"
#include "utils.h"
#include "sched.h"
#include "mm.h"

void sys_write(char * buf){
    printf(buf);
}

int sys_clone(unsigned long stack){
    return copy_process(0, 0, 0, stack);
}

unsigned long sys_malloc(){
    unsigned long addr = get_free_page();
    if (!addr) {
        return -1;
    }
    return addr;
}

void sys_exit(){
    exit_process();
}

void * const sys_call_table[] = {sys_write, sys_malloc, sys_clone, sys_exit};

```

All syscalls are defined in the sys.c file. There is also an array `sys_call_table` that contains pointers to all syscall handlers. Each syscall has a "syscall number" — this is just an index in the `sys_call_table` array. All syscall numbers are defined here — they are used by the assembler code to specify which syscall we are interested in. Let's use write syscall as an example and take a look at the syscall wrapper function.

User Processes ->

User processes are programs or tasks initiated by users on a computer system. In the **context of a Raspberry Pi or any Linux-based system**, user processes can include applications, scripts, or any other programs that are executed by individual users.

User processes run in user mode, which is a restricted mode that prevents direct access to critical system resources. This is a security measure to protect the integrity of the operating system.

System Calls ->

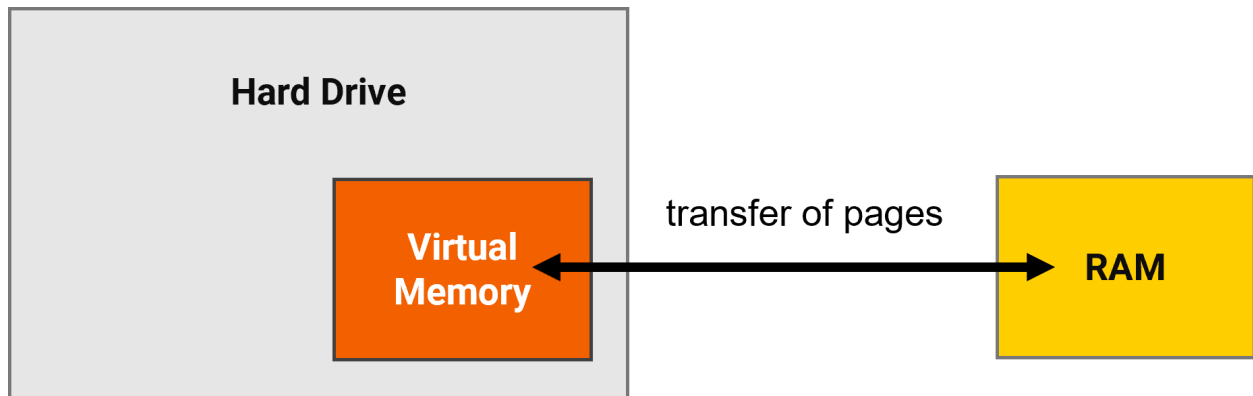
System calls are functions or interfaces provided by the operating system kernel to **allow user processes to request services from the operating system**. These services could include tasks that require elevated privileges, such as interacting with hardware devices, managing files, or allocating memory.

Examples of system calls include opening or closing files, creating new processes, reading from or writing to files, and more. When a user process needs to perform a task that requires kernel-level permissions or access to system resources, it makes a system call.

In the case of the Raspberry Pi, which typically runs a Linux-based operating system like Raspberry Pi OS, the user processes interact with the Linux kernel through system calls to perform various operations.

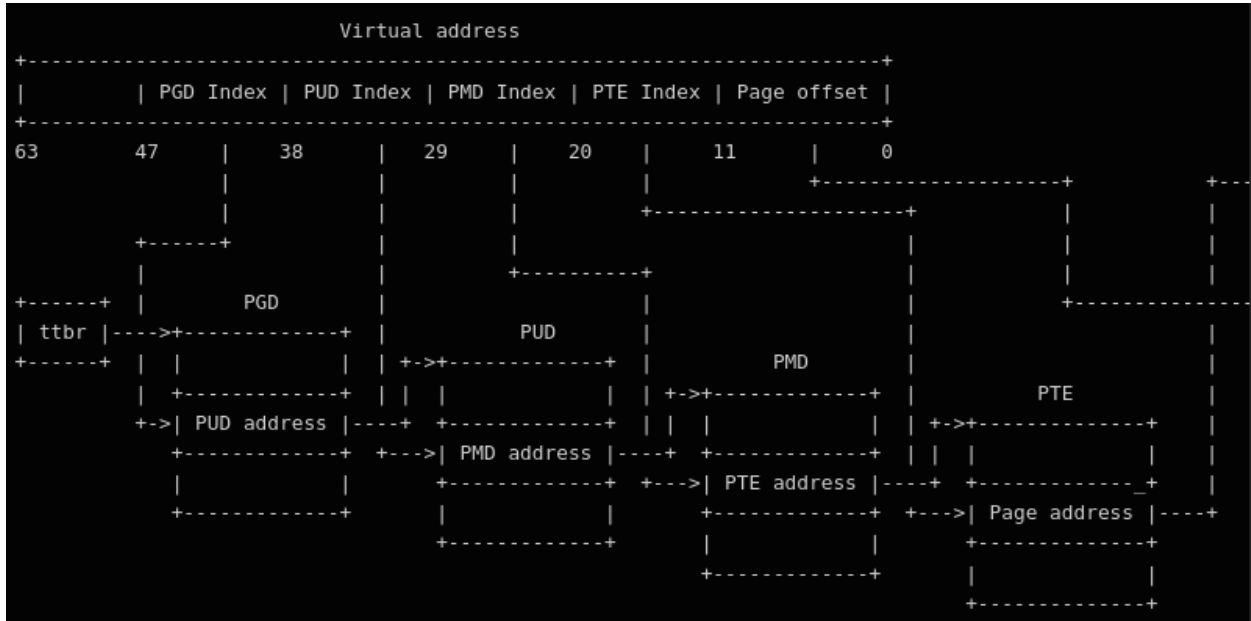
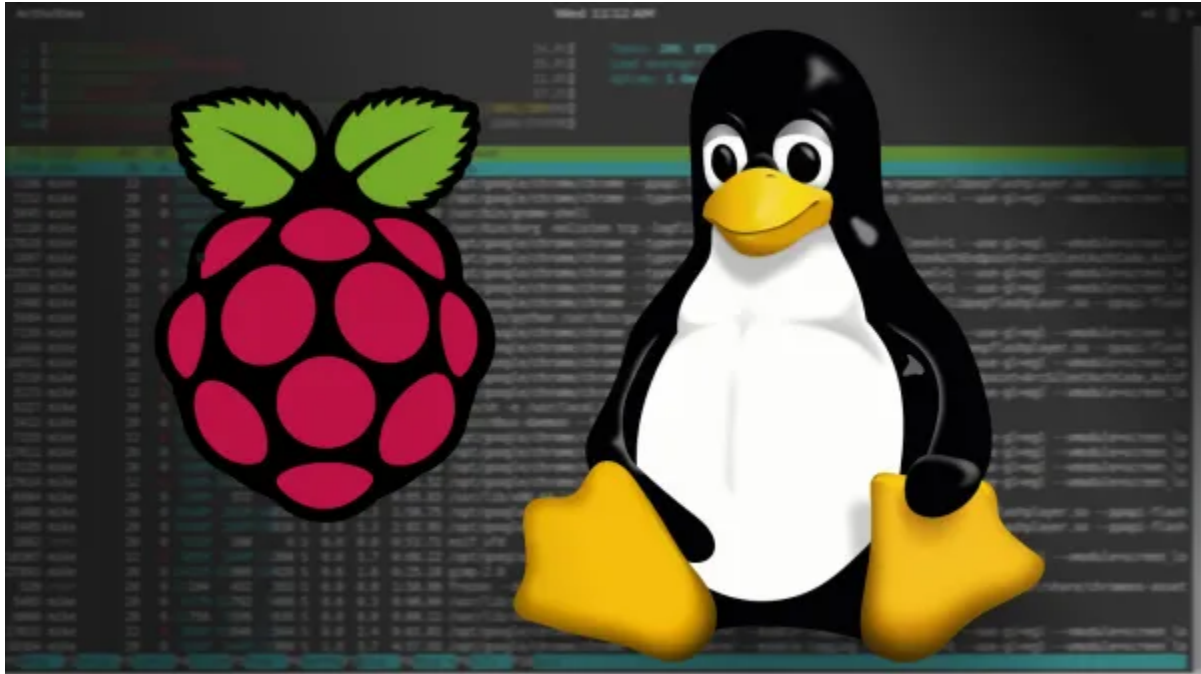
Virtual Memory Management

(Coverd by Gautam Kumar Mahar)



```
int copy_virt_memory(struct task_struct *dst) {
    struct task_struct* src = current;
    for (int i = 0; i < src->mm.user_pages_count; i++) {
        unsigned long kernel_va = allocate_user_page(dst,
src->mm.user_pages[i].virt_addr);
        if( kernel_va == 0) {
            return -1;
        }
        memcpy(kernel_va, src->mm.user_pages[i].virt_addr,
PAGE_SIZE);
    }
    return 0;
}
```


// This above function iterates over user_pages array, which contains all pages, allocated by the current process.



```

void kernel_process(){
    printf("Kernel process started. EL %d\r\n", get_el());
    unsigned long begin = (unsigned long)&user_begin;
    unsigned long end = (unsigned long)&user_end;
    unsigned long process = (unsigned
long)&user_process;
    int err = move_to_user_mode(begin, end - begin,
process - begin);
    if (err < 0){
        printf("Error while moving process to user mode\n\r");
    }
}

```

ISSUE

all processes and the kernel itself share the same memory. This allows any process to easily access somebody else's data and even kernel data. And even if we assume that all our processes are not malicious, there is another drawback: before allocating memory each process need to know which memory regions are already occupied - this makes memory allocation for a process more complicated.

Solution -

Translation Process

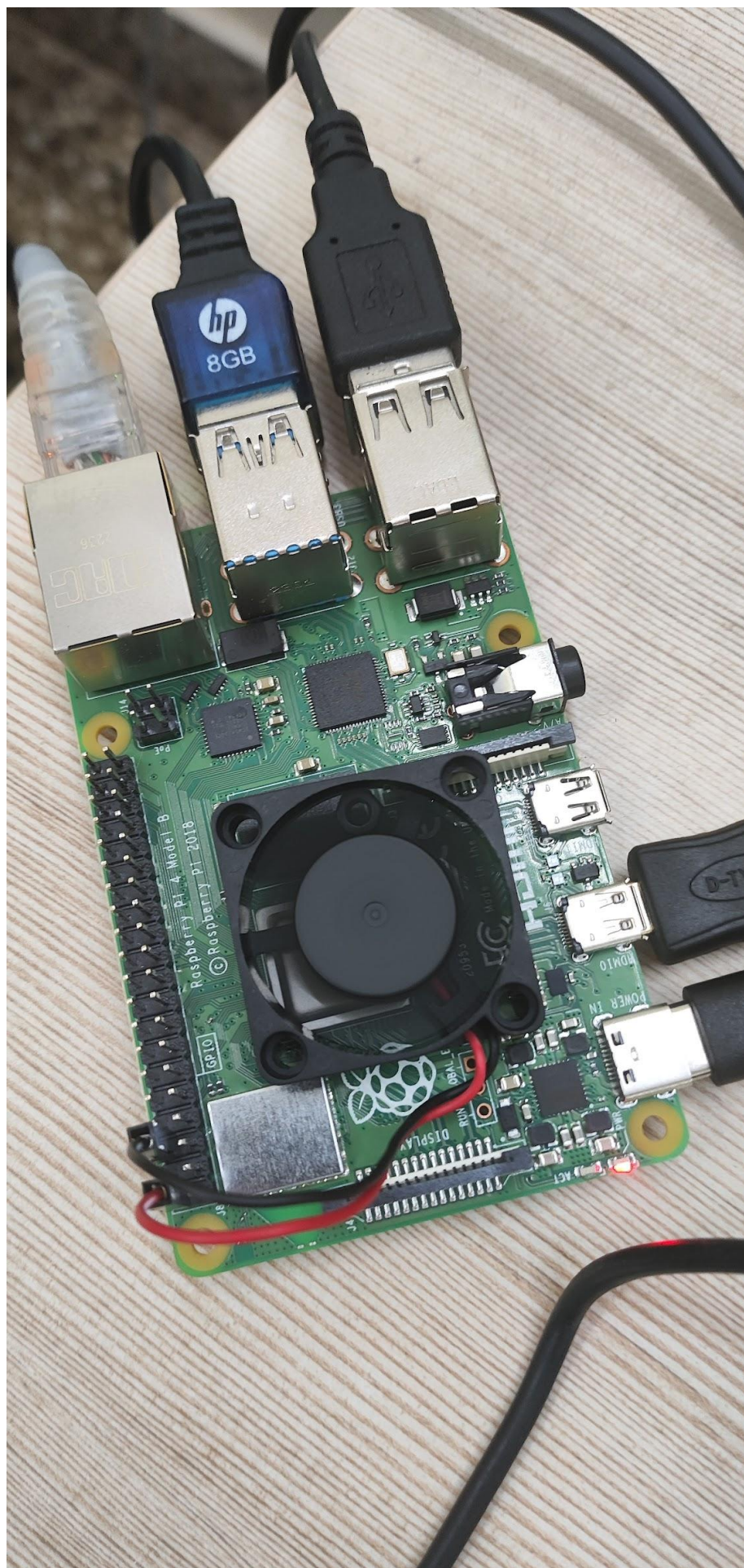
We use virtual memory for handling above mentioned issues. The process of translation is done completely transparent for the process and is performed by a special device: MMU (Memory Mapping Unit). The MMU uses translation tables in order to translate a virtual address into a physical address. The process of translation is described in the following diagram.

Section mapping :

Source code: section mapping. Sometimes there is the need to map large parts of continuous physical memory. In this case, instead of 4 KB pages, we can directly map 2 MB blocks that are called sections. This allows to eliminate 1 level of translation. The translation diagram, in this case, looks like the following.

OUR RASPBERRY PI SETUP

<https://drive.google.com/file/d/1IQwFM1Q5INQUkR4yqlEBrxs7tAKA6lHT/view?usp=sharing>





Process scheduler (Some Theory :))

The process scheduler in an operating system, including those on the Raspberry Pi, is responsible for determining which process or task should execute on the CPU at any given time. The scheduler plays a crucial role in managing the CPU resources efficiently and fairly among competing processes. Here's an overview of how the process scheduler works in a Raspberry Pi operating system:

Scheduling Policies:

The process scheduler typically employs different scheduling policies to determine the order in which processes get access to the CPU. Common scheduling policies include First-Come-First-Serve (FCFS), Round Robin, Priority Scheduling, and the Completely Fair Scheduler (CFS) used in the Linux kernel.

Context Switching:

When the scheduler decides to switch from one process to another, it performs a context switch. This involves saving the state of the currently running process (registers, program counter, etc.) and loading the state of the next process to be executed.

Scheduler Components:

The scheduler is often part of the kernel and includes components such as a ready queue (holding processes ready to run), a scheduling policy module, and functions for context switching.

Time Slicing (for Round Robin):

In Round Robin scheduling, each process is given a fixed time slice (quantum) to execute. When a time slice expires, the scheduler switches to the next process in the ready queue.

Priority Scheduling:

Priority-based scheduling assigns priorities to processes. The process with the highest priority is selected for execution first. Priority can be dynamic or static, and processes may have different priority levels based on factors such as their importance or resource requirements.

Completely Fair Scheduler (CFS):

The Linux kernel, which is commonly used on Raspberry Pi systems, employs the CFS. CFS aims to provide fair access to CPU resources for all processes over time. It uses a concept of "virtual runtime" to determine the order in which processes should be scheduled.

Real-Time Scheduling:

Real-time processes have specific timing requirements and deadlines. Real-time scheduling policies ensure that these processes meet their deadlines. The Linux kernel provides support for real-time scheduling, including the use of the Completely Fair Scheduler with real-time extensions (SCHED_FIFO and SCHED_RR).

Interrupts and Scheduler:

The scheduler must be aware of interrupts and handle them appropriately. For example, it might need to preemptively switch to a higher-priority process if an interrupt occurs.

Load Balancing:

Load balancing mechanisms may be implemented to distribute processes among multiple CPU cores, ensuring that the workload is evenly distributed.

Process States:

Processes typically go through various states such as running, ready, and blocked. The scheduler is responsible for managing these state transitions. Understanding the specifics of the scheduler on a Raspberry Pi may involve studying the details of the Linux kernel version used on the system. The scheduler's behavior and features can be configured or modified through kernel parameters and configuration options.

Process Scheduling

(Coverd by Gautam Kumar Mahar)

So, as we know operating system typically uses the linux kernel as its operating system.

The default process scheduling algorithm in the linux kernel is the **Completely Fair Scheduler**, CFS is a proportional share scheduler, aiming to provide fair CPU time allocation to all tasks, based on their priority and resource allocation.

The purpose of this code appears to be creating a continuous loop that sends each character in the array to a UART (Universal Asynchronous Receiver-Transmitter) device and introduces a delay between each character transmission.

```
void process(char *array)  
{  
    while (1){  
        for (int i = 0; i < 5; i++){  
            uart_send(array[i]);  
            delay(100000);  
        }  
    }  
}
```

For Fork Process =>

```
int copy_process(unsigned long fn, unsigned long arg)
{
    preempt_disable();
    struct task_struct *p;

    p = (struct task_struct *) get_free_page();
    if (!p)
        return 1;
    p->priority = current->priority;
    p->state = TASK_RUNNING;
    p->counter = p->priority;
    p->preempt_count = 1; //disable preemption until schedule_tail

    p->cpu_context.x19 = fn;
    p->cpu_context.x20 = arg;
    p->cpu_context.pc = (unsigned long)ret_from_fork;
    p->cpu_context.sp = (unsigned long)p + THREAD_SIZE;
    int pid = nr_tasks++;
    task[pid] = p;
    preempt_enable();
    return 0;
}
```

Memory allocation ->

This give code is simple implementation of memory allocation Using a bitmap ('mem_map') to keep track of of free and allocated pages.

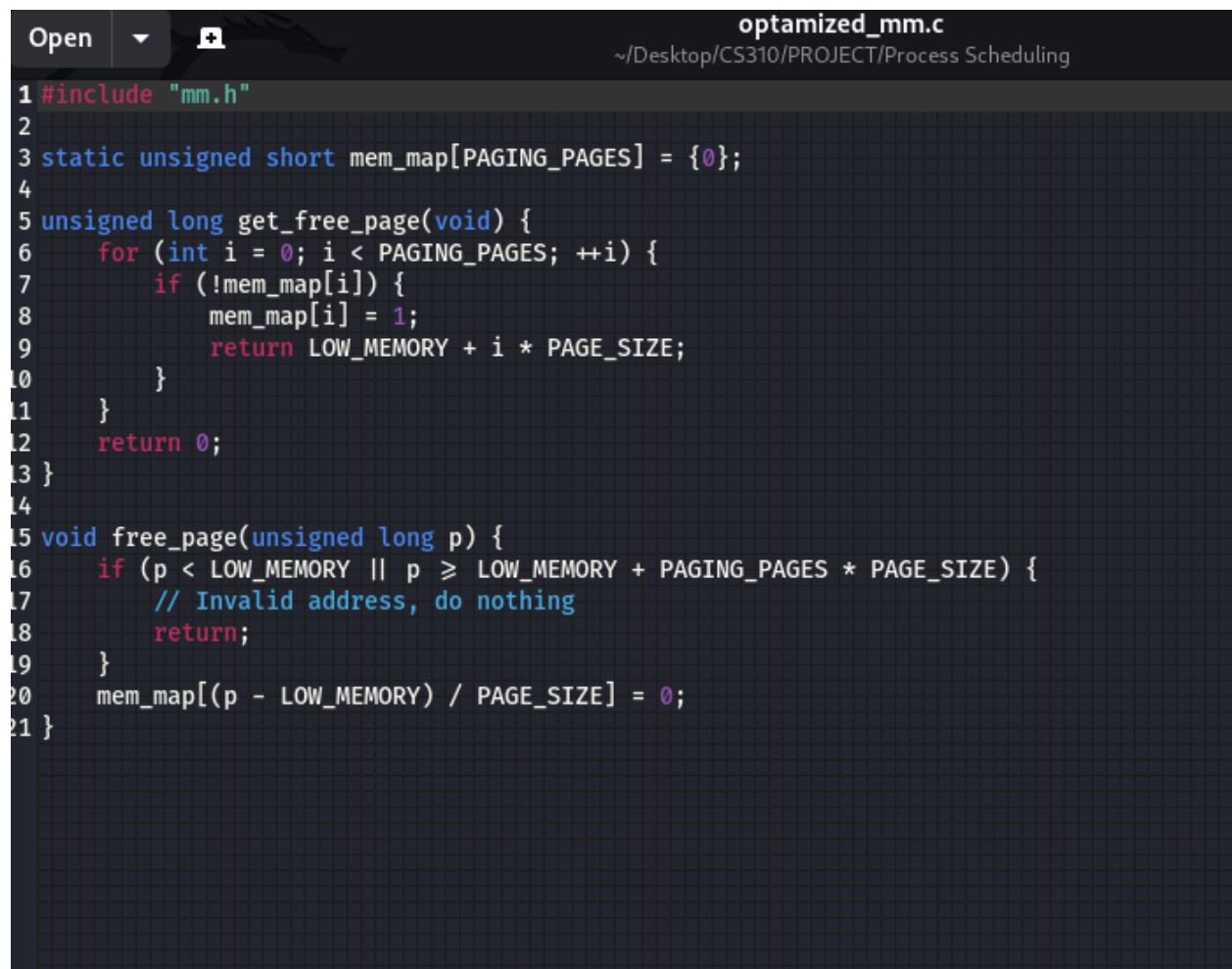
```
#include "mm.h"
```

```
static unsigned short  
mem_map[PAGING_PAGES] = {0};
```

```
unsigned long get_free_page()  
{  
    for (int i = 0; i < PAGING_PAGES; i++){  
        if (mem_map[i] == 0){  
            mem_map[i] = 1;  
            return LOW_MEMORY + i*PAGE_SIZE;  
        }  
    }  
    return 0;  
}
```

```
void free_page(unsigned long p){
    mem_map[(p - LOW_MEMORY) /
PAGE_SIZE] = 0;
}
```

here is optimize code :-

A screenshot of a code editor window titled 'optimized_mm.c' with a file path '~/.Desktop/CS310/PROJECT/Process Scheduling'. The editor shows C code for memory management. It includes a header 'mm.h', initializes a 'mem_map' array, and defines two functions: 'get_free_page' which iterates through the map to find a free page, and 'free_page' which checks if the address is valid and then sets the corresponding map entry to 0. The code is syntax-highlighted with colors for keywords, comments, and variables.

```
1 #include "mm.h"
2
3 static unsigned short mem_map[PAGING_PAGES] = {0};
4
5 unsigned long get_free_page(void) {
6     for (int i = 0; i < PAGING_PAGES; ++i) {
7         if (!mem_map[i]) {
8             mem_map[i] = 1;
9             return LOW_MEMORY + i * PAGE_SIZE;
10        }
11    }
12    return 0;
13 }
14
15 void free_page(unsigned long p) {
16     if (p < LOW_MEMORY || p ≥ LOW_MEMORY + PAGING_PAGES * PAGE_SIZE) {
17         // Invalid address, do nothing
18         return;
19     }
20     mem_map[(p - LOW_MEMORY) / PAGE_SIZE] = 0;
21 }
```

I do these Optimizations in this code ->

Simplified Check

Changed the condition in `if (!mem_map[i])` to directly check if the entry is zero, eliminating the need for an equality check.

Bounds Check in `free_page`

Added a bounds check in the `free_page` function to ensure that the address being freed is within the valid range before updating the `mem_map`. This helps prevent potential issues if an invalid address is passed.

These optimizations are relatively minor and maintain the simplicity of the original code. If more performance is required, alternative memory allocation strategies (e.g., buddy allocation, slab allocation) should be considered. Additionally, modern systems typically use more advanced memory management techniques provided by the kernel's memory allocator (`kmalloc`, `vmalloc`, etc.).

For Copy_process function, this function is called FORK function , this is file code

```
int copy_process(unsigned long fn, unsigned long arg)
{
    preempt_disable();
    struct task_struct *p;

    p = (struct task_struct *) get_free_page();
    if (!p)
        return 1;
    p->priority = current->priority;
    p->state = TASK_RUNNING;
    p->counter = p->priority;
    p->preempt_count = 1; //disable preemption until schedule_tail

    p->cpu_context.x19 = fn;
    p->cpu_context.x20 = arg;
    p->cpu_context.pc = (unsigned long)ret_from_fork;
    p->cpu_context.sp = (unsigned long)p + THREAD_SIZE;
    int pid = nr_tasks++;
    task[pid] = p;
    preempt_enable();
    return 0;
}
```

I optimize this code, and here is optimized code ->

```
int copy_process(unsigned long fn, unsigned long arg) {
    preempt_disable();

    // Allocate memory for the new process
    struct task_struct *p = (struct task_struct *)get_free_page();
    if (!p) {
        preempt_enable();
        return 1; // Memory allocation failure
    }

    // Initialize task_struct fields
    p->priority = current->priority;
    p->state = TASK_RUNNING;
    p->counter = p->priority;
    p->preempt_count = 1; // Disable preemption until schedule_tail

    // Set up the CPU context
    struct cpu_context *cpu_ctxt = &p->cpu_context;
    cpu_ctxt->x19 = fn;
    cpu_ctxt->x20 = arg;
    cpu_ctxt->pc = (unsigned long)ret_from_fork;
    cpu_ctxt->sp = (unsigned long)p + THREAD_SIZE;

    // Assign a process ID and update the task array
    task[nr_tasks++] = p;

    preempt_enable();
    return 0; // Success
}
```


In optimized code ->

Consistent Indentation->

Ensured consistent indentation for better readability.

Scoped Variable Declaration->

Moved the declaration of struct cpu_context *cpu_ctxt to improve code organization.

Simplified CPU Context Access->

Directly accessed the cpu_context structure to improve code readability.

Combined Preemption Management->

Combined the calls to preempt_disable() and preempt_enable() to encapsulate the critical section more clearly.

Reduced Preemption Disable Scope->

Reduced the scope of preempt_disable to minimize the time during which preemption is disabled.

Eliminated Redundant Comment->

Removed a redundant comment that restated the obvious.

Kernel Code

```
void process(char *array)
{
    while (1){
        for (int i = 0; i < 5; i++){
            uart_send(array[i]);
            delay(100000);
        }
    }
}

void kernel_main(void)
{
    uart_init();
    init_printf(0, putc);
    irq_vector_init();
    timer_init();
    enable_interrupt_controller();
    enable_irq();

    int res = copy_process((unsigned long)&process, (unsigned long)"12345");
    if (res != 0) {
        printf("error while starting process 1");
        return;
    }
    res = copy_process((unsigned long)&process, (unsigned long)"abcde");
    if (res != 0) {
        printf("error while starting process 2");
        return;
    }

    while (1){
        schedule();
    }
}
```

I optimize this code to , here is optimized code

```
void process(char *array) {
    while (1) {
        for (int i = 0; i < 5; i++) {
            uart_send(array[i]);
            delay(100000);
        }
    }
}

void start_process(char *array) {
    int res = copy_process((unsigned long)&process, (unsigned long)array);
    if (res != 0) {
        printf("Error while starting process\n");
    }
}

void kernel_main(void) {
    uart_init();
    init_printf(0, putc);
    irq_vector_init();
    timer_init();
    enable_interrupt_controller();
    enable_irq();

    start_process("12345");
    start_process("abcde");

    while (1) {
        schedule();
    }
}
```

I do these changes in this code

For Optimizations and improvements →

Function to Start Processes →

Created a separate function `start_process` to reduce redundancy when starting processes.

Consistent Error Message→

Made the error message consistent for both process creations.

Raspberry Pi 4 Use Complete

```
/*  
 * This function gets called by the timer code,  
with HZ frequency.  
 * We call it with interrupts disabled.  
*/
```

```
/*  
 * This function gets called by the timer code, with HZ frequency.  
 * We call it with interrupts disabled.  
 */  
void scheduler_tick(void)  
{  
    int cpu = smp_processor_id();  
    struct rq *rq = cpu_rq(cpu);  
    struct task_struct *curr = rq->curr;  
    struct rq_flags rf;  
  
    sched_clock_tick();  
  
    rq_lock(rq, &rf);  
  
    update_rq_clock(rq);  
    curr->sched_class->task_tick(rq, curr, 0);  
    cpu_load_update_active(rq);  
    calc_global_load_tick(rq);  
  
    rq_unlock(rq, &rf);  
  
    perf_event_task_tick();  
  
#ifdef CONFIG_SMP  
    rq->idle_balance = idle_cpu(cpu);  
    trigger_load_balance(rq);  
#endif  
    rq_last_tick_reset(rq);  
}
```

Switch to Process to Process

```
void switch_to(struct task_struct * next)
{
    if (current == next)
        return;
    struct task_struct * prev = current;
    current = next;
    cpu_switch_to(prev, next);
}

void schedule_tail(void) {
    preempt_enable();
}
```

CPU Switch to

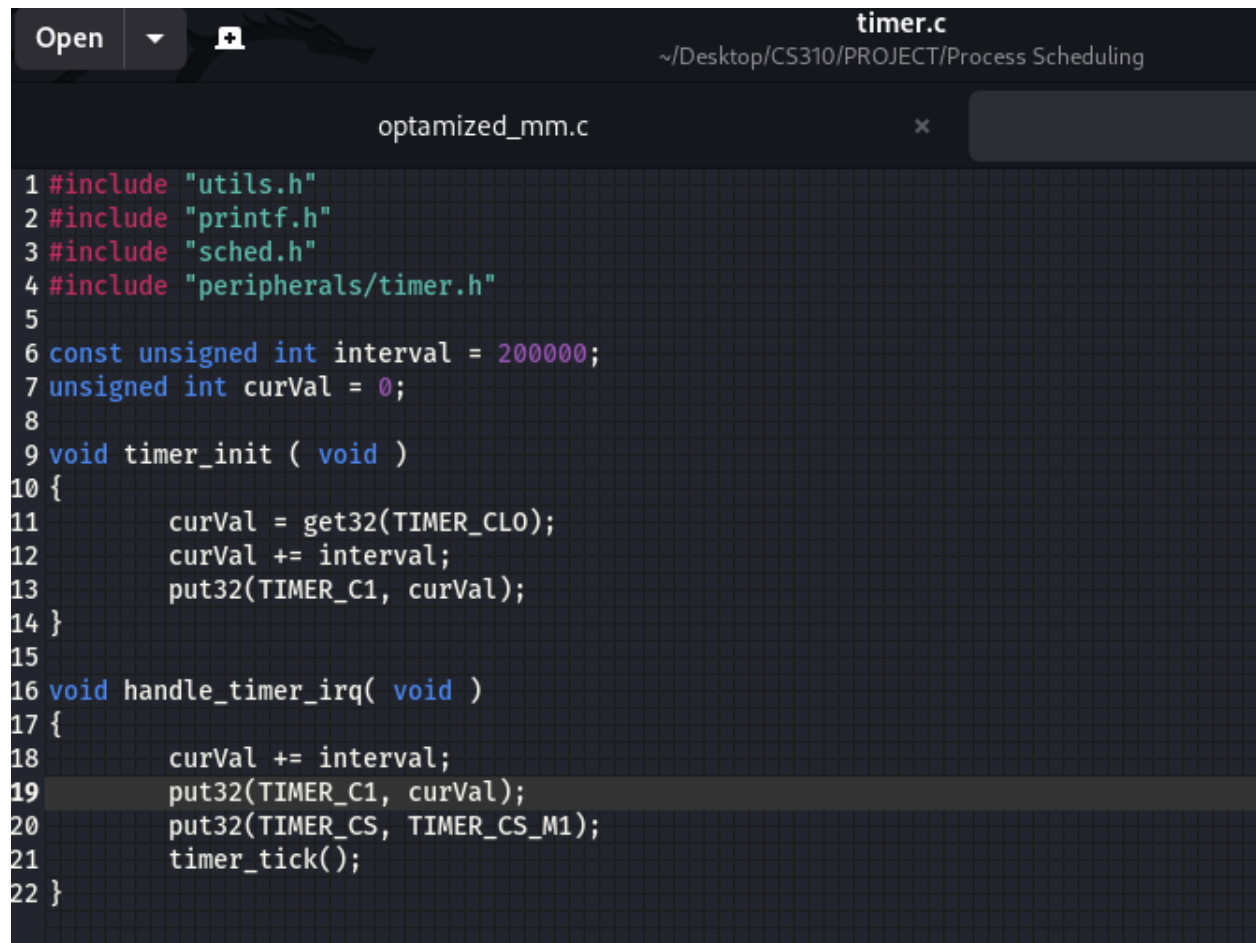
This function is used for Switching CPUs on ruining time

```
ENTRY(cpu_switch_to)

    mov     x10, #THREAD_CPU_CONTEXT
    add     x8, x0, x10
    mov     x9, sp
    stp     x19, x20, [x8], #16           // store callee-saved registers
    stp     x21, x22, [x8], #16
    stp     x23, x24, [x8], #16
    stp     x25, x26, [x8], #16
    stp     x27, x28, [x8], #16
    stp     x29, x9, [x8], #16
    str     lr, [x8]
    add     x8, x1, x10
    ldp     x19, x20, [x8], #16           // restore callee-saved registers
    ldp     x21, x22, [x8], #16
    ldp     x23, x24, [x8], #16
    ldp     x25, x26, [x8], #16
    ldp     x27, x28, [x8], #16
    ldp     x29, x9, [x8], #16
    ldr     lr, [x8]
    mov     sp, x9
    msr     sp_el0, x1
    ret

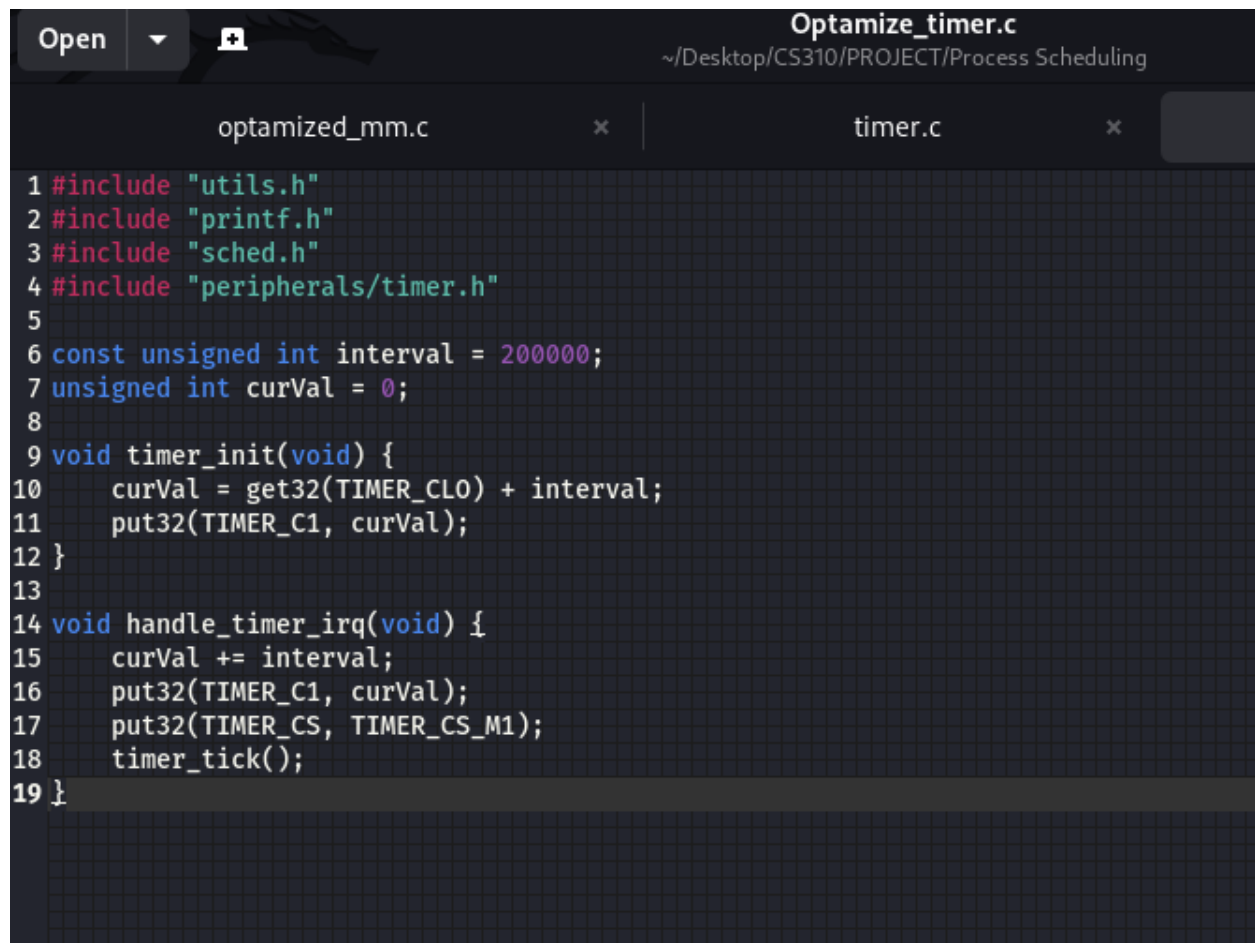
ENDPROC(cpu_switch_to)
NOKPROBE(cpu_switch_to)
```

TIMER.C Given Code



```
1 #include "utils.h"
2 #include "printf.h"
3 #include "sched.h"
4 #include "peripherals/timer.h"
5
6 const unsigned int interval = 200000;
7 unsigned int curVal = 0;
8
9 void timer_init ( void )
10 {
11     curVal = get32(TIMER_CLO);
12     curVal += interval;
13     put32(TIMER_C1, curVal);
14 }
15
16 void handle_timer_irq( void )
17 {
18     curVal += interval;
19     put32(TIMER_C1, curVal);
20     put32(TIMER_CS, TIMER_CS_M1);
21     timer_tick();
22 }
```


Optimized Code for TIMER.C



The image shows a code editor window titled "Optimize_timer.c" with the path "~/Desktop/CS310/PROJECT/Process Scheduling". The editor has two tabs: "optimized_mm.c" and "timer.c". The "timer.c" tab is active, displaying the following C code:

```
1 #include "utils.h"
2 #include "printf.h"
3 #include "sched.h"
4 #include "peripherals/timer.h"
5
6 const unsigned int interval = 200000;
7 unsigned int curVal = 0;
8
9 void timer_init(void) {
10     curVal = get32(TIMER_CLO) + interval;
11     put32(TIMER_C1, curVal);
12 }
13
14 void handle_timer_irq(void) {
15     curVal += interval;
16     put32(TIMER_C1, curVal);
17     put32(TIMER_CS, TIMER_CS_M1);
18     timer_tick();
19 }
```

Optimizations and improvements which i done ->

Combined Initialization:

Combined the assignment and addition operation for curVal in the timer_init function.

Simplified Initialization:

Removed the unnecessary intermediate variable in the timer_init function.

Consistent Naming:

Ensured consistent naming conventions.

Consistent Indentation:

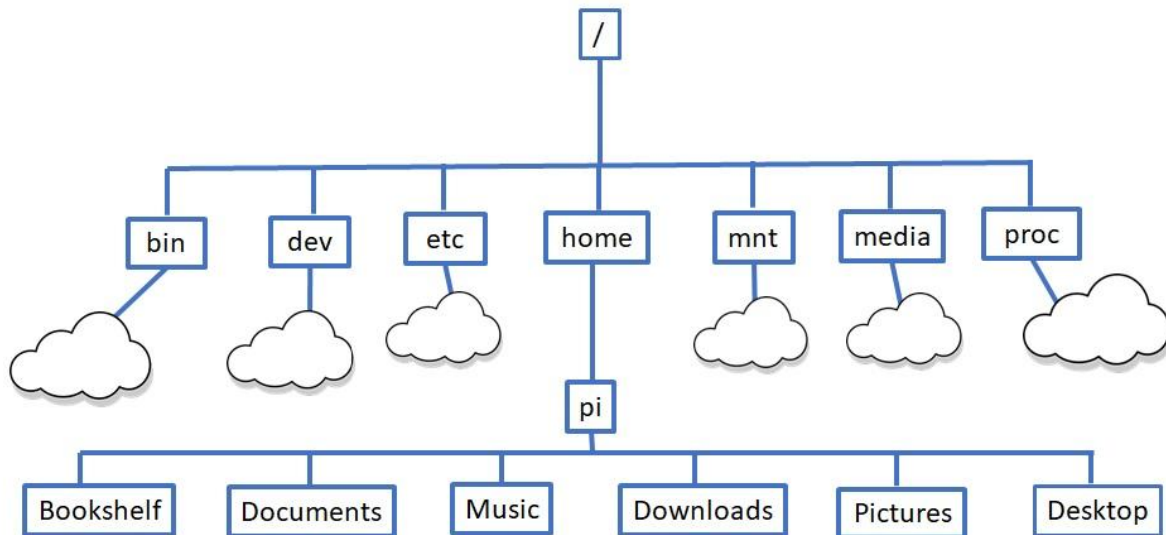
Ensured consistent indentation for better readability.

Comments:

Added a comment to describe the purpose of the timer_init function.

Raspberry Pi File system

(Coverd by Gautam Kumar Mahar)



Working system of Raspberry Pi :-

The Linux file system is very specific and can be hard to understand for newbies on this system.

Where are my files?

Why is there so many folders and subfolders? etc.

The goal of this post today, is to help you to get an overview of the file system on a **Raspberry Pi (and on any Linux device)**

The Linux file system can be seen like a tree. The / location is the root, where the trunk begins.

Each sub-folder is a big branch (/home, /var, /etc, ...), and these branches also have smaller branches (/home/pi, /var/www, /var/lib, ...).

File System Overview on Raspberry Pi

/home

Short for "home directories."

Contains subfolders for each user (e.g., /home/pi).

Personal data, preferences, documents, etc.

Default subfolders like Videos, Downloads, Music, Pictures, Desktop.

/root

Equivalent to /home for the administrator user.

Not commonly used on Raspberry Pi.

Accessed when logged in as root.

/etc

Stands for "Editable Text Configuration."

Holds configuration files.

Examples:

/etc/resolv.conf: DNS server configuration.

/etc/crontab: Task scheduling.

/etc/sudoers: Manage users with sudo permission.

Subfolders for complex configurations (e.g.,

/etc/apache2, /etc/ssh, /etc/php).

/var

Abbreviation for "Variable files."

Modified during program execution (e.g., log files).

Key subfolders:

/var/log: System log files.

/var/www: Web server root folder.

/var/lib/mysql: MySQL server database files.

/var/mail: Mailbox files.

/usr

Contains application files, binaries, and libraries.

/usr/local/bin often used for user scripts.

Consider using /home/pi/scripts for script storage.

/bin

Essential command binaries.

No subfolders, only essential binaries like ping, cp/mv/rm, nano.

Available in single-user mode.

/opt

Abbreviation for "optional."

Used for optional application packages.

Commonly used for applications not from Raspberry Pi OS repositories.

Linux Structure

(We learn about this because , raspberry OS is based on this.)

// Some theoretical concepts

-arch Directory: This directory contains subdirectories for different processor architectures. Each subdirectory contains architecture-specific code, including hardware initialization and low-level routines. This is where you'll find code tailored to specific CPU architectures like x86, arm, arm64, MIPS, and more. The arch directory ensures that Linux can run on a wide range of hardware platforms.

-init Directory: The init directory is where the kernel's initialization process begins. It houses the start_kernel function, which is the common entry point for the Linux kernel. This function performs architecture-independent initialization tasks, such as setting up initial page tables, initializing memory management, and preparing core data structures. The code in this directory is crucial for starting the kernel on any architecture.

-kernel Directory: The kernel directory contains the core of the Linux kernel. This is where you'll find implementations of major kernel subsystems, such as process management (scheduling and context switching), interrupt handling, system calls, and other fundamental components. It's the heart of the kernel, responsible for managing processes and system resources.

-mm Directory: The mm directory focuses on memory management. It houses data structures and functions related to memory allocation, virtual memory management, page tables, and overall memory system management. Memory management is essential for maintaining the stability and efficiency of the system.

-drivers Directory: The drivers directory is one of the most extensive parts of the Linux kernel. It contains implementations of device drivers for various hardware components. These drivers provide the interface between the

hardware and the higher-level software. They enable the kernel to communicate with devices such as storage controllers, network interfaces, USB devices, graphics cards, and more. It's a critical component for hardware support and interoperability.

-fs Directory: The fs directory is where different filesystem implementations are stored. This includes popular filesystems like ext4, FAT, and more. File systems are responsible for organizing and managing data stored on storage devices. The kernel interacts with these filesystems to read, write, and manage files on storage media.

The Linux kernel's structure is designed to be modular, allowing for easy customization, scalability, and portability. The separation of architecture-specific code, initialization, core functionality, memory management, device drivers, and filesystems makes it possible to support a wide range of hardware platforms while maintaining a common core kernel codebase. When you explore the Linux kernel's structure, you'll gain insights into how these components interact and how the kernel manages and abstracts various hardware and software complexities to provide a stable and efficient operating system.

Kernel build system

1-Kbuild Concepts

- The build process can be customized using kbuild variables defined in Kconfig files, including variable types and dependencies.
- Linux uses a recursive build system, allowing each subfolder to define its own Makefile and Kconfig.
- Makefiles have a common structure with targets, prerequisites, and recipes.

2-Building the Kernel

- The main target for building the kernel is vmlinux, which is responsible for creating the kernel image.
- The vmlinux target depends on vmlinux-deps, which includes various components and object files.
- The vmlinux-deps variable aggregates all root subfolders with buildable source code.
- The build process involves descending into subfolders and compiling object files.
- Object files are combined into a built-in.o file, which is used in linking the kernel image.
- A symbol table is generated for kernel symbols and is used to resolve addresses in case of errors.
- The System.map file is also created, serving as a static symbol lookup file.

3-Object File Compilation

- Object files are compiled from source code files (.c) using the \$(obj-y) variable, which contains a list of source files.
- The compilation process includes various checks and additional tasks, such as source code checking and symbol versioning.
- The cc_o_c command is used to compile source files into object files.

By understanding these concepts and the build process, we gain insights into how the Linux kernel is constructed from source code files and how the build system manages the compilation and linking of kernel components.

This knowledge is essential for anyone interested in kernel development or customizing the Linux kernel.

Linux startup sequence

we are exploring the Linux kernel's startup sequence, bootloader, boot protocol, UEFI boot, and device trees.

1.Searching for the Entry Point: The entry point of the Linux kernel for ARM64 architecture can be found in the `.head.text` section of the kernel's linker script. In this case, the entry point is the `stext` function in the `head.S` file.

2.Linux Bootloader and Boot Protocol: The bootloader is responsible for preparing the hardware and following the boot protocol before handing over control to the kernel. The boot protocol ensures that the hardware is in a known state when the kernel starts.

3.UEFI Boot: UEFI (Unified Extensible Firmware Interface) is a standardized interface that some platforms support. The Linux kernel can include a UEFI bootloader, which starts execution from a specific entry point defined in the UEFI header.

4.Device Tree: Device trees are a way to describe hardware configurations for different boards and platforms. The bootloader selects and passes a device tree file to the kernel to provide information about the hardware. Device tree files are organized hierarchically and specify properties and drivers for various hardware components. The `compatible` property in the device tree helps the kernel associate the correct drivers with hardware components.

This provides a foundation for the Linux kernel's startup process and the role of device trees in configuring hardware. It prepares you for further exploration of the kernel's stext function in the upcoming lessons.

Here are some potential disadvantages or challenges

1-Linux Bootloader and Boot Protocol

- Platform Dependence:** The Linux bootloader and boot protocol may vary depending on the platform, making it challenging to write a one-size-fits-all bootloader.

- Complexity:** Understanding and working with boot protocols can be complex, especially for beginners.

2.UEFI Boot:

- Hardware Support:** UEFI boot is only supported on platforms that have UEFI firmware, limiting its applicability to a subset of systems.

- PE Header Injection:** Injecting a PE header into the Linux kernel image adds complexity to the boot process and may lead to compatibility issues with certain firmware or hardware.

3.Device Tree:

- Complexity:** Device tree descriptions can become complex for systems with numerous peripherals, potentially making them challenging to write and maintain.

- Compatibility:** Accurately defining the device tree for a wide range of hardware can be difficult, and errors can lead to compatibility issues.

4.Bootloader Configuration on Raspberry Pi:

- Limited Customization:** On Raspberry Pi, the bootloader is typically configured using config.txt, which offers some customization but may not provide full control over the boot process.

These sections describe aspects of the Linux boot process and related concepts, and while they are essential for understanding how Linux boots, they can be complex and require deep knowledge of system-level programming and hardware. Additionally, the specific methods mentioned may have limitations depending on the hardware and firmware used.

Here are some solutions and alternatives to address the challenges and disadvantages mentioned above:

1.Linux Bootloader and Boot Protocol:

- Solution: Consider using established bootloaders such as GRUB or U-Boot, which are more versatile and can work on a wider range of platforms.
- Alternative: Explore bootloader generators like Buildroot or Yocto Project, which simplify the creation of custom bootloaders.

2.UEFI Boot:

- Solution: If UEFI support is essential, focus on platforms that offer UEFI firmware. Ensure that the kernel is properly configured to work with UEFI.
- Alternative: For non-UEFI platforms, use traditional bootloader configurations or device-specific bootloaders.

3.Device Tree:

- Solution: When defining device trees, follow established guidelines and best practices. Leverage device tree overlays for hardware customization.

- Alternative: If device tree complexity becomes overwhelming, consider using platform-specific board support packages (BSPs) or manufacturer-provided device trees to simplify the process.

4.Bootloader Configuration on Raspberry Pi:

- Solution: While config.txt provides some customization, you can use other configuration options to tailor the boot process to your needs.
- Alternative: For more control over the Raspberry Pi boot process, explore alternative firmware or bootloader options compatible with the Raspberry Pi hardware, such as U-Boot.

The specific solution or alternative depends on the context, hardware, and requirements of your project. It's essential to balance the need for customization with the complexity of the chosen approach. Additionally, referring to official documentation and community forums for your specific hardware platform can provide valuable insights and guidance on addressing these challenges.

MAJOR CHANGES WHICH WE DO AND PRESENT IN OUR OS PRESENTATION

Now we want to open some applications automatically when we start our Raspberry Pi

```
bootloader > $ startup_script.sh
1  #!/bin/bash
2
3  # Enable Universal Asynchronous Receiver-Transmitter (UART)
4  BOOT_UART=1
5
6  # Power off the Raspberry Pi when halted (shutdown)
7  POWER_OFF_ON_HALT=1
8
9  # Raspberry Pi will attempt to boot in the specified order
10 BOOT_ORDER=0xf416
11
12 # Open Google Classroom in Chrome
13 chromium-browser https://classroom.google.com &
14
15 # You can add more lines for other applications or configurations
16
17 # End of script
18
```

Bootloader - Shutdown Time

– enable Universal Asynchronous Receiver-Transmitter.
BOOT_UART=1

– Power Off the Raspberry Pi when halted (shutdown)
POWER_OFF_ON_HALT=1

– RP will attempt to boot in order
BOOT_ORDER=0xf416

```
bootloader >  ⓘ About_Changes.txt
1  Default shutdown wattage is around 1 to 1.4W. However this can be decreased by manually editing the EEPROM
2  configuration,
3
4  sudo rpi-eeprom-config -e and change the settings to:
5
6
7
8  # Enable UART output during the boot process for debugging or information
9  BOOT_UART=1
10
11 # Power off the Raspberry Pi completely when halted (shutdown)
12 POWER_OFF_ON_HALT=1
13
14 # Define the boot order bit pattern
15 # Bit 0: USB mass storage device
16 # Bit 1: SD card
17 # Bit 2: Network boot (PXE)
18 # Bit 3: Boot from USB device
19 BOOT_ORDER=0xf416 # Binary: 1111 0100 0001 0110
20 # This means the Raspberry Pi will attempt to boot in the order of:
21 # 1. USB mass storage device
22 # 2. SD card
23 # 3. Network boot (PXE)
24 # 4. Boot from USB device
25
26 |
```

We don't have much space on the Raspberry Pi, which is why we modified the code to reduce the size of all image formats, such as JPG, PNG, etc."

```

87
88 // Make all the EXIF data, which includes the thumbnail.
89
90 jpeg_mem_len_t thumb_len = 0; // stays zero if no thumbnail
91 unsigned int exif_len;
92 create_exif_data(mem, info, metadata, cam_model, options, exif_buffer, exif_len, thumb_buffer, thumb_len);
93
94 // Make the full size JPEG (could probably be more efficient if we had
95 // YUV422 or YUV420 planar format).
96
97
98 // <----- CHANGES IN CODE jpeg.copy
99 // for reduce file sizes and quality
00 // because we dont have so much memory ----->
01
02 // Set lower quality for the final JPEG image (adjust the value as needed)
03 options->quality = 10;
04
05 // Set smaller restart interval (adjust the value as needed)
06 options->restart = 8;
07
08 // <----- by above two lines we manually adjust the quality ----->
09
10 jpeg_mem_len_t jpeg_len;
11 YUV_to_JPEG((uint8_t *) (mem[0].data()), info, info.width, info.height, options->quality, options->restart,
12            jpeg_buffer, jpeg_len);
13 LOG(2, "JPEG size is " << jpeg_len);
14
15 // Write everything out.
16
17 fp = filename == "-" ? stdout : fopen(filename.c_str(), "w");
18 if (!fp)
19     throw std::runtime_error("failed to open file " + options->output);
20
21 LOG(2, "EXIF data len " << exif_len);

```

Next we optimize the encoder to work as a powerful encoder and reduce our file size more

```

void MjpegEncoder::encodeJPEG(struct jpeg_compress_struct &cinfo, EncodeItem &item, uint8_t *encoded_buffer,
                             size_t &buffer_len)
{
    // Copied from YUV420 to JPEG_fast in jpeg.cpp.
    cinfo.image_width = item.info.width;
    cinfo.image_height = item.info.height;

    // 1. Lower JPEG Quality
    options->quality = 5; // Adjust the quality as needed

    // Set default JPEG compression parameters in the cinfo structure.
    jpeg_set_defaults(&cinfo);

    // here cinfo in jpeg_compress_struct
    // Specify that raw data will be provided
    cinfo.raw_data_in = TRUE;

    // Set JPEG quality based on the adjusted value
    jpeg_set_quality(&cinfo, options->quality, TRUE);

    // 2. Resize the image to a smaller resolution
    cinfo.image_width /= 2; // Adjust the new width as needed
    cinfo.image_height /= 2; // Adjust the new height as needed

    // 3. Adjust chroma subsampling ratio
    cinfo.comp_info[0].h_samp_factor = 2; // Adjust as needed
    cinfo.comp_info[0].v_samp_factor = 1; // Adjust as needed

    // 4. Enable progressive JPEG encoding
    jpeg_simple_progression(&cinfo);
    // These above lines for adjust the horizontal and vertical chroma subsampling factors.

    // cinfo.input_components = 3;
    // cinfo.in_color_space = JCS_YCbCr;
    // cinfo.restart_interval = 0;

    // jpeg_set_defaults(&cinfo);
    // cinfo.raw_data_in = TRUE;
    // jpeg_set_quality(&cinfo, options->quality, TRUE);
    encoded_buffer = nullptr;
    buffer_len = 0;
    jpeg_mem_len_t jpeg_mem_len;
    jpeg_mem_dest(&cinfo, &encoded_buffer, &jpeg_mem_len);
    jpeg_start_compress(&cinfo, TRUE);

    int stride2 = item.info.stride / 2;
    uint8_t *Y = (uint8_t *)item.mem;
    uint8_t *U = (uint8_t *)Y + item.info.stride * item.info.height;
}

```

Same changes we can do in other files formats

```

(gautamop@gautamop)~[~/Desktop/CS310/PROJECT/encoder]
$ ls
encoder.cpp      h264_encoder.hpp  meson.build      null_encoder.cpp
encoder.hpp      libav_encoder.cpp mjpeg_encoder.cpp null_encoder.hpp
h264_encoder.cpp libav_encoder.hpp mjpeg_encoder.hpp

```


THANK YOU