哈爾濱工業大學 (Shenzhen)
HARBIN INSTITUTE OF TECHNOLOGY

Written in Rust based on RISC-V64

Multi-core operating system UltraOS

Project member: Li Chenghao (team leader)

Gong Haochen

Ren Xiangyu

Instructors: Xia Wen, Jiang Zhongming

Table of contents

1 Overview

UltraOS is a RISC-V64-based multi-core operating system written in Rust, and can run on bare Machine K210 platform (including 400MHZ dual-core RISC-V processor) and qemu simulator.

1.1 Project Background and Significance

From the perspective of computer system teaching in Harbin Institute of Technology (Shenzhen), whether it is in theoretical teaching Still in the actual experimental operation, it seems relatively weak. As one of the three King Kong of the computer profession, the operator The operating system is the core of understanding almost program operation and hardware resource management. From the teaching point of view, it is precisely because Due to the complexity of its management mechanism, the operating system has become a practical and theoretical "hurdle".

From a horizontal comparison, the students of Huazhong University of Science and Technology have created their own team "Windless Wind". Focus on research in the direction of computer operating systems, including hardware abstraction, device drivers, and new operating system implementations. In-depth exploration projects such as discovery and research. Looking at Tsinghua University again, there are three mountains of rCore, uCore and zCore, The language, the cornerstone of the operating system, and the microkernel have been explored respectively, and they have also been successfully transformed into experimental guidelines. tutorials and concrete teaching-friendly code. The undergraduate operating system education of these two schools is not only in the basic operation The system experiment teaching has its own unique mode, and has carried out in-depth exploration.

At the same time, I also noticed that the undergraduates of the school are still continuing to explore in the direction they are interested in, and And formed a certain research group. To sum up, we think that the Computer Department of Harbin Institute of Technology (Shenzhen) The key to traditional talents lies in three points: rich theoretical support (for courses), own experimental design (for course practice) experience), cutting-edge exploration (there are instructors in the corresponding field for in-depth digging).

We focus on solving the second point: own experimental design. So we based ourselves on developing the operating system on the real machine, To support the design of free course experiments, rather than being constrained by others. In this way, in the course task design, there can be With greater degrees of freedom, there can also be greater freedom and space for further deepening understanding.

At the same time, we also hope to be able to inherit in the form of Github project team, course experiment, computer system group, etc. As well as iteration, this is necessary for the cultivation of talents.

1.2 Overview of research at home and abroad

For industry or academia, the operating system will be divided into two aspects from the broadest point of view: kernel and microkernel. The design method adopted by UltraOS is a macro kernel, but in fact, due to its simplicity

The design concept, as well as the simplicity of its functions, can be transformed into a microkernel under the framework. The following introduces the microkernel And the domestic and foreign research status of the macro kernel.

For the microkernel, the most famous is the L4 stage, which is the most cutting-edge stage. Let's discuss L4 first previous stage.

The microkernel was expected from the beginning for its excellent development design and robustness, but in a hardware Can not be outstanding era, and not able to achieve a good influence. The essence of a microkernel is to combine most of its internal Kernel communication, placed in user mode, to put as many kernel components as possible in user programs. Thus, the kernel program It can be restarted by the microkernel when a failure occurs, and inter-process communication technology is used when communication is needed. This This looks very good, but it faces two major problems: frequent inter-process communication greatly reduces performance, and users Programs may introduce new security issues.

Before the microkernel L4 version, this has always been a big problem. L4 adopts a new architecture, the process Inter-communication, that is, IPC, has increased by 10 to 20 times, and a new era has been opened since then. Not only that, in the performance of L4 During the change process, some experts used logical reasoning to prove that the operating system is completely safe, and in the macro kernel, This is an incredible method of proof. Since then, the microkernel has made considerable breakthroughs in terms of security and performance. Officially challenged the macro kernel.

In the L4 series of systems, different microkernel versions corresponding to different application ranges are produced, and embedded style, some are born for security. But one of the most influential is still seL4, which has also undergone an evolution Deductive reasoning security verification, and has a very prominent performance advantage over similar microkernels. Meanwhile, Google The Fuchsia operating system based on the Zircon microkernel was recently launched, which has also achieved greater influence in the industry.

For the macro kernel, the most famous is the open source operating system Linux. Linux is written in C language and has non- Very powerful functions and performance, loved by the majority of programmers. The problem is with potential security and robustness issues, Due to the large number of kernel codes, it is difficult to detect them systematically, and the only way to find them is to eliminate them. It is impossible to avoid all security issues from the design, and the system will occasionally crash.

In fact, most operating systems used in industry are not simple macrokernels and microkernels, but A hybrid core of the two. Compared with the microkernel, more important kernel programs are added to the kernel to Guaranteed performance and security. But compared to the macro kernel, some kernel-level programs are placed in the user mode. Representatives of the hybrid kernel are the MacOS operating system and the Windows system based on the XNU hybrid kernel. The former is composed of Due to its structural advantages, it is rarely attacked by computer viruses.

1.3 Main work of the project

UltraOS is committed to developing a RISC-V64-based multi-core operating system that can run on physical On the bare metal K210 platform (including 400MHZ dual-core RISC-V processor). It should be noted here that UltraOS The ultimate goal is teaching-oriented, that is, to provide a precedent for exploration, so that the future operating system talents Can contribute to the cultivation. This means that we need to both work towards the industry and remove some of the Branches that are applied in engineering but are not conducive to understanding the core concepts. At the same time, during the adaptation process of the game, I We should also choose programs that are as general as possible. In short, neither performance nor function is the ultimate goal, he They may become intermediate targets in our development process, but the end point of UltraOS still gives everyone a control Reference to the process of operating system development and architecture design.

In order to avoid the anti-engineering development logic of "making wheels", we used the rCoreTutorial-v3 code (2021.03.26 version) framework, and develop on it, so that we can Start quickly and gradually improve the operating system functions from top to bottom, from user program support to hardware abstraction supremacy The following improvements, including new functions, enhanced robustness, improved performance, enhanced compatibility, etc., are finally realized Development of UltraOS.

Usually, discussions of operating systems focus on the design of the kernel, but formal operating systems require More and more complete support. The two most important parts are: hardware abstraction interface SBI and user abstraction Abstract interface ABI. The former is used to realize the independence of the hardware platform, including interrupt proxy, soft processing of invalid instructions and other operations, so that the operating system kernel does not need to care too much about the differences in hardware platforms, and at the same time it can The software level extends the lack of functions at the hardware level and prolongs the life cycle of the hardware platform. The latter makes the user program There is no need to care too much about the specific differences of the operating system, so that its programs can run on different operating systems.

For this goal, we need to carry out specific design and implementation. For the hardware platform of K210, I We used the RustSBI project developed by Luo Jia of Huazhong University of Science and Technology to realize the abstraction of hardware. at the same time We have also modified it to better support the operating system functions from the bottom layer. And for using For user support, we have built standard libraries for Rust and C, enabling users to write application program, and does not need to pay too much attention to the interface of the operating system.

Now turn your attention to the kernel. The kernel architecture mainly includes three parts: process management, memory management, file file system. Process management mainly solves the creation of the initial process, process maintenance, creation, scheduling and exception handling. Memory management mainly addresses page table management, process memory allocation and management, and memory layout initialization. File system It supports EXT2-like and FAT32 file systems, and mainly solves the problems of hard disk drive access and file system management.

At the same time, it should be noted that all parts must support the consistency and security of concurrent access, that is,

Supports multi-core operation.

2 Demand Analysis

UltraOS is committed to realizing a streamlined multi-core operating system, which makes it have good scalability, and at the same time

It can also support important functions. Based on this goal, UltraOS focuses on functions rather than

performance.

In terms of functions, all functions of UltraOS must support the dual-core operation of the K210 real machine. At the same time, in order to

Able to participate in the first National Undergraduate Computer System Ability Cultivation Competition, the kernel implementation group in the operating system track, and at the same time

Taking into account the system call implementation of the basic architecture, we finally concluded that the system calls that should be implemented are as follows:

```
const SYSCALL_GETCWD: usize =
17; const SYSCALL_DUP: usize = 23;
const SYSCALL_DUP3:usize = 24;
const SYSCALL_FCNTL:usize = 25;
const SYSCALL_IOCTL:usize = 29;
const SYSCALL_MKDIRAT: usize = 34;
const SYSCALL_UNLINKAT: usize = 35;
const SYSCALL_LINKAT: usize = 37;
const SYSCALL_UMOUNT2: usize = 39;
const SYSCALL_MOUNT: usize = 40;
const SYSCALL_FACCESSAT: usize =
48; const SYSCALL_CHDIR: usize = 49;
const SYSCALL_OPENAT: usize = 56;
const SYSCALL_CLOSE: usize = 57;
const SYSCALL_PIPE: usize = 59; const
SYSCALL_GETDENTS64: usize = 61; const
SYSCALL_LSEEK: usize = 62; const
SYSCALL_READ: usize = 63; const
SYSCALL_WRITE: usize = 64; const
SYSCALL_WRITEV: usize = 66; const
SYSCALL_SENDFILE: usize = 71; const
SYSCALL_PSELECT6: usize = 72; const
SYSCALL_READLINKAT: usize = 78; const
SYSCALL_NEW_FSTATAT: usize = 79;
const SYSCALL_FSTAT:usize = 80; const
SYSCALL_FSYNC:usize = 82; const
SYSCALL_UTIMENSAT:usize = 88;
```

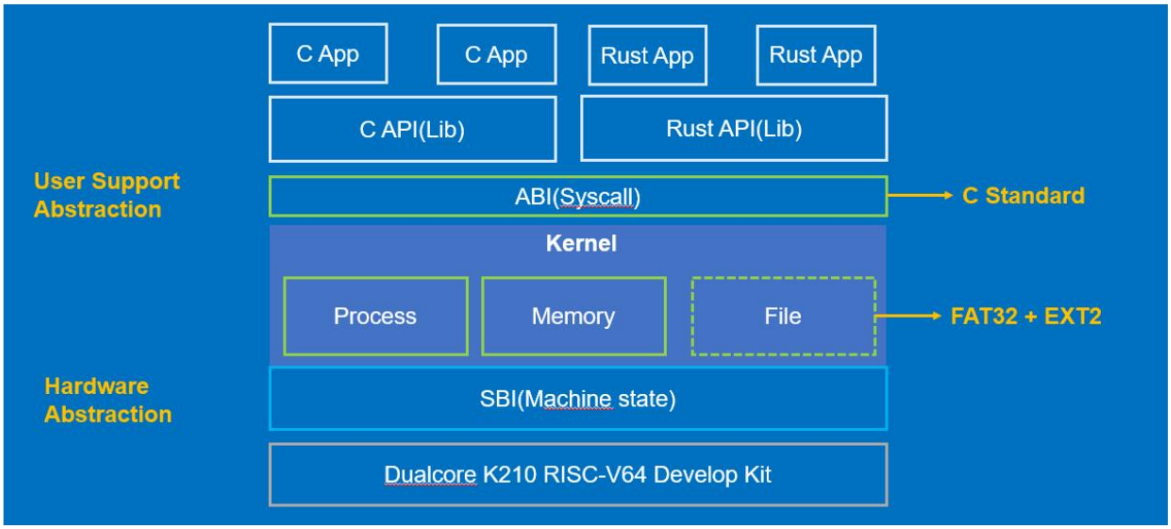
```
const SYSCALL_EXIT: usize = 93;
const SYSCALL_EXIT_GRUOP: usize =
94; const SYSCALL_SET_TID_ADDRESS: usize
= 96; const SYSCALL_NANOSLEEP: usize = 101;
const SYSCALL_GETITIMER: usize = 102; const
SYSCALL_SETITIMER: usize = 103; const
SYSCALL_CLOCK_GETTIME: usize = 113; const
SYSCALL_YIELD: usize = 124; const
SYSCALL_KILL: usize = 129; const
SYSCALL_SIGACTION: usize = 134; const
SYSCALL_SIGRETURN: usize = 139; const
SYSCALL_TIMES: usize = 153; const
SYSCALL_UNAME: usize = 160; const
SYSCALL_GETRUSAGE: usize = 165; const
SYSCALL_GET_TIME_OF_DAY: usize = 169;
const SYSCALL_GETPID: usize = 172; const
SYSCALL_GETPPID: usize = 173; const
SYSCALL_GETUID: usize = 174; const
SYSCALL_GETEUID: usize = 175; const
SYSCALL_GETGID: usize = 176; const
SYSCALL_GETEGID: usize = 177; const
SYSCALL_GETTID: usize = 177; const
SYSCALL_SBRK: usize = 213; const
SYSCALL_BRK: usize = 214; const
SYSCALL_MUNMAP: usize = 215; const
SYSCALL_CLONE: usize = 220; const
SYSCALL_EXEC: usize = 221; const
SYSCALL_MMAP: usize = 222; const
SYSCALL_MPROTECT: usize = 226; const
SYSCALL_WAIT4: usize = 260; const
SYSCALL_PRLIMIT: usize = 261; const
SYSCALL_RENAMEAT2: usize = 276;
```

For the support of the evaluation system, we also need to support non-online library project dependencies, non-external device dependencies

Functions such as startup, automatic running of evaluation programs (both serial and parallel), etc.

3 System Design

3.1 Overall system architecture design



Brief architecture of UltraOS system

UltraOS adopts a modular design idea to ensure the development speed and the robustness of the final result. same

At the same time, the modular development also takes into account the RISC-V architecture's own regulations on privilege levels. The instruction set will typically

The privilege level of program operation is divided into three parts: Machine Mode, Supervisor Mode and User Mode.

The corresponding program execution permissions are distributed from high to low. According to such characteristics, we divide the construction of the operating system into

For three parts: SBI, kernel and Standard Library, the three parts correspond to three privilege levels

run mode. Below we briefly introduce the main design ideas and architecture of each functional module.

SBI, also known as Supervisor Binary Interface, can be seen as a Machine Mode provided to Supervisor

Functional interface to the Mode program. The interface strives to achieve hardware abstraction, including the instruction set of the CPU, and

Other hardware resources owned by the computer, and after abstraction, maintain the industry's norms, and the hardware performance

The inconsistency part, through the same abstraction, can finally make the upper layer software not need to care too much about the inconsistency of the hardware

The specificity and hardware details enable the upper layer software to have good logic and compatibility. in the operating system

Among them, this layer can also be called HAL (Hardware Abstraction Layer). RustSBI is at this layer, the main implementation

ÿÿstdio, hsm(hart state management),ipi(inter-processor interrupt support), reset, timer

five parts.

Kernel, called the operating system kernel, is the core part of UltraOS design, mainly including process management,

There are three parts of memory management and file system. Process management mainly realizes the control of user processes, including

Switching between processes, control of process status, process response to kernel service requests, setting and processing of hardware interrupts
Reason and so on. Memory management mainly implements memory isolation of user programs, including user dynamic heap allocation and page table management.
, page frame allocation, and memory data reading and construction of user programs, etc. The file system is mainly for hard disk and other IO
For devices, we have implemented EXT2-like and FAT32 file systems, as well as a virtual file system using memory as the medium.
The main function of the former is adding, deleting, modifying and checking the hard disk, initializing the hard disk and detecting the file system, hard disk drive and extraction.
Abstraction of the kernel and the abstraction of the file system to the kernel, file system access, modification, update, etc.; the function of the latter is
In order to uniformly abstract physical files, devices, pipelines, etc. into files, it is convenient for kernel management.

The kernel not only needs to implement its interrupt service mechanism, but also needs a synchronization service mechanism, that is, a system call.
The interface composed of system calls is called ABI. The ABI we built meets the call standard specified by the C language, providing users with
Provides user service abstraction. But the directly exposed ABI is not user-friendly, so we also built the user
standard library. At the same time, in order to support the writing of C language and Rust language, we have added C language and Rust language
The standard library of the language means that we natively support the writing and running of cross-language programs.

3.2 Sub-module design

This part mainly introduces three main parts: process management, memory management, and file system.

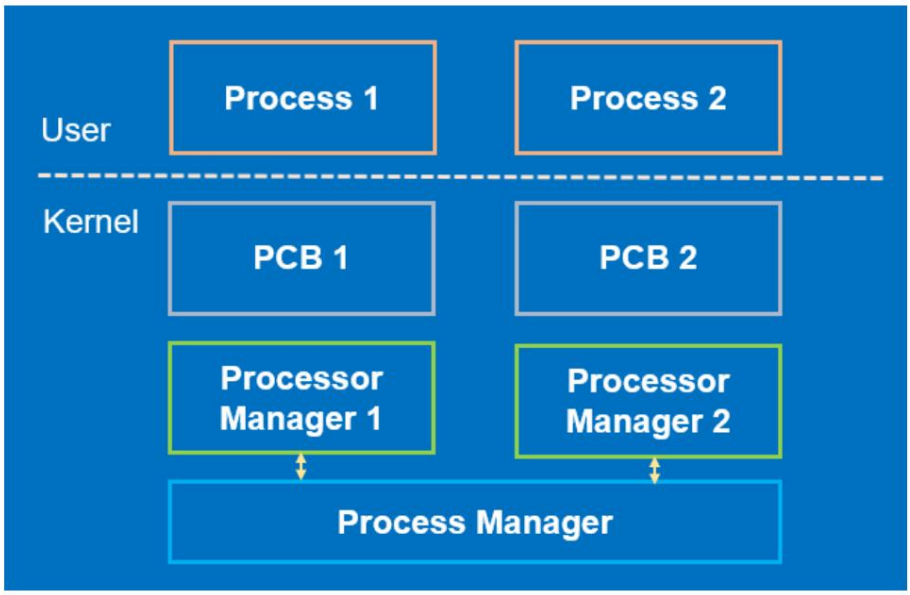
3.2.1 Process Management

The main functions of the process management module are: process initialization, process loading and analysis, process switching, process status
state building blocks.

The process management module is the three core modules of UltraOS, and because it involves the operation of the process, the process management
The module has a very high degree of coupling with the kernel, and it is also the most intricate part. We will process management
Hierarchical distinction, so that we can manage the process with a common logic, this idea
The responsibilities of different modules in the process management are accurately divided. But it is worth noting that the process management
There is a big difference between the division and implementation of the sub-level, because the level locks the specific correspondence between each part, but
It does not focus on specific implementations, or even specific functions.

We divide process management into four levels: process, process control block, core scheduler, and process manager.
Its hierarchical level is arranged in order from low to high.

Its schematic diagram is as follows:



process hierarchy

3.2.1.1 Process control block

The process itself runs in user mode. To abstract its information, we use Process Control Block (PCB) is the process control block to manage information, and the process control block is managed in the kernel. process control block There is a one-to-one relationship with the process. Further, the process control block contains the resource information owned by the process, Process running information, relationship between processes, etc. In fact, because the process is the core object managed by the operating system, The resources it owns include memory and files, so the internal information of the process control block includes memory management and file System related implementation and management.

3.2.1.2 Core Manager

In order to manage the information of the process and perform corresponding operations on it, we introduce the core scheduler, or It is a core manager, and its essence is a process manager owned by each core. Each core in the CPU at the same time Only one process can run at a time. At this time, the core manager will hold the PCB of the process to indicate its ownership, and Corresponding to the CPU core. In this way, we can achieve a one-to-one correspondence, which is convenient and logically managed. because Processes on the core will be switched over time, so the core manager needs to switch between processes.

3.2.1.3 Process Manager

The switching between processes is managed by the process manager, which is different from the core manager in that it does not Not unique to each core, but common to all cores. The reason for this is that UltraOS manages the

From a global point of view, each process may be scheduled to run by any core. process management

The main responsibility of the processor is the scheduling of the global process, including the scheduling implementation of the waiting process, which can support different scheduling

algorithm, and the core manager does not need to care about the scheduling algorithm, it only needs to ensure that it can switch processes.

3.2.1.4 Process mechanism

Before we only focused on how the process is organized, but there are considerable differences in implementation, and at the same time

Specific functions are also under the guise of the interior. Note that the programming language we use, Rust, is an object-oriented

In other words, under our hierarchical design, Rust has excellent implementation possibilities.

According to the design of each abstract level of the process, we follow the corresponding relationship, and also construct the process control block,

Structures and methods corresponding to the core manager and the process manager. But at the same time, the actual design requires a

Changes in some details to support complete implementation, so UltraOS also built a process id number allocator, which

The allocator is used to allocate Pid, and the corresponding kernel stack space. Also, for each struct implementation, I

We not only need to consider the corresponding relationship between them and the internal responsible area, but also need to consider the specific requirements,

Including process copying, replacement and other logic, these are the content of the specific implementation.

In fact, although the process organization is parallel to the file system and memory management, it is actually still at the top.

The layer module, which is mixed with memory and file system, plays a leading role. The role of the PCB is to collect

The final interface of Dacheng probably includes the following parts:

- ÿ Process information: process context, relationship between processes, process identification, etc.
- ÿ Memory information: the memory occupied by the process, including address space, heap, vma, etc.
- ÿ File information: file descriptor table, current path, etc.
- ÿ Auxiliary information: clocks, resource usage and limits, timers, signals, etc.

3.2.2 Memory Management

Memory management is mainly divided into kernel address space, user address space and page table structure. Among them, the kernel uses

The user space is mainly responsible for the startup of the system, the switching between processes, and the stack functions during the kernel mode system call.

function, while the user stack is mainly responsible for loading user applications, running data storage and dynamic content

storage allocation.

The biggest problem with the K210 platform is the lack of memory. Excluding the space managed by SBI, the operating system kernel plus

The space it manages is only 6MB. Therefore, we have made many optimizations in the memory management part, the principle of optimization

It is roughly divided into two types: CoW and Lazy. The specific method is described in the system implementation.

At the same time, in order to achieve the above-mentioned goals, UltraOS designed the virtual address space vma, page table, address space, kernel heap, page frame manager.

ÿ Address space: Describes all owned address spaces and information of the entire process. includes page tables, Vma and page tables and owned page frames.

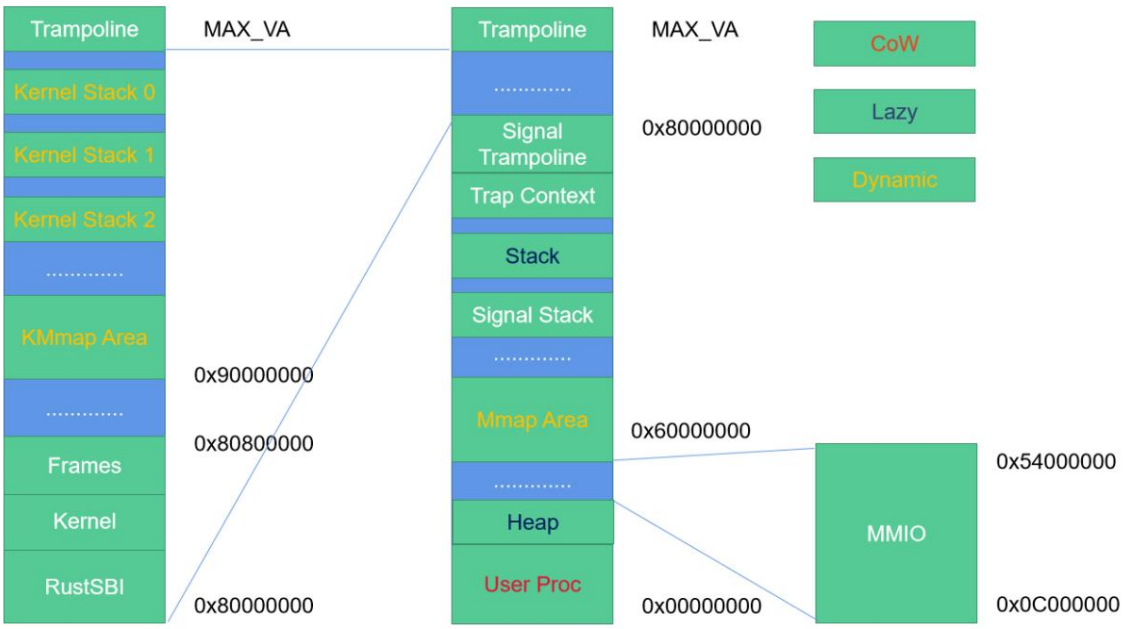
ÿ Page table: manage the page table of the process and the mapping relationship in it. You can manage page table entries, you can also use It does software translation of addresses.

ÿ Page frame manager: used to manage free physical page frames not occupied by the kernel program, including release of page frames release, acquisition, etc.

ÿ Vma: It is mainly born for mmap, which manages the mapping relationship of linear virtual address space.

ÿ Kernel heap: Manages the heap allocation and exception handling of the kernel.

The memory layout of UltraOS is shown in the figure below:



UltraOS fuses the kernel and user page tables so that we don't have to Perform page table switching. At the same time, the prerequisite for fusion is that the virtual address spaces of the kernel and users do not overlap at all.

3.2.3 File system and devices

UltraOS inherits the UNIX design philosophy of "everything is a file". We split the files into actual files and abstracted

There are two categories of image files. The former is an ordinary file, and the latter can be any system object with read and write functions, such as

Such as equipment, pipelines, etc. To this end, our file system is divided into two parts: the disk file system and the kernel virtual

virtual file system.

Ultra OS supports two kinds of disk file systems, namely the simple file system referring to EXT2 and FAT32 files

system. Because the final submission of this version is a file system that supports FAT32, and the overall structure of the two is similar, so this

This article only introduces the FAT32 file system. In our design, the kernel and file system are loosely coupled such that

Better scalability. In addition, because UltraOS supports dual-core, the file system also makes corresponding

Appropriate design.

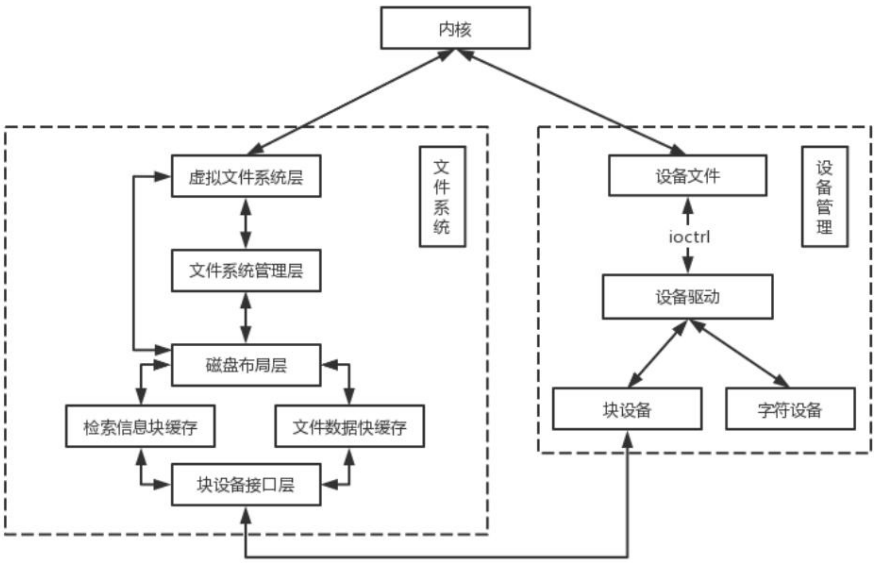
In the kernel, we have a high level of abstraction for disk files, as long as the file system implements the required interface,

Both can be used in conjunction with the kernel. We also abstract any other object with read and write functions, as long as

After the file interface is realized, the kernel can manage it in the form of files.

3.2.3.1 Structural design

The overall structure and data path of the UltraOS file system are as follows:



Disk file system mainly consists of block device interface layer, block cache layer, disk layout layer, file system manager

layer and virtual file system layer. The lower module provides services for the upper module, and the file system management layer is the upper and lower

Multiple modules provide services. The device management part consists of device files, device drivers, and device entities

composition.

3.2.3.2 Function module introduction

This section will briefly introduce the five levels mentioned above, kernel support, and concurrent access.

(1) Kernel virtual file system

The kernel virtual file system coordinates all types of files, and abstracts different readable and writable objects into a unified

Interface, mainly for system calls. Through these interfaces, related system calls can be implemented in a consistent programming mode

Now, it can not only improve the code reuse rate, but also has strong scalability.

(2) Device management

The diversity of equipment is often the practical key to the direction of the system. Due to the nature of the competition, UltraOS did not

There are rich device drivers now, but we have set up a framework for possible future expansion, combining device drivers and kernel

Decoupling. Device management is the bridge between the kernel and device drivers. For the kernel, it needs to provide an interface so that the driver

Automatically obtain control information from the user; for the driver, it needs to provide an interface for kernel control and scheduling.

(3) Block device interface layer

In order to run on virtual machines and development boards, the file system must support different block devices, such as SD cards or

Disk mirroring. The block device interface layer is used to interface with different block devices and shield different blocks for the file system

Set another difference.

(4) Block cache layer

The reading and writing of I/O devices is the key to affecting the performance of the file system. In order to improve performance, it is necessary to use the locality principle

Reasonably design the cache to reduce the number of I/O device reads and writes. Furthermore, in order to avoid overwriting of different types of block data caused by

Efficiency drops, we designed a two-way cache to store file data and retrieve information separately.

Another advantage of using disk cache is that it can shield specific block read and write details, thereby improving efficiency. exist

In our design, the upper module can directly request the required blocks from the cache, and the specific read, write, and replacement processes are handed over to

Done by cache.

(5) Disk layout layer

This layer really starts to organize the file system. FAT32 has many important disk data structures, such as boot

Boot sector, file system information sector, FAT, directory entry, etc. They consist of different fields, storing files

System information, some fields also have specific values. The job of the disk layout layer is to organize these data structures

structure, and provide a convenient interface for the upper layer to obtain or modify information.

(6) File system management layer

The file system manager layer is the core of the entire file system, which is responsible for the startup and overall structure of the file system organization, maintenance of important information, allocation and recycling of clusters, and some practical computing tools. The layer is Other modules provide practical interfaces related to FAT32. If other modules have any related calculation or processing Operations, such as acquiring/reclaiming clusters, calculating addresses, unit conversion, file name processing, etc., can call the module's interface.

(7) Disk virtual file system layer

The virtual file system layer is mainly responsible for providing an interface for the kernel and shielding the internal details of the file system. The job is to implement complex functions. In this layer, we define a virtual file structure to describe the file, It has a corresponding relationship with the short directory entry, which together serve as the entry point for accessing files. This layer implements the common Functions such as create, read, write, find, delete, etc.

4 System Implementation

UltraOS uses Rust language for specific implementation, using rCoreTutorial-v3 (wyf, THU) as Its main framework, RustSBI (Luojia, HUST) is implemented as the underlying hardware abstraction. On this basis, carry out Modified to match all designs described above.

4.1 Process management

When designing the module, we divided the kernel's implementation of the process into three parts, and then we Describe its specific implementation in detail. According to the principle of narrative from the outside to the inside, we follow from the high level to the low level Described next, namely: process controller, core manager, process control block.

4.1.1 Process controller

Currently. The scheduling algorithm adopted by the process controller is FIFO algorithm, so the corresponding implementation is process ready, etc. waiting queue. The process control block references of all waiting processes are arranged in the queue, and each time the core scheduler needs to perform When the process is switched, the process control block at the head of the queue will be taken out from the queue as the next running process of the core.

When the process is ready to wait, it also needs to be put into the queue as a candidate scheduling object. while FIFO

The algorithm will dequeue according to the order of entering the queue.

```
pub struct TaskManager {
    ready_queue: VecDeque<Arc<TaskControlBlock>>,
}
```

Here, we can freely implement various scheduling algorithms, but there are only three necessary interfaces: initialization,

Add ready process and remove ready process. The process controller can use other algorithms for process scheduling optimization

priority.

Note that because UltraOS adopts a multi-core design, for access to the shared process controller, we set

A mutex is designed to ensure that only one core can access the structure at a time.

4.1.2 Core Manager

For the core manager, it stores the currently running running process and its unique kernel stack

Pointer, the kernel stack pointer is dedicated to process switching.

The Idle stack pointer, the unique kernel stack pointer just mentioned, is set to the kernel's

Initial stack, each core has a unique kernel initial stack, when the process is switched, the kernel holds the

Program unique kernel stack, and when switching, it is necessary to switch the kernel stack to the Idle stack, and the corresponding temporary

registers are saved. The Idle stack is actually the initial stack, but the initial stack disappears after the initialization of the kernel is completed.

It has become an Idle stack. The reason for this distinction is that the Idle stack only serves for process scheduling. In fact, the Idle stack corresponds to

The location of the program indicated by the context is an endless process scheduling loop, always looking for the next switch

process, if the next process is successfully obtained, the stack will be switched to the kernel stack of the corresponding process, if no possible

Switch process (no ready process), return to the kernel stack of the original process.

Except for the use of the Idle stack, other uses correspond to the specific use of the process control block. First, nuclear

The processor provides basic usage methods: get information about the currently running process, get the currently running core

Core information to obtain the core's Idle stack pointer.

Of course, the most important thing is the process scheduling of the core manager. It will keep looking for the next process if

If the next process is successfully obtained, the stack is switched to the kernel stack of the corresponding process. If no switchable process is found (no

There is a ready process), then return to the kernel stack of the original process.

4.1.3 Process control block

The process control block is mainly responsible for the control of the process. Process information includes: Trap context pointer and corresponding page

Frame, heap information, segment information (including page table), process parent-child relationship, process current status and exit code, progress

The file system path of the process and the file resources it owns.

Its content can be summarized as:

ÿ Process information: process context, relationship between processes, process identification, etc.

ÿ Memory information: the memory occupied by the process, including address space, heap, vma, etc.

ÿ File information: file descriptor table, current path, etc.

ÿ Auxiliary information: clocks, resource usage and limits, timers, signals, etc.

```
pub struct TaskControlBlockInner {
    // task
    pub trap_cx_ppn: PhysPageNum,
    pub task_cx_ptr: usize, pub
    task_status: TaskStatus, pub parent:
    Option<Weak<TaskControlBlock>>, pub children:
    Vec<Arc<TaskControlBlock>>, pub exit_code: i32, //
    memory pub memory_set: MemorySet, pub base_size:
    usize, pub heap_start: usize, pub heap_pt: usize, pub
    mmap_area: MmapArea, // file pub fd_table: FdTable,
    pub current_path: String, // info pub address:
    ProcAddress, pub rusage: RUsage, pub itimer:
    ITimerVal, // it_value if not remaining time but the time t









oh alert
    pub siginfo: SigInfo, pub
    trapcx_backup: TrapContext, // resource
    pub resource_list: [RLimit;17],

}
```

Based on the information it has, the process control block provides the most basic methods, including: Get the current process's

Stack top pointer, page table, state and other methods. The complex methods provided by the process control block are described next.

(1) Create a new process (control block): the process control block must uniquely correspond to a process, so its initialization

It is not possible to just fill in the data of the corresponding structure, but to build from the data of the process.

ÿ The process control block will construct the user address space according to the content of the program (the program is set as an ELF file)

And at the same time allocate the physical page frame to the corresponding space.

ÿ Assign the corresponding Pid and the kernel stack bound to the Pid. Initialize the kernel stack with an empty context to

It adapts to the existing process scheduling operation structure design.

ÿ Create the corresponding process control block and pre-allocate three file descriptors (standard input and output and error).

At the same time, we need to note that the direct creation of a process is only likely to be the creation of the first process initproc.

His processes are created by fork or exec.

(2) Copy (fork) process (control block): in general, it is to copy an almost identical

process (and process control block), so we need to construct an identical user space and allocate

The corresponding space also allocates Pid and the corresponding kernel stack. But note that fork also needs to establish a parent-child relationship.

(3) Dynamic heap growth: according to the growth (decrease) of the heap, allocate (recycle) the corresponding page frame, and record

Record the corresponding heap information.

4.1.4 Auxiliary information

Auxiliary information includes three categories:

ÿ Timer: This timer is responsible for sending time soft interrupt signals periodically or once according to time.

ÿ Resource recorder: record resource usage during process running, including time consumption, and fall into the kernel

times and so on.

ÿ Signal processor: used to store and process signals.

4.2 Memory Management

Memory management is mainly designed for user and kernel memory address spaces.

At the same time, memory management is not simply a matter of address design. UltraOS also needs to meet the needs of processes and file systems.

The unique requirements of the system, the former we take the design of the trap context of process switching as an example, and the latter takes the design of mmap

as an example.

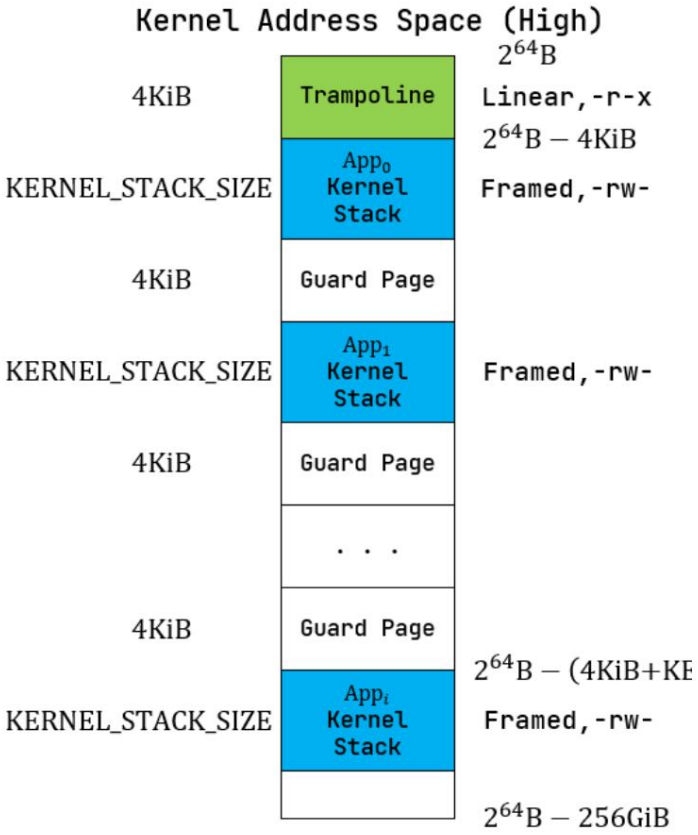
4.2.1 Kernel address space

From the start of the operating system, the kernel address space needs to load the data required for startup, and load

No. 0 initial process initproc and user command line process usershell. In the process of running the program, the kernel address space is divided into the kernel

stack space used by multiple applications, and separated by the guardpage mechanism

open.

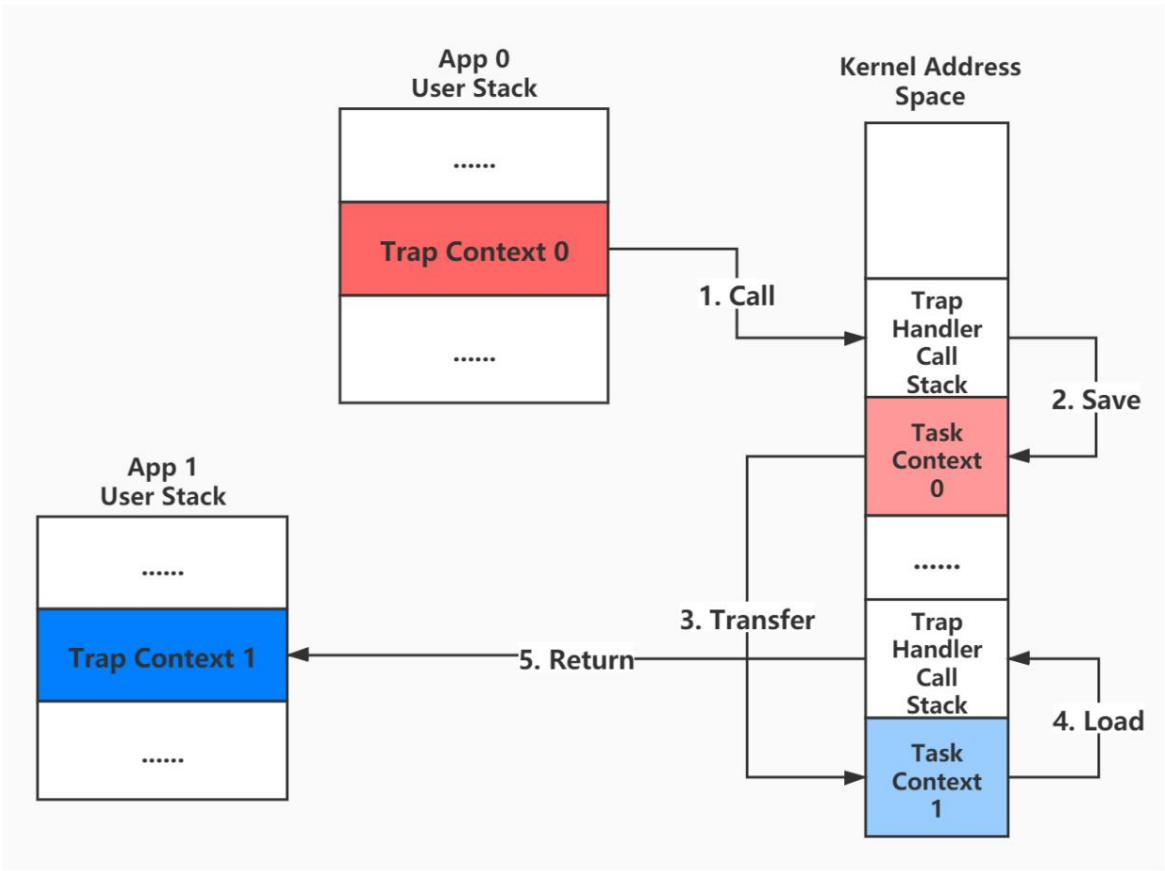


4.2.2 Trap Context Corresponding Memory Design

When the operating system switches between different processes and executes different user programs in parallel, it needs to enter

The kernel mode is used to complete process scheduling and switching tasks.

As shown in the picture,

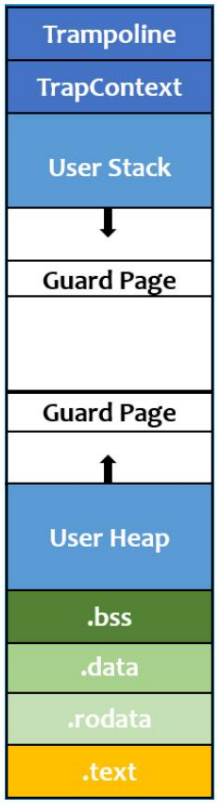


When you need to switch the process, you first need to enter the kernel state through the trap, so you need to pass the TrapContext structure to save the register context of the current user mode (saved in the kernel corresponding to the user program address space stack), and enter the TrapHandler, the relevant Trap information is also stored in the TrapHandlerCallStack. Complete system scheduling in kernel mode and prepare to switch to the next user mode

At sequence time, save the information of the previous process through TaskContext. At this time, the stack pointer sp can be switched to another Kernel stack and load the TaskContext of another soon-to-run process, and pass the next process's TrapHandlerCallStack and TrapContext are restored to the site of the process, and return to the user state, and start to process Execute the next user program.

4.2.3 User address space

The figure on the right shows the user address space distribution.



The user address is mainly managed through the data structure of MemorySet, and

And through the page table and physical address space mapping, create a new user application program each time

When the corresponding process is executed, a dedicated MemorySet will be created, and data will be read

from the binary file through the from_elf interface. MemorySet Below

Contains each section of space (data, text, heap, stack, ...) in the user address,

And they are represented by a separate data structure, which will be passed when creating a new one.

The map function creates a mapping in a page table.

When performing fork or clone, MemorySet also provides special

The interface of the gate is used to realize the generation of new processes through these two system calls

```
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}
```

Methods of MemorySet.

The springboard is at the top of the user address space, linking the virtual address and the real address (the virtual address of the springboard and

The real address is the same for each process), used to switch between different user process spaces, and

And you can save the TrapContext here when the trap is trapped.

The stack space of the user address is used to transfer parameters when functions call each other, and the one under the user stack is the

The application's pre-allocated static memory space.

The heap space of the user address starts from the top address (except for the springboard and TrapContext) to the

The next growth is the dynamic memory space, which is increased through the sys_brk() system call, and

The space is managed by BuddyAllocator, and the user program can obtain it through the new() function.

sys_brk is also realized through the system call of sbrk. sys_sbrk is the dynamic allocation of user heap space

Memory interface, which can grow and shrink the dynamic memory of a process, through the grow_proc() function

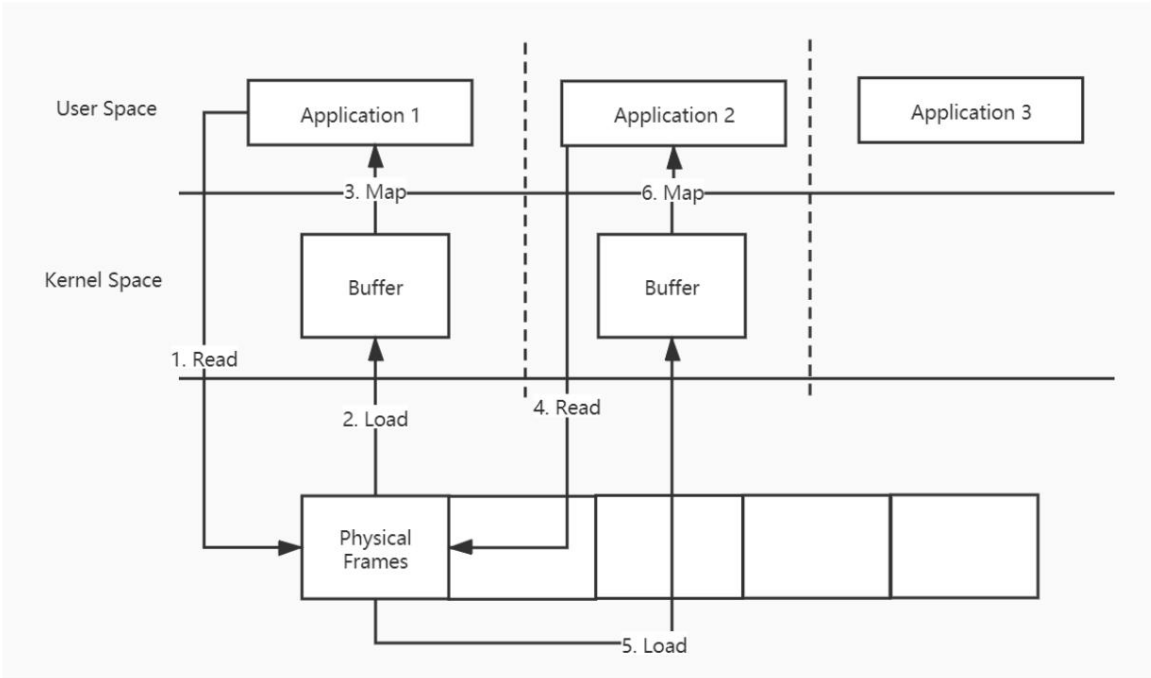
now.

The heap space also belongs to the MemorySet of the process in which it resides. At the beginning, the space size is 0, and

The heap pointer moves up and down as the space grows and shrinks. Through the grow_proc function, the user memory exists in the heap space Increase the size of the response in the middle, and pass the entire memory address to buddy_allocator, so the dynamic memory space of random size can be obtained in the application through new() of BuddyAllocator.

4.2.4 mmap and munmap design

Normally, if users need to access files or I/O interfaces, they need to call sys_open of the kernel, sys_read, sys_write, these provided file system syscalls to operate. As shown in Figure 1, if multiple processes When their respective applications need to access the same certain disk block, they need to be in their own user memory space Create a corresponding file buffer buffer, so as to realize the read/write operation of the file, and receive the limit of the file lock, It is not possible to write to the same file at the same time.



normal file buffer

But through the sys_mmap and sys_munmap system calls, you can move between file disk blocks and memory space At this time, the file block can be regarded as a memory space for reading and writing operations. need a certain When the disk block is called by the process for the first time, the corresponding space is opened in the memory address, and the process is performed through mmap Mapping, mapping it to a virtual memory address, and building a page table. When accessing this memory at the very beginning,

Trigger the invalid exception of the page table. At this time, the exception handling interrupt will read the data in the disk block through the file operation and

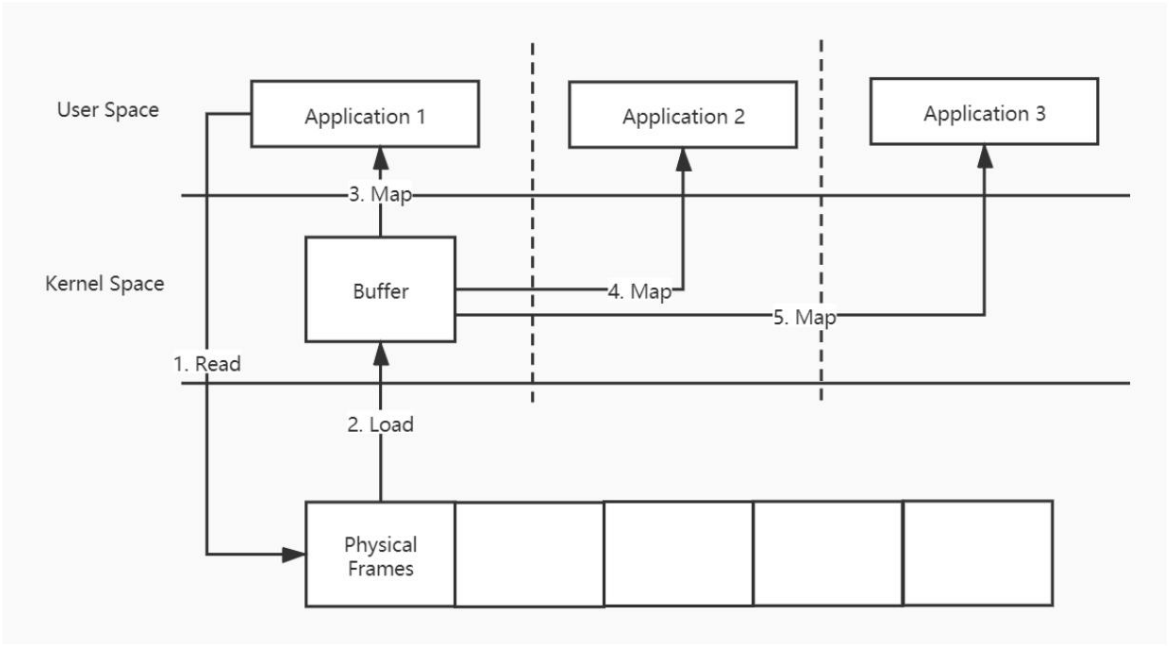
And copy it into the memory; after that, you can directly operate on this piece of memory data every time. and has a new

When other processes want to access this file, calling sys_mmap again can directly share this memory space, and according to it

The permission flag to determine whether this memory can be shared to read/write.

When a process finishes accessing this file block and needs to release it, call sys_munmap, if all processes bind

If the specified space is munmapped, the mapping will be released through the system call.



mmap file buffer

The space of mmap is also allocated in MemorySet by establishing a corresponding data structure when the process is established, but the

allocation of virtual address space of mmap cannot be flexibly managed in MemeorySet.

Therefore, the data structures of MmapArea and MmapSpace are added to manage disk file blocks and corresponding

The mapping relationship between the physical buffer and virtual memory and the establishment of the page table, these two data structures themselves are not

Does not allocate memory space, but calls the interface of the page table to establish the mapping of virtual and real addresses, and records itself

All information related to mmap in order to dynamically create and adjust mmap space.

Whenever a new mmap is created, first pass the size of the file to be mapped to Mmap, and get the file size that can be created

Set up the virtual address mapped by the page table, and then complete the address mapping in TaskControlBlock. Then enter as

This mmap and the newly created MmapSpace read and write the content of this file through the fat32 file interface

Enter the corresponding physical buffer address to complete the mmap system call.

4.2.5 kmmap design

UltraOS initially uses the kernel heap to read executable files and store them in user space after parsing. The kernel heap uses

The split nature of the partner management system means that significant additional space is required to support large executables

Files such as Busybox require additional space greater than 2M. But in practice, we don't need to

Large kernel heap, so these page frames allocated to executable files need to be released in time.

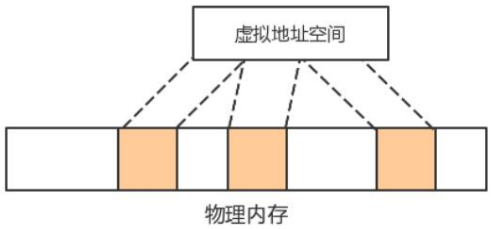
For this we implemented kmmap. kmmap is the mmap of the kernel service, which maps files to memory,

At the same time, continuous virtual addresses are provided to the kernel through non-identity mapping. With kmmap, we can be more flexible

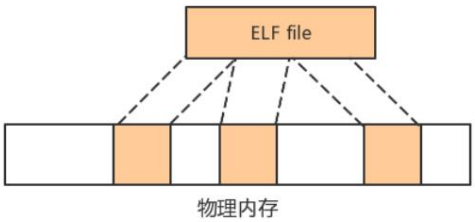
Allocate and deallocate memory for files loaded by the kernel. For example, when loading a user program, it has the following process:

ÿ According to the file size, the memory manager allocates enough page frames, and at the same time establishes a continuous virtual address in the kernel page table

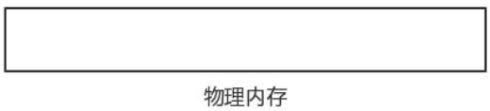
mapping.



ÿ Load the executable file into the mapping area for parsing



ÿ After parsing is complete, release the page frame



After our tests, when using the heap to load files, the total memory required to run Busybox is greater than

6.5MB. When using the kmmap strategy, we can reduce the heap to 512KB, and finally only need a peak occupancy of 4.38MB to run the

kernel and a complete Busybox, reducing the peak memory occupancy by 32.6%.

4.2.6 Copy on Write optimizes the memory occupied after fork

The specific method of the CoW strategy is that when a process calls fork() to generate a new process, it needs to create a new process for the new process.

The process creates a new address space MemorySet, the content of this address space should be the address space of the parent process

An exact copy of , so the content is the same. In the original case, this process would perform a full copy, but

Since the child process may immediately call exec() to load the elf file of the new user program, the original memory is completely

If it is completely cleared, the original copying time is wasted. CoW approach is to create a new child process, the child

The virtual page of the process corresponds to the original physical page of the parent process, that is, the parent and child share the same physical page.

In this case, both the parent and the child are only allowed to read these spaces and cannot perform write operations. If they want to perform write operations

In other words, the writing party will apply for the allocation of a new physical page frame for the virtual address page to be written.

This virtual address page can be remapped in its own page table. At this time, a single physical page frame can be exclusively used, or it can be

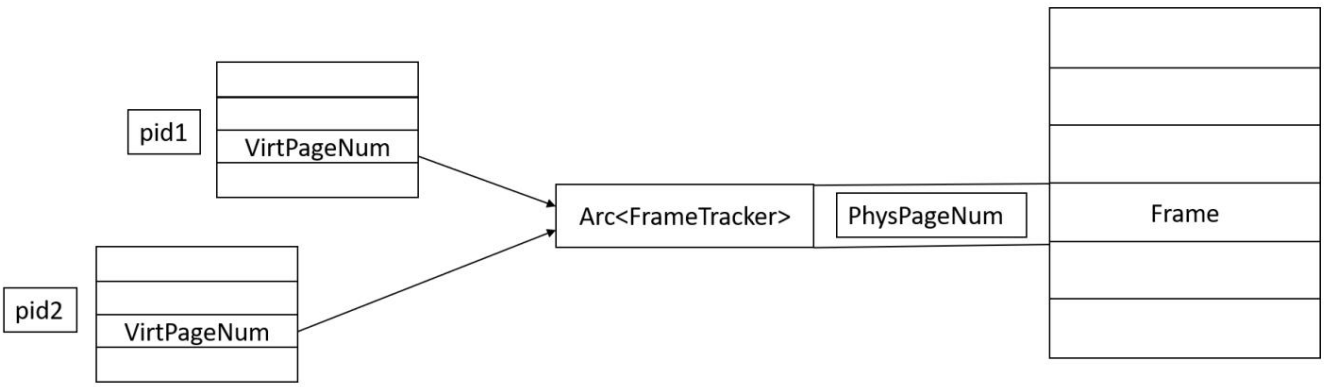
Row write operation.

The number of references RefCount to a physical page frame determines whether it is a physical page of a CoW policy. When one

When the virtual page of a new process maps this physical page frame, its RefCount+1, when RefCount>1

When , it means that more than one process references the physical page frame, that is, CoW is used; when a process takes

When unmapping, RefCount-1.



4.2.7 Lazy Stack & Heap: Dynamically allocated stack space

In UltraOS at the beginning, we used pre-allocation and mapping to reserve for each user program

A fixed-size heap/stack space, but in fact this space is not fully allocated during the running of most programs

Use all of it, or just a very small part of it. Therefore, as the heap grows (brk) and the stack grows (push)

To dynamically grow, when a virtual page needs to be used, then allocate a physical page frame to it, which can save enough

Enough space, in order to avoid insufficient physical memory space that can be allocated when the program performs high-intensity fork().

But for the stack space Stack, each program will use at least a certain amount of memory, which causes each program

The first few stack spaces must be allocated. Under Lazy's strategy, every time you have to enter the trap to allocate them,

In fact, it slows down the execution speed of the system. We refer to the method of Linux, which happens to assign a small amount to each process in advance.

A fixed amount of stack space, for most programs with small stack usage intensity, these pre-allocated spaces are just right

It is enough for them to use without triggering lazy alloc. For a small number of programs with high intensity, the space allocated later is different

It is determined that all are used, so trigger the trap to allocate when it is used.

Different from the continuous storage method of the program segment and data segment corresponding to the elf file in the memory space, the lazy strategy

The virtual pages that have been mapped to the page table in the stack segment may be discrete and discontinuous, and the addresses are also separated, so

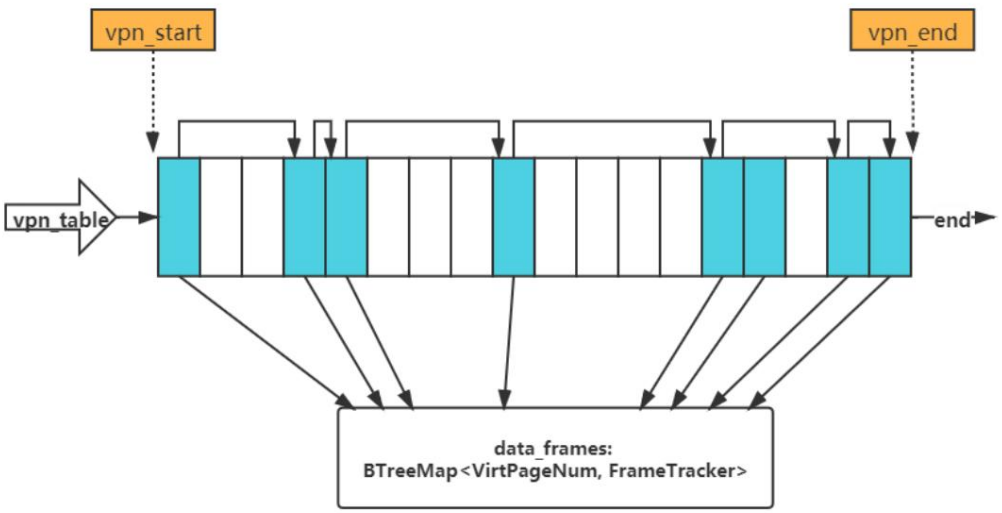
This is different from the method of saving memory space segments through VPNrange in the maparea data structure. We use a data structure

similar to a linked list to string together the scattered stack memory virtual pages allocated by lazy. Each segment corresponds to a

The linked list of virtual pages, of course, also needs other information, such as the start and end addresses of the segment, so we put this

These contents are encapsulated in the data structure of chunkarea, and integrated into a certain process like maparea

Go in MemorySet.




```
pub struct Chunk
    vpn_table: V
    data_frames
    map_type: Ma
    map_perm: Ma
    mmap_start:
    mmap_end: Vi
}
```

After optimizing lazy, we focus on its main goal: the saving effect of memory space (physical page frame)

After the test, the programs used for the test are Busybox and lmbench in the first and second stages of the final.

	without lazy alloc			with lazy alloc			Compare
test on program	before runing program	after	**pages consume**	before runing programafter	after	**pages consume**	**improve**
lmbench_all lat_syscall -P 1 null lmbench_all lat_select -n 100 -P 1 file lmbench_all lat_sig -P 1 install lmbench_all lat_sig -P 1 catch	1107	662	445	1232	912	320	28.02%
lmbench_all lat_proc -P 1 fork lmbench_all lat_proc -P 1 exec lmbench_all lat_proc -P 1 shell	1107	566	541	1232	880	352	34.94%
busybox_all	1103	368	735	1279	624	655	10.88%

From the test results, we can see that in lmbench test programs such as fork/exec, the lazy strategy will single

The memory space occupied by the program is optimized by about 35%.

4.2.8 Lazy Mmap: mmap space saving strategy

In the busybox test, we frequently encountered insufficient memory and exhausted physical page frame allocation.

Part of the reason is that mmap is called in the test to perform memory mapping on a very large file (exceeding the total hardware memory size)

Shooting operations, but only a small part of them are actually used for reading and writing. Therefore, in the process of mmap, a large part of

file data does not need to be mapped into the memory in advance, and the same lazy strategy as the stack is adopted.

Slightly, only use trap to perform page table mapping and memory allocation on the page currently to be read and

written. We have tried a more flexible allocation strategy. When the size of the file mapped by mmap is only 1 page, the large

The probability is to read/write this page later, so we judge the size of the mapping when mmap

Break, if len<=PAGE_SIZE, then directly allocate memory space for mapping, otherwise adopt lazy strategy.

4.3 File system and devices

Because the file system and device management system are very large, only the implementation of the core functions of the modules at each layer is introduced here.

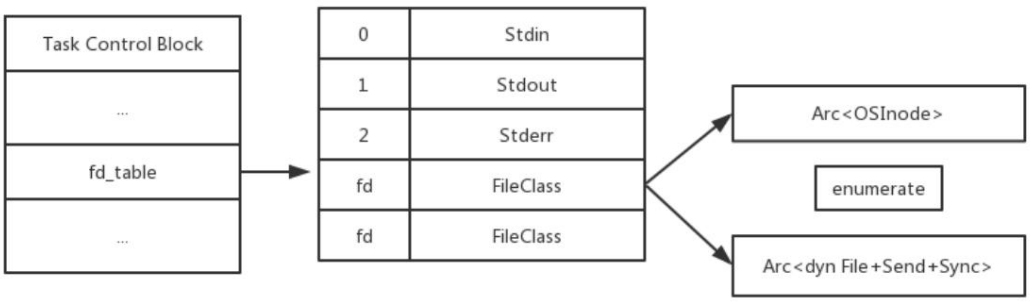
4.3.1 Kernel Abstract File System

In the kernel, we define the OSInode structure to represent the file, which contains the read and write tags, when forward offset, and a reference to the corresponding virtual file. Whether it is a file or a directory, OSInode is used in the kernel form exists. In addition to the actual file, there are also some objects that can be read and written, such as some devices, Pipe, etc. They all implement the file interface File. File contains four methods: determine readable/writable, read, and write.

Stdio controls the UART implementation through the RustSBI interface, and does not have physical files, so such abstract files Components cannot be described by OSInode. The problem is that each process uses the file descriptor table (fd_table) Maintain open files. In the implementation of rCore-Tutorial, it uses Rust's dynamic distribution mechanism, in fd_table stores references to arbitrary structures that implement the File interface. But the problem with this is that taking from fd_table The out-of-file references can only call the methods contained in the File, and OSInode-specific methods, such as find and create, are all Not available.

In order to solve this problem, we define an enumeration class FileClass, which currently contains two types, divided into Not abstract files and entity files. The abstract file uses a dynamic allocation mechanism for any structure that implements the File interface Body reference, entity file is a reference to OSInode. Storing FileClass in fd_table can ensure maintenance Different types of files can also maintain different types of files.

Under this design, the files maintained by the process are as follows:



It can be seen that the abstract class file also implements the Send and Sync interfaces, which is Rust's cross-thread security in order to ensure Features provided for sharing.

In addition to maintaining open files, the kernel also supports mounting. The specific implementation is to maintain a mount table, access the target When recording, first retrieve the mount table, and if the directory mounts other directories, then access the mounted directory. Because the game is temporarily There is no need to mount different devices, we have temporarily disabled this function to improve performance.

4.3.2 Device Management

The system usually uses the device tree to obtain device information, which is provided by the Bootloader. But since UltraOS uses
Bootloader does not provide a device tree, and considering efficiency issues, we do not use the device tree for the time being, and only call the platform
The relevant library implements the driver for the required hardware, and retrieves it in the form of a table.

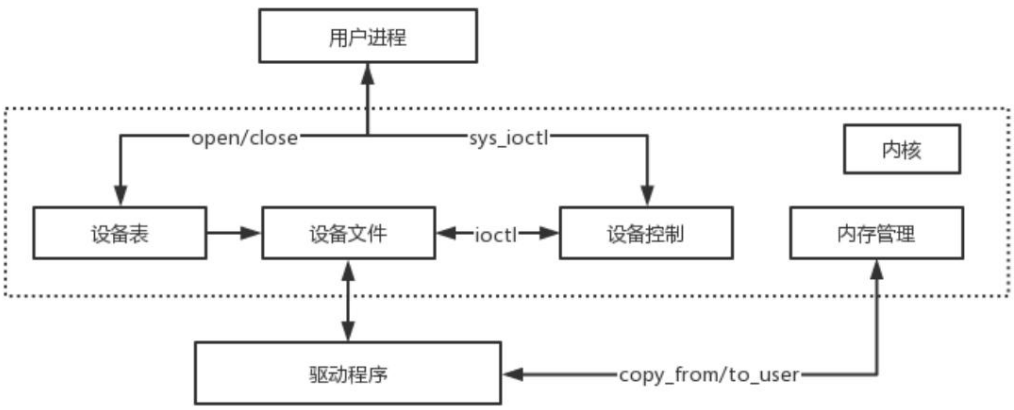
The purpose of device management is to provide a bridge between the kernel and drivers. For scalability, UltraOS hopes to
All kernel-independent programs are decoupled, including drivers. Therefore, we need to define a unified interface,
Facilitate kernel and driver interaction.

Usually, the user process controls the IO device through the sys_ioctl system call. control of sys_ioctl
The control object is a file, so we abstract all required devices into virtual files stored in memory. specific
In terms of implementation, we implemented the File trait for the device and added the ioctl method to the File trait. Device drivers can be rooted
Implement some methods defined by the File trait as needed. When the system starts, each device is initialized and loaded into the device table
for maintenance.

The parameters of sys_ioctl are file descriptors, control commands, and control parameters. The difficulty in realizing it is that the control
The control parameter may be an integer or a structure. This leads to the fact that it is impossible for the kernel to target different classes
Different types of control management, but all control should be managed by the target device driver, which is also necessary
Reasons for decoupling. To this end, our design goal is that the kernel does not need to know what the user is doing to the device.
control, and does not need to know how the driver implements the control; the driver does not need to touch the user's address
Space, you can only get the parameters you need. Under this goal, the kernel is only responsible for data forwarding, and the user and device
The behavior of the device is completely transparent.

We have implemented two interfaces of copy_from_user/copy_to_user in the memory management module of the kernel. pass
With these two interfaces, the driver can pass structure parameters of any size between the user and the user. we used
UserBuffer is used to assist the implementation of the interface to deal with the situation where the structure spans pages.

The device management framework of UltraOS is as follows:



4.3.3 Block device interface layer

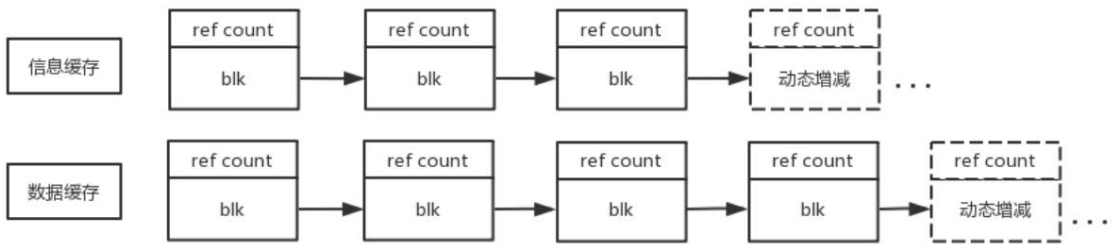
The block device interface layer defines the interface that the block device driver needs to implement. Through this layer, the file system accesses the block Devices will be transparent. We use Rust's Trait to define an abstract interface for block devices, which includes read_block and write_block two methods. For device drivers, in order to improve the efficiency of development, we use the open source project The current SD card driver of rCore-Tutorial.

4.3.4 Block Cache Layer

The file system uses blocks (sectors) as the basic unit of reading and writing, so the basic unit of caching is also blocks. Refer to CPU cache In order to design the separation of storage instructions and data, we divide the block cache into information cache and data cache. information cache for Blocks that store retrieval information, such as file system information sectors, FAT, directories, etc., and data caches are used to cache save file data. The advantage of this is obvious: because the development board memory is small, we cannot set a large Block cache, if double cache is not used, once a relatively large file is operated, the block of the file will fill the entire cache, Overwrite the retrieval information block, resulting in inefficient operations such as directory operations, retrieval operations, and information modification, especially when modifying When changing files, FAT and disk information sectors are frequently accessed. Using two caches avoids the problem of overwriting.

For the cache itself, we set its single size to 10 blocks, which is 5KB, and the total double cache is 10KB. because Because UltraOS has not yet implemented a perfect clock, it is difficult to implement a rigorous LRU strategy. For this reason, we have used nearly LRU-like algorithm - Clock algorithm. We use a queue to maintain all cache blocks, the queue uses The VecDequeue provided by the alloc library can dynamically increase or decrease on the heap, thereby flexibly reducing memory usage. arts The upper layer of the hardware system accesses the cache block through the smart pointer Arc, which counts the reference atoms. When the queue is full, look for Blocks with 0 references can be replaced.

The cache structure is shown in the following figure:



Through caching, we shield the specific details of sector reading and writing. We define get_block/info_cache to And modify_block/info_cache four interfaces, through these interfaces, the upper module does not need to consider the start of the partition For problems such as offset, you only need to ask the cache for the required logical block number, and the cache will help it get it. In order to make the upper mold The block can directly operate on the cache block according to the required data type. These four interfaces support passing in the closure, through the closure

The package can specify the data type and the process of processing, and then operate on the data of the specified type in the block.

The UltraOS file system supports partitioned disks, which actually requires cache support. start in management

After the file system, it will read the number of hidden sectors (the offset of the first partition) from sector 0, and pass it to the cache

After that, the cache layer will add this offset every time it reads and writes the block device, and then the upper module only needs to use

Use logical block numbers.

4.3.5 Disk layout layer

The disk layout layer is used to organize FAT32-specific data structures. It should be noted that K210 requires aligned read

Write, that is, the address of a variable of a certain type must be aligned with the size of the type. By viewing the FAT32 document, we can see that the guide

Some fields of a sector are not aligned, for example halfword type fields may be byte aligned. For these fields,

We read it in the form of a byte array, and design an interface to encapsulate its reading and writing so that the upper layer can directly use the field type control.

First introduce the boot sector and the extended boot sector. They give basic information about the block device, such as sectors

Size quantity, cluster size, version, verification signature, number of reserved sectors, location of file information sectors, etc.

If the device is partitioned, the boot sector also contains the start sector of the first partition. boot sector information

Usually do not need to be modified, they are only responsible for checking and providing relevant information to the management when the file system is activated.

interest. In fact, these two sectors are only read when the file system starts. Because the fields of these two sectors are relatively

More, we define a complete structure, and use all fields as members to correspond one-to-one with disk data. Make

When using it, it is necessary to read the corresponding block from the cache, and then convert the reference to the block into a reference to the corresponding structure.

Next is the file system information sector. This sector has five important fields, the first two fields are verification signatures,

The third is the number of remaining clusters of the FS, the fourth field is the initial free cluster, and the fifth field is the verification signature. Obviously,

The third and fourth fields often need to be read and written. The valid fields of this sector are distributed at the beginning and end of the sector, and the fields in the middle are all invalid

Byte, in order to save space, it is not necessary to create a structure containing the field members of the entire sector. We define a

FSInfo structure, which only contains a member of the sector number, but implements rich interfaces, including signature verification

verification, reading and writing of the cluster information field. It is reasonable to reserve only one member of the sector number, because the reading and writing of the sector is through the cache

storage, and the interface provided by the cache only needs sector number and offset information.

Then there is the directory entry structure. The structure of directory items is complex and there are many fields, so a complete structure is still achieved

Volumes correspond one-to-one with disk data. The directory entry of FAT32 is 32 bytes long, including long file name directory entries and short files

There are two types of file name directory entries. When the file name is long, use multiple long name directory entries plus a short name directory entry.

form storage. All information about the file is stored in the short name directory entry, including name, extension, attributes, creation/access

Ask/modify time, size, starting cluster number, etc., so we use it as the access entry of the file. for long and short names

Directory items, we have designed a rich interface for encapsulation, so that the upper layer can use intuitive data when accessing

Read and write information by type without having to consider complex internal structures. The core interface of the short name directory entry is as follows:

interface	describe
initialize	Initializes a directory entry with the specified information
get_pos	Get the sector and offset of the file
read_at	read file at specified offset
write_at	Write the file at the specified offset, the upper module needs to ensure that the file size is sufficient
as_bytes/as_bytes_mut convert data structure to byte array	
checksum	Computes the checksum of the short filename used to populate the fields of the longnamed directory entry

The short name directory entry is relatively simple, which uses 11 bytes to store the file name, including the 8-byte name and 3-byte extension

name, encoded in ASCII. Long directory entries use scattered 26 bytes to store file names, encoded in Unicode

code. In order to shield the internal storage details, we implemented the get_name interface, which returns in the form of a string

The filename in the directory entry, for use by other modules.

Finally, there is FAT, the file allocation table. FAT is the core of the FAT-like file system. In FAT32 file system

In , files are organized in a chain structure with clusters as units, and the next cluster number of each cluster is stored in FAT. FAT32

There are 2 FATs, and FAT2 is used as the backup of FAT1, so the write operation needs to be performed synchronously, and the read operation finds a fault

When a failure occurs, it needs to be switched to another FAT in time. In terms of implementation, we define a FAT structure, which contains two

Members are the starting sector numbers of FAT1 and FAT2 respectively. As a result, all operations on FAT are directly through the cache

save. We have implemented a rich interface for FAT, and the core interface is defined as follows:

interface	describe
calculate_pos	Calculate the sector and offset where the cluster number corresponds to the entry
next_free_cluster Get the cluster number of the available cluster	
get_next_cluster Get the next cluster of the current cluster	
set_next_cluster sets the next cluster of the current cluster	
get_cluster_at	Get the i-th cluster of a cluster chain (i is a parameter)
final_cluster	Get the last cluster of a cluster chain
get_all_cluster_of Get all clusters of a cluster chain starting from the specified cluster	
count_cluster_num counts the number of clusters from the specified cluster to the end of a cluster chain	

The algorithm implemented by the above interface is relatively simple, mainly for repeatedly querying the FAT, so I won't go into details here. but through this

With such a package, other modules can easily operate on FAT. For example, the management module can intuitively

Clusters are allocated and reclaimed, and the virtual file layer can calculate the number of clusters occupied by files or directories through the interface.

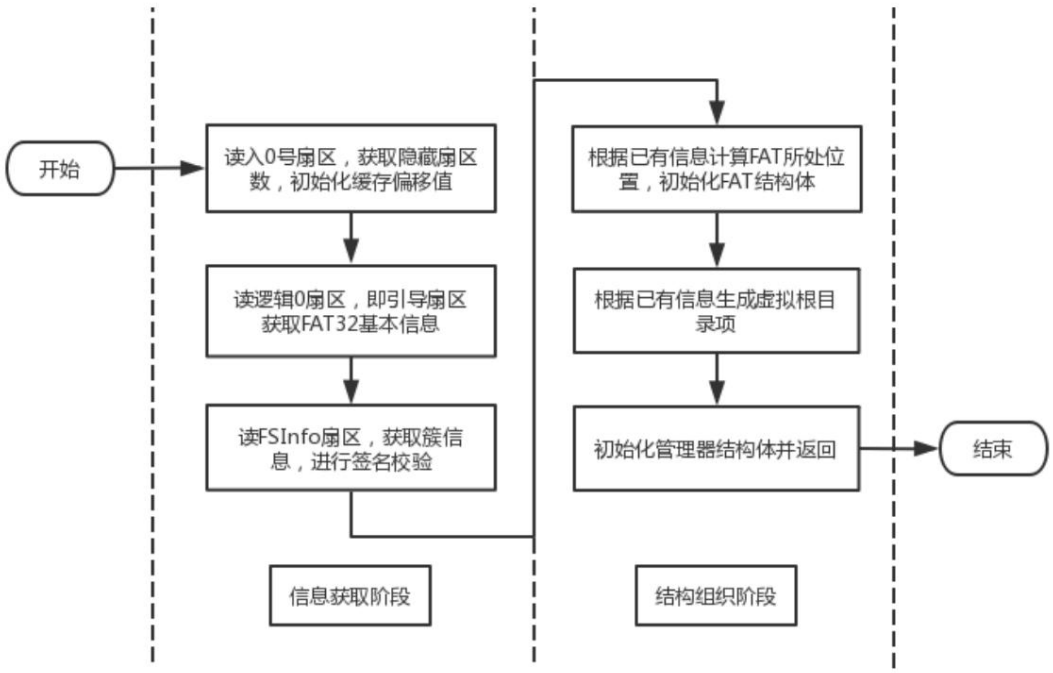
4.3.6 File system management layer

The structure of the manager is defined as follows:

FAT32Manager	
member	describe
block_device	A reference to a block device, using Rust's dynamic allocation to support different devices
fsinfo	A reference to the file system information sector
sectors_per_cluster the number of sectors per cluster	
bytes_per_sector bytes per sector	
fat	References to FAT
root_sec	Update the cluster number of the directory
vroot_direct	Virtual root entry. The root directory has no directory entries, imported to be consistent with other files

The first task that the manager is responsible for is to start the file system, which needs to read the data of the boot sector and perform

Verify, and then simply organize the data structure. The startup process of the file system is as follows:



For other modules, the management layer provides practical tools. Whenever FAT32-related calculations are required, or

The relevant information of the file system or the file system can call the interface of the management layer. Its services include computing files from start sector, split long filename, format filename, generate short filename, calculate clusters required for file size, obtain Get available clusters, unit conversions, and more. Among them, file name formatting refers to converting from strings to stored in directory entries The byte array format of , and complete the vacant positions at the same time. FAT32 uses long name in the case of long file name The directory entry is stored in the form of entry + short-name directory entry, and the file name in the short-name directory entry is compressed from the long file name. The commonly used compression method is: take the first 6 digits of the file name + number + suffix. The short file name generation interface is used to implement the Function. Algorithms of other interfaces are relatively simple and will not be described here.

4.3.7 Virtual file system layer

In the virtual file system layer, we abstract files into virtual files, and design the VFile structure, whose composition Members include commonly used file information, the sector and offset of the short directory entry of the file, FS manager and block device interface references. As mentioned above, this file system uses short directory entries as access entries, so the virtual files and short directory entries here corresponding to the entry. The following are the important interfaces provided by VFile to the kernel:

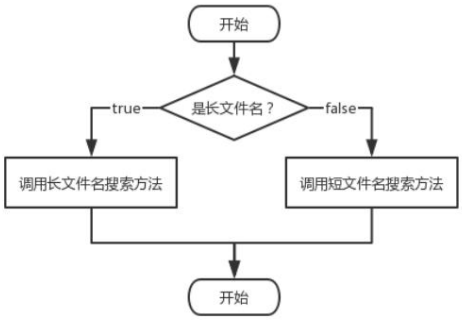
interface	describe
create	Create a file under the relative path of the current directory
find_vfile_bypath	Find files in the current directory, support recursive search
read_at	Read data at the specified offset in the current file
write_at	Write data at the specified offset of the current file, and the cluster allocation will be automatically completed
clear	clear current file
ls	List all files in the current directory
say_info	Get the information of the specified directory item in the current directory
stat	Get information about the current file

The above interfaces are mainly used by the kernel, and some private methods are not listed here.

The following introduces some core algorithms:

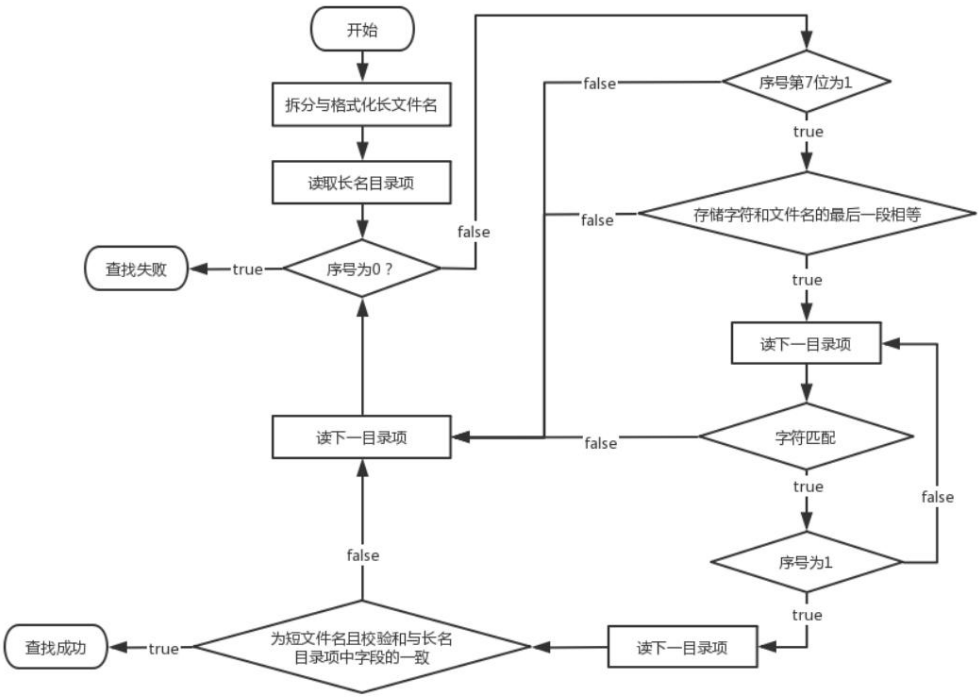
The FAT-like file system is different from EXT in that it is a chain structure rather than an index structure. implement EXT The difficulty is that the logic of indirect indexing is more complicated, but FAT does not have to consider this problem. FAT file system The difficulty is that it distinguishes between long and short file names, and uses directory entries of different structures to store them.

Firstly, the file search algorithm is introduced, the core of which is directory item retrieval. The overall algorithm flow is as follows:



The algorithm first judges the length of the file name, and then calls the corresponding method, because the matching of short file names is relatively simple.

Only the long name matching algorithm is shown here:



Long-named directory entries are stored in the form of "continuous long-named directory entries + corresponding short-named directory entries", for example

abcdefghijklmn.txt, will be stored as follows:

Low address 2: n.txt	
ﾀ	1ÿabcdefghijklm
High address ABCDEF~1 TXT	

Therefore, when matching, it must be matched from the back to the front. To implement this process, we use a stack to store

Divided long file names: After splitting, push them onto the stack in order, and each time a segment is matched, a segment is popped up, so as to realize the reverse order.

Then introduce the file increment algorithm, which is mainly divided into two types: directory and file. In VFile, we implement

The increase_size method is used to allocate enough clusters before writing. For a file, its directory entry has a record size

field, so when incrementing, just calculate the difference between the number of clusters required for the new size and the old size, and then call the management

Layer assignment interface is enough. For directories, FAT32 does not record their size in directory entries, so when calculating,

You need to use the interface provided by FAT to calculate the number of clusters currently in use, and then calculate the number of clusters required based on the new size.

Then call the cluster allocation interface of the management layer according to the difference.

The core ideas of other algorithms are basically the same as those of search algorithms, such as create and ls, which will not be introduced here.

4.3.8 Concurrent access

UltraOS supports dual-core, and the two cores may access the file system at the same time. For correctness of concurrent access, I

They are guaranteed by means of read-write locks. The advantage of using read-write locks is that if two processes with different cores

When you need to read a resource, you can access the resource at the same time, and only mutually exclusive access when writing the resource.

In this file system, there are three locking objects: FAT, file system manager, and cache block.

For FAT, most of its accesses are queries, and it is enough to obtain a read lock. It may only be used when adding, deleting, or modifying files.

Modify FAT, so better concurrency can be achieved. The modification position of FAT can be regarded as random, because it needs

The modified entries are dynamically changed. There is no guarantee of the allocated or reclaimed cluster number, and therefore no guarantee of where the table entry is

data blocks, and atomicity cannot be guaranteed when executing a single transaction. For example, when allocating clusters, it may be necessary to allocate

Multiple clusters, and the entries corresponding to these multiple clusters may be located in different data blocks, so only locking a certain block does not

It can ensure that the entries on other blocks are not tampered with by another process. For this reason, it is necessary to lock the entire FAT.

For the file system manager, it is mainly responsible for organizing the global file system and providing services for other layers.

It itself will not be accessed by writes, so in principle no locking is required. But in order to expand its functionality in the future,

Read-write locks are still added when in use. The cache block lock is the most basic lock of the entire file system, because the file system uses block

For reading and writing units. The block granularity is a compromise, if the granularity is further reduced, such as by directory entry

lock, because of the nature of FAT32's variable-length directory entries, the control mechanism will be extraordinarily complicated, and even affect efficiency;

If the granularity is enlarged, such as locking the entire file system, concurrent access cannot be performed and performance will be reduced. In contrast,

Block-level locks are simple to implement, but also flexible. Block-level locks actually lock the entire file,

Because this file system uses the short name directory entry as the entry to access the file, and accessing the short name directory entry needs to obtain its

A lock on a block is therefore equivalent to locking the entire file.

5 System Test

5.1 Test preparation

The overall test of UltraOS supports two platforms and user program test cases in two application languages.

On the emulator qemu, UltraOS will use the FAT32 file system image as the file system carried by qemu System. On the K210, the SD card is used as the hardware support of the file system, and the corresponding driver control is realized Corresponding read and write functions of SD card.

On the user program, we run the corresponding program written in Rust and C to determine whether the corresponding program is full meet the corresponding requirements. The program includes combinations of various system calls, and tests under different stresses.

The test of the file system is mainly divided into two parts, one part is the test of the file system itself, and the other part is a test for the relevant system call. The file system of UltraOS is loosely coupled with the kernel, so files can be The software system is tested. In order to ensure the reliability of the test, we directly format the SD card according to FAT32, and then Test on the card. In order to improve the test efficiency, we divide the test into three phases: first use the card reader to The SD card is connected to the virtual machine, and the file system is verified separately on the virtual machine; then continue to use the card reader, and the The SD card is used as a virtual disk of QEMU, and the kernel is run on QEMU to verify kernel support and system calls. certificate; finally verify in K210. The difficulty of debugging in the three stages increases in turn, but the first two steps can be found and resolve most issues. In addition, FAT32 is a common format for SD cards, both Windows and Linux support it File system. In order to verify the reliability of our file system, we should also ensure that after operating on SD, other systems The SD card can still be read and written normally, and the operations reserved by this file system can be recognized.

5.2 Test method

Table 5-1 Test methods for system call functions and performance

No. Test object 1. System basic index test		testing method
(1) System calls can be used normally		After the kernel is started, the functional test corresponding to the system call is carried out, and the basic requirements can be met.
2. Functional test (user mode test)		
(1) User program exits		Test the exit of for to see if the return value of pid meets the corresponding requirements, and the exit status of the child process can correspond to the status obtained by the parent
(2) Colorful character test		process. Use various CSI control sequences to see if you can print special characters on interrupts.

(3) Process replication test	High-intensity process copy and exit, observe whether the pid satisfies the corresponding relationship, and call sleep to extend the process life cycle when the process is running.
(4) Matrix calculation (5) Pipeline test	Use matrix calculations to test the correct operation of computational programs. Tests the pipeline for data transfer errors. Make the process continuously yield to other processes to see if the active time slice is successfully surrendered.
(6) Actively give up scheduling test	

Table 5-2 Test methods for file system functions and performance

No. Test object 1. System basic index	Test methods
test	
(1) File system startup test	Format the SD card as FAT32 and partition it, run the file system code, check whether the file system can correctly verify the block device and read in the basic information of FAT32. Format the SD card as FAT32, write content under Linux or Windows, you should
(2) Compatibility test	ensure that the file system can be read. This file system writes content to the SD card, and it should be able to be accessed normally on Linux or Windows.
2. Functional test	
(1) File creation test Create files in the root directory. (2) File read and write test Write random strings into the created file, and then read them out for comparison. (3) The content of the written file, reads it again, and checks whether the read length is 0 (4) The directory creation test creates a directory in the root directory, and creates a multi-level directory in the same directory and perform a random string read and write test	
(6) File search test	Search the directory or file under the multi-level directory, open and check whether the content is correct. Delete the file, recursively delete the directory, and check whether the cluster change after deletion is correct. Use the system call test program provided by the competition to verify
(7) File/directory deletion test	
(8) System call test 3. Performance test	
(1) Large file read and write test	This test is to verify whether the file system operates reliably on large files. The test is divided into 8 rounds. Randomly generate character strings ranging in size from 4 to 5000 sectors, write them into the file, and read them out after writing to check whether they are consistent.

5.3 Test results

Table 5-3 Test results of user program function

No. 1.	Test items	indicators Test results
Functional index test		
(1) User program exits		The return value of pid meets the corresponding requirements, and the exit status of the child process corresponds to the status obtained by the parent process. Characters can be displayed
(2) Colorful character test		in different colors using various CSI control sequences

(3) Process replication test		Show. High-intensity process replication and exit, pid satisfies the corresponding relationship. Using matrix calculation, the matrix result is correct. The data transfer
(4) Matrix calculation (5)		of the test pipeline is correct. Make the process keep giving in to other processes,
Pipeline test (6) Active		success.
yield scheduling test		

Table 5-4 Test results of functional indicators and performance

indicators of the file system	test The file system can be stated	System basic indicator FAT32 file system test (2)
Functional test (1) File creation test		
		There is no abnormality or difference in the use of SD cards in different environments, and the system is compatible
		The file system can correctly create files. The file
(2) File read and write test		system can be read and written correctly. After reading and writing, the file system can be accessed normally under other platforms. The file system can be correctly cleared. The file system can correctly
(3) File clearing test (4) Directory		create directories.
creation test (5) File reading		
and writing test in the directory	The file system can read and write files correctly in multi-level directories	
(6) File search test		The file system can correctly search for existing files, and report an error in time when the file does not exist. The file can be deleted correctly. The number of
(7) File/directory deletion test		clusters available after deletion is the same as before creation, and the deleted file cannot be found on other platforms. Related system call tests, a total of 17
(8) System call test 3.		
Performance test		
(1) Large file read and write test		Through the large file read and write test, a total of 6596 sectors were read, written, and cleared, and it took 6041ms

6 Summary and Outlook

6.1 Work summary

(1) Rust language

(2) Multi-core operating system

(3) Support 59 system calls

(4) High-performance optimization: memory weak consistency optimization, lazy, CoW, file system double file block cache, etc.
mechanism

(5) Signal mechanism: the process supports process signal soft interrupt.

(6) Support C language program and Rust language user program writing and running (provide regression testing basis)

(7) FAT32 virtual file system

(8) Hybrid debugging tool: Monitor (combined with static macro printing and dynamic gdb features)

(9) Detailed project documentation, development process support, and comprehension-friendly code constructs and comments

6.2 Future Outlook

(1) We expect to be able to build UltraOS step by step in addition to internal design documents and design teaching
The operating system implementation document, and the corresponding code is built into the code suitable for the experimental operation, and according to the order
Sections provide different code frameworks.

(2) At present, our SBI uses RustSBI built by Luojia, which makes us build
Among them, I don't know much about the unique problems of the K210 hardware platform, such as hardware initialization and interrupt settings.
The subsequent debugging steps become more and more difficult.

(3) Robustness of operating system multi-core support: For multi-core operating systems, we are in normal use
It has been able to achieve almost no errors, but we still found that in extreme cases or stress tests,
There will still be unexpected effects, which may be worthy of our vigilance, and at the same time make more in-depth improvements to UltraOS,
for more stable operation.

(4) Kernel monitor: At present, most of us can only observe from the user level or the print information of the kernel
Monitor the state of the kernel. We need to monitor according to the kernel monitor, including software and hardware implementations
Kernel process running status, kernel memory status, file system status, etc.

(5) GUI support: Although the memory of the operating system is small, we can use the onboard 16MB SSD
Perform page swapping to increase the size of the memory. In this way, we hope to support the GUI and be able to implement basic

image library.

(6) External facility support: We noticed that the resources in the class have LCD displays and cameras, and have been

There have been projects for the development of peripherals, including external WIFI to achieve Internet communication, etc., which will operate

A key challenge in deriving the system to true utility.