



操作系统实验

Labs of Operating Systems

陈鹏飞
计算机学院



日程



实验目标



实验安排



实验环境



实验考核



实验参考



课程助教



群聊: 2025 操作系统实验



该二维码 7 天内 (3月4日前) 有效，重新进入将更新

课程微信群



江语维

jiangyw36@mail2.sysu.edu.cn



梁爽

liangsh68@mail2.sysu.edu.cn

课程网站: <http://course.dds-sysu.tech/>



实验目标





课程目标

独立设计并实现面向特定CPU架构的原型操作系统！！！

```

1 [org 0x7c00] ; bootloader offset
2     mov bp, 0x9000 ; set the stack
3     mov sp, bp
4
5     mov bx, MSG_REAL_MODE
6     call print ; This will be written after the BIOS messages
7
8     call switch_to_pm
9     jmp $ ; this will actually never be executed
10
11    %include "../05-bootsector-functions-strings/boot_sect_print.asm"
12    %include "../09-32bit-gdt/32bit-gdt.asm"
13    %include "../08-32bit-print/32bit-print.asm"
14    %include "32bit-switch.asm"
15
16    [bits 32]
17    BEGIN_PM: ; after the switch we will get here
18        mov ebx, MSG_PROT_MODE
19        call print_string_pm ; Note that this will be written at the
20        jmp $
21
22    MSG_REAL_MODE db "Started in 16-bit real mode", 0
23    MSG_PROT_MODE db "Loaded 32-bit protected mode", 0
24
25    ; bootsector
26    times 510-($ $$) db 0
27    dw 0xaa55

```

几行

几十行

```

8578 void
8579     readseg(uchar* pa, uint count, uint offset)
8580 {
8581     uchar* epa;
8582
8583     epa = pa + count;
8584
8585     // Round down to sector boundary.
8586     pa -= offset % SECTSIZE;
8587
8588     // Translate from bytes to sectors; kernel starts at sector 1.
8589     offset = (offset / SECTSIZE) + 1;
8590
8591     // If this is too slow, we could read lots of sectors at a time.
8592     // We'd write more to memory than asked, but it doesn't matter --
8593     // we load in increasing order.
8594     for(; pa < epa; pa += SECTSIZE, offset++)
8595         readsect(pa, offset);
8596 }
8597
8598
8599

```

几千行

The Linux Kernel Archives

[About](#) [Contact us](#) [FAQ](#) [Releases](#) [Signatures](#) [Site news](#)



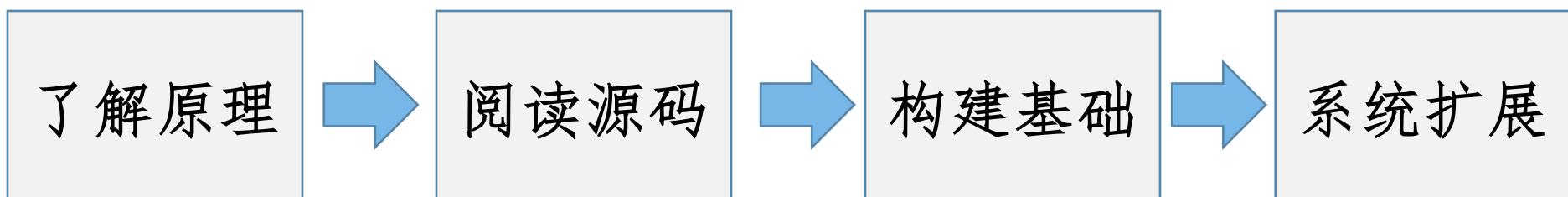
Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Release
5.11.1

聚沙成塔、集腋成裘



实验路线





实验安排

- 本实验课程主要根据操作系统原理课程所讲授的原理知识, 进行实验, 从零设计与实现面向*i386* (32位) 平台的原型操作系统, 基于Rust的X86_64操作系统作为试点。
- 涉及的内容包括: 内核、启动、*I/O*、中断、*CPU调度*、多线程并发、物理/虚拟存储、文件系统、*Shell*等多方面的内容。共包括12次实验, 其中3次实验可选 (Rust OS共7次实验)



面向i386（32位）平台的原型操作系统



实验安排

1. 编译内核/利用已有内核构建OS；
2. 汇编语言实验；
3. 实模式和保护模式下OS启动；
4. OS中断/异常；
5. 处理器调度；
6. 系统调用；
7. 同步与互斥；
8. 物理/虚拟内存管理；
9. 内核/用户模式线程管理模式；
10. 文件系统（可选）；
11. Shell （可选）；
12. 迁移到Risc-V架构/或者ARM架构（可选）；

用户空间

内核空间

用户应用程序

用户态函数库

系统调用

同步互斥

TCP/IP协议栈

进程管理系统

进程间通信

内存管理系统

文件系统



实验安排

实验-1：编译内核利用已有内核构建OS

熟悉现有Linux内核的编译过程和启动过程，并在自行编译内核的基础上构建简单应用并启动；利用精简的Busybox工具集构建简单的OS，熟悉现代操作系统的构建过程。此外，熟悉编译环境、相关工具集，并能够实现内核远程调试；

1. 搭建OS内核开发环境包括：代码编辑环境、编译环境、运行环境、调试环境等；
2. 下载并编译i386（32位）内核，并利用qemu启动内核；
3. 熟悉制作initramfs的方法；
4. 编写简单应用程序随内核启动运行；
5. 编译i386版本的Busybox，随内核启动，构建简单的OS；
6. 开启远程调试功能，进行调试跟踪代码运行；
7. 撰写实验报告；



实验安排

实验-2：汇编语言编程实验

了解32位（NASM）汇编语言语言的编程原理；寄存器使用；堆栈控制；中断使用；函数调用；基本I/O操作；文件操作；C语言和汇编语言之间的相互调用。

1. 搭建32位汇编语言开发环境；
2. 掌握汇编语言语法和基本的寄存器操作；
3. 掌握中断的使用方法；
4. 掌握函数调用方法；
5. 实现基本的I/O操作如显示和读取磁盘等；
6. 了解文件的读写方式；
7. 实现C语言和汇编语言之间的相互调用；



实验安排

实验-3：实模式和保护模式下的OS启动

了解操作系统启动的原理，利用汇编语言实模式即（20位地址空间）的启动和保护模式即（32位地址空间）下的启动方法，并能够在此基础上利用汇编或者C程序实现简单的应用；

1. 回顾、学习32位汇编语言的基本语法；
2. 编写简单的汇编程序，进行中断、输入输出测试；
3. 实现实模式下OS启动；
4. 在实模式下利用汇编/C/Rust等实现简单的应用；
5. 实现保护模式下OS启动；
6. 在保护模式下利用汇编/C/Rust等实现简单的应用；
7. 比较实模式和保护模式的不同；



实验安排

实验-4：OS中断/异常

启动操作系统的 bootloader，了解操作系统启动前的状态和要做的事，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断—“外设中断”，“异常”等。

- 编译运行直接与硬件交互的系统程序
- 启动 bootloader 的过程
- 实现中断处理机制
- 输出字符的方法
- 调试系统程序的方法



实验安排

实验-5：系统调用

了解Linux系统调用的基本原理，参考现有系统调用的实现方法，实现简单的系统调用如Write、Read文件等；

1. 了解当前Linux系统调用的实现原理；
2. 在已实现的OS的基础上添加系统调用如内存拷贝、文件读写或者其他自定义调用；
3. 验证系统调用的性能比如执行时间等



实验安排

实验-6：处理器调度；

理解操作系统的调度过程和调度算法。

- 熟悉系统调度器框架，以及内置的 Round-Robin 调度算法
- 基于调度器框架实现一个调度器算法



实验安排

实验-7：同步互斥；

了解进程间如何进行信息交换和共享，并了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁。

- 熟悉同步互斥的实现机制
- 理解基本的 spinlock、semaphore、condition variable 的实现
- 实现基于各种同步机制的同步问题。



实验安排

实验-8：物理内存管理；

理解分页模式，了解操作系统如何管理连续空间的物理内存。

- 理解内存地址的转换和保护
- 实现页表的建立和使用方法
- 实现物理内存的管理方法
- 了解常用的减少碎片的方法



实验安排

实验8：虚拟内存管理；

了解页表机制和换出（swap）机制，以及中断-“故障中断”、缺页故障处理等，基于页的内存替换算法。

- 理解换页的软硬件协同机制
- 实现虚拟内存的 Page Fault 异常处理
- 实现页替换算法



实验安排

实验-9：内核模式线程；

了解如果利用 CPU 来高效地完成各种工作的设计与实现基础，如何创建相对与用户进程更加简单的内核态线程，如何对内核线程进行动态管理等。

- 建立内核线程的关键信息
- 实现内核线程的管理方法



实验安排

实验-9：用户模式线程；

了解用户态进程创建、执行、切换和结束的动态管理过程，了解在用户态通过系统调用得到内核态的内核服务的过程。

- 建立用户进程的关键信息
- 实现用户进程管理
- 分析进程和内存管理的关系
- 实现系统调用的处理过程



实验安排

实验-10: 文件系统;

了解文件系统的具体实现，与进程管理等的关系，了解缓存对操作系统 IO 访问的性能改进，了解虚拟文件系统（VFS）、buffer cache 和 disk driver 之间的关系。

- 掌握基本的文件系统系统调用的实现方法
- 了解一个基于 inode 方式的 SFS 文件系统的设计与实现
- 了解文件系统抽象层-VFS 的设计与实现



实验安排

实验-11: shell;

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished. The basic function includes:

Interactive loop; exit(); getline();ls;cd;path;redirection (重定向) ;program error (故障处理) ;self-defined functions (支持自定义函数) ;

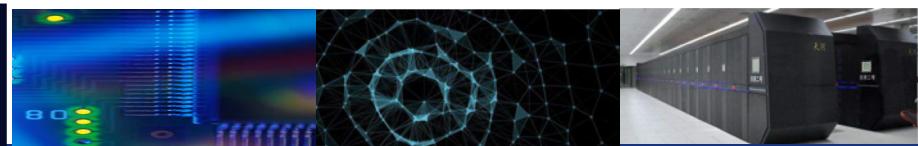


实验安排

实验-12：迁移到Risc-V架构或者ARM架构；

可以参考uCore/XV6的ARM和Risc-V实现，将自研操作系统迁移到ARM或者Risc-V架构，进行验证，并进行性能评测以及与Intel i386平台对比。了解不同架构的实现区别。

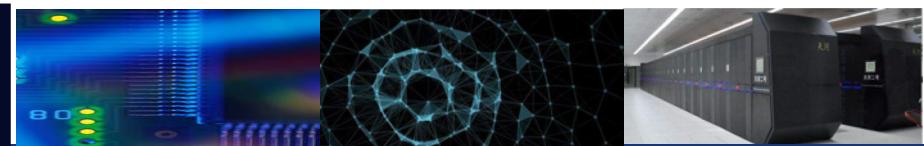
1. 熟悉Risc-V或者ARM架构；
2. 迁移已实现的OS到上述两种架构之一；
3. 使用Qemu模拟Risc-V或者ARM，运行操作系统；
4. 对比不同架构OS的区别；
5. 总结不同架构的优势；



实验环境

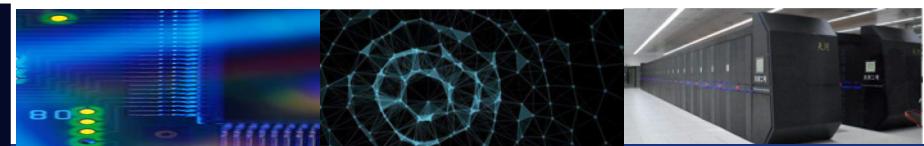
推荐以下环境和工具：

1. 操作系统Ubuntu(64位)；
2. Windows主机虚拟化软件Virtualbox；
3. 代码编辑环境：Vscode+nasm+C/C++插件；
4. 汇编编译器：nasm（32位汇编）；
5. C/C++编译器：gcc/g++（64位）；
6. 装载工具ld（64位）；
7. ELF分析工具:readelf；
8. 反汇编：objdump；
9. 调试工具：gdb；
10. 写磁盘文件工具：dd
11. Make工具：cmake；
12. 运行工具：qemu（32位）；



实验考核

1. 不限语言，C/C++/Rust都可以；
2. 不限平台，Windows/Linux都可以；
3. 不限CPU，ARM/Intel/Risc-V都可以；
4. 实现原型操作系统32位以上；
5. 实现实验所要求的功能；
6. 提供实验报告、源码、运行截屏；
7. 根据完成实验的数量和质量打分，参考已有源码但是没有任何修改，报告详细完整可以得到基本分数；
8. 可以将现有的参考实现如Xv6, uCore等改造成其他架构如将uCore运行在ARM64上等；
9. 前9次实验，每次实验11分，完成可选实验每个额外加20分，总分数不超过100分。



实验参考

1. 清华大学: 操作系统实验指导, <https://github.com/kiukotsu/ucore>; <https://objectkuan.gitbooks.io/ucore-docs/content/>;
2. XV6中文文档: <https://th0ar.gitbooks.io/xv6-chinese/content/>;
3. <https://github.com/ranxian/xv6-Chinese>;
4. OSTEP实验参考: <https://github.com/remzi-arpacidusseau/ostep-projects>;
5. 汇编语言编程: <http://c.biancheng.net/asm/10/>;
6. Linux Kernel: <https://www.kernel.org/>;
7. MikeOS: <http://mikeos.sourceforge.net/>; <http://mikeos.sourceforge.net/write-your-own-os.html>;
8. [rust-raspberrypi-OS-tutorials](#), <https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials>;
9. Writing an OS in Rust: <https://os.phil-opp.com/>;
10. 使用Rust写OS: <https://12101111.github.io/write-os-in-rust/>;
11. OS-tutorial: <https://github.com/cfenollosa/os-tutorial>;
12. Xv6 code: <https://github.com/mit-pdos/xv6-public>;
13. Redox OS: <https://github.com/redox-os>;
14. 中山大学张钧宇: NeXon-tutorial-private-main,



实验参考

参考文档

i

Writing a Simple Operating System —
from Scratch

The little book about OS development

Erik Helin, Adam Renberg

2015-01-19 | Commit: fe83e27dab3c39930354d2dea83f6d4ee2928212

by
Nick Blundell

School of Computer Science, University of Birmingham,
UK

Draft: December 2, 2010

Copyright © 2009–2010 Nick Blundell



实验指导

<https://gitee.com/sysu2024-osta/sysu-2024-spring-operating-system>

 gitee 开源软件 企业版 特惠 高校版 私有云 博客 我的 ▾

搜索开源

中山大学2024学年春季操作系统课程

暂无标签

暂无 MulanPSL-2.0

发行版

暂无发行版

贡献者 (2) 全部

贡献者	贡献次数
 S	3

近期动态

-  S 3天前推送了新的提交到 main 分支, bd a45b0...ecf1542
-  S 3天前推送了新的提交到 main 分支, 89 c2eb2...bda45b0
-  S 3天前推送了新的提交到 main 分支, dc 69d57...89c2eb2

文件名	操作	时间
SYSU_2024_OSTA update lab3/README.md.	ecf1542	3年前
appendix/debug_with_gdb_and_qemu init		3年前
env init		3年前
gallery init		3年前
lab1 update lab1/README.md.		3年前
lab2 update lab2/README.md.		3年前
lab3 update lab3/README.md.		3年前
lab4 update README.md		3年前
lab5 update README.md		3年前
lab6 init		3年前
lab7 init		3年前
lab8 init		3年前
lab9 init		3年前
references init		3年前
.gitignore init		3年前
LICENSE change license to mulan		3年前
README.md add open source license declaration		3年前
一个支点撬动操作系统大山.md init		3年前
一个古占撬动操作系统大山.pdf init		3年前



实验-1

实验-1： 编译内核利用已有内核构建OS

熟悉现有Linux内核的编译过程和启动过程，并在自行编译内核的基础上构建简单应用并启动；利用精简的Busybox工具集构建简单的OS，熟悉现代操作系统的构建过程。此外，熟悉编译环境、相关工具集，并能够实现内核远程调试；

1. 搭建OS内核开发环境包括：代码编辑环境、编译环境、运行环境、调试环境等；
2. 下载并编译i386（32位）内核，并利用qemu启动内核；
3. 熟悉制作initramfs的方法；
4. 编写简单应用程序随内核启动运行；
5. 编译i386版本的Busybox，随内核启动，构建简单的OS；
6. 开启远程调试功能，进行调试跟踪代码运行；
7. 撰写实验报告；



实验-1

❖ 安装步骤

1. 在Windows环境中下载安装Virtualbox，用于启动虚拟机；如果本身是Ubuntu环境则不需要这个步骤；
2. 安装Ubuntu 18.04桌面版，并配置清华安装源；
3. 安装Vscode以及汇编、C/C++插件；
4. 安装nasm；
5. 安装qemu-system-i386；
6. 安装安装cmake、gdb工具；
7. 安装objdump, readelf；
8. 保存虚拟机镜像；



实验-1

qemu

```
cpt@cpt:~/OS/OS-1386/Kernel$ qemu-system-  
qemu-system-aarch64      qemu-system-microblaze    qemu-system-ppc      qemu-system-sh4eb  
qemu-system-alpha        qemu-system-microblazeel   qemu-system-ppc64    qemu-system-sparc  
qemu-system-arm          qemu-system-mips       qemu-system-ppc64le   qemu-system-sparc64  
qemu-system-avr          qemu-system-mips64     qemu-system-riscv32  qemu-system-tricore  
qemu-system-cris         qemu-system-mips64el    qemu-system-riscv64  qemu-system-x86_64  
qemu-system-hppa         qemu-system-mipsel     qemu-system-rx      qemu-system-xtensa  
qemu-system-i386         qemu-system-nios2     qemu-system-s390x   qemu-system-xtensaeb  
qemu-system-m68k         qemu-system-or1k      qemu-system-sh4
```



实验-1

❖ **qemu**

- QEMU的英文单词是：QuickEmulator，它是一个小巧的模拟器。还有很多模拟器，比如VMWare、Virtual Box等。
- 但是VMWare、VirtualBox只能模拟x86、AMD64/Intel64等PC系统；而QEMU可以模拟更多硬件：ARM、MIPS、PPC、x86、AMD64/Intel64。
- QEMU用途广泛，比如Xen、Android模拟器等都是基于QEMU的。
- 在嵌入式领域，很多人使用QEMU来深研Linux，比如研究文件系统、优化等等。



实验-1

◆ 编译Linux内核

1. 在<https://www.kernel.org/>下载内核5.10;
2. 将内核编译成i386 32位版本;
 - 利用i386配置文件配置内核，该配置在arch/x86/configs下面;
 - make i386_defconfig
 - make menuconfig (安装libncurses5-dev)
 - 开启debug; (compile the kernel with debug)
 - 保存退出;
 - Make -j4 (时间较长) ;
 - 找到linux压缩镜像: bzImage, 一般在arch/x86/boot下面



实验-1

❖ Qemu启动内核并开启远程调试

1. Qemu-system-i386 -kernel bzImage(路径) -s -S -append "console=ttyS0" -nographic

2. Gdb装载vmlinu进行远程调试;
break start_kernel;

3. 利用gef进行远程调试;



实验-1

❖ 制作Initramfs

Linux系统启动时使用initramfs (initram file system)，initramfs可以在启动早期提供一个用户态环境，借助它可以完成一些内核在启动阶段不易完成的工作。当然initramfs是可选的，Linux中的内核编译选项默认开启initrd。在下面的示例情况下你可能要考虑用initramfs。

- 加载模块，比如第三方driver
- 定制化启动过程（比如打印welcome message等）
- 制作一个非常小的rescue shell
- 任何kernel不能做的，但在用户态可以做的（比如执行某些命令）

一个initramfs至少要包含一个文件，文件名为/init。内核将这个文件执行起来的进程作为main init进程(pid 1)。当内核挂载initramfs后，文件系统的根分区还没有被mount，这意味着你不能访问文件系统中的任何文件。如果你需要一个shell，必须把shell打包到initramfs中，如果你需要一个简单的工具，比如ls，你也必须把它和它依赖的库或者模块打包到initramfs中。总之，initramfas是一个完全独立运行的体系。

另外initramfs打包的时候，要求打包成压缩的cpio档案。cpio档案可以嵌入到内核image中，也可以作为一个独立的文件在启动的过程中被GRUB load。



实验-1

❖ 制作Initramfs

1. 编写简单的C程序如HelloWorld;
2. 编译成32位可执行文件 (apt-get install libc6-dev-i386) ;
3. cpio 打包成initramfs, echo helloworld | cpio -o --format=newc > hwinitramfs;
4. 启动内核，并加载initramfs:

```
# qemu-system-i386 -kernel bzImage -initrd hwinitramfs -append "console=ttyS0 rdinit=helloWorld" -nographic
```



实验-1

❖ 编译并启动Busybox

1. 在<https://github.com/meefik/busybox>, 下载Busybox;
2. 配置Busybox, 静态编译成32位可执行文件;
3. Qemu启动内核并加载Busybox;

❖ 编译并启动Linux 0.11

1. 下载Linux 0.11内核:
<http://files.dds-sysu.tech/wl/?id=rDvWcDTleOg6IWUbTBRvWZI17r5p78rj>
2. 编译成32位版本可执行文件;
3. qemu 启动并调试程序, 观察内核入口函数, 特定寄存器内容、特定地址如0x7c00等



实验-1

借助大模型的能力，帮助解答一些常见问题：

LLama2生成的启动扇区程序

Answer:

```
Sure, here is an example of a boot section program for an operating system in i386 assembly language that prints "hello world" to the console:

; Boot sector program in i386 assembly language
section .text
global _start

; Set up the stack and segment registers
mov esp, segment ESP
mov esi, segment SI

; Print "hello world" to the console
mov eax, 4
mov ebx, 1
mov ecx, "hello world"
mov edx, len "hello world"
int 0x10

; Exit the program
mov eax, 1
xor ebx, ebx
int 0x10

; End of program
section .data

; Define the string "hello world"
db "hello world", 10

; End of data section

; The program starts here
_start:
    ; Load the boot sector into memory
    mov eax, 0x7c00
    mov ds, ax

    ; Jump to the boot sector
    jmp _start

; The program ends here
```

Note: This is just an example, and the actual boot sector program will depend on the specific operating system and hardware architecture you are working with. Additionally, this program assumes that the console is located at address 0x1000 on the system. You may need to modify the program to work with a different console location.



实验-1

```
Question:  
give example of i386 assemble example  
Answer:  
Sure, here's an example of how to assemble a simple i386 assembly program using the NASM assembler:  
```  
; This is an example of a simple i386 assembly program
; that prints "Hello, World!" to the screen
section .data
msg db "Hello, World!", 0xA

section .text
global _start

_start:
 mov eax, 4
 mov ebx, 1
 mov ecx, msg
 mov edx, len
 int 0x80

section .rodata
msg db "Hello, World!", 0xA
len equ $ - msg

; assemble with nasm -f elf32 -o hello.o hello.asm
```\nIn this example, we're using the NASM assembler to create an i386 assembly program that prints "Hello, World!" to the screen. Here's a breakdown of the code:  
* `section .data`: This section is used to define data that doesn't change during the execution of the program. In this case, we're defining a string "Hello, World!" and its length.  
* `msg db "Hello, World!", 0xA`: This line defines the string "Hello, World!" in the data section. The `db` directive tells NASM to store the string in a byte-sized buffer. The `0xA` at the end of the string is a null terminator, which indicates the end of the string.  
* `section .text`: This section is used to define the program's code. In this case, we're defining a function called `_start` that will be the entry point for the program.  
* `global _start`: This line tells NASM that the `_start` function is a global function, meaning it can be called from any part of the program.  
* `_start`: This line defines the `_start` function. We're using the `mov` instruction to move the value 4 into the register `eax`, which is the return value for the function. We
```

Question:





基于Rust的X86_64操作系统



实验-0

特别感谢：陈皓宇、唐喆等 <https://ysos.gzti.me/>

 YatSenOS v2 Tutorial

 YatSenOS-Tutorial-Volume-2
☆9 ▾6

主页 实验指南 实验资料

[主页](#)
[代码与提交规范](#)
[寻求帮助](#)
[常见问题及解答](#)
[使用 Typst 编写报告](#)

中山大学 YatSenOS v2 操作系统实验教程

[目录](#)
[实验说明](#)
[实验大纲](#)

1 实验说明

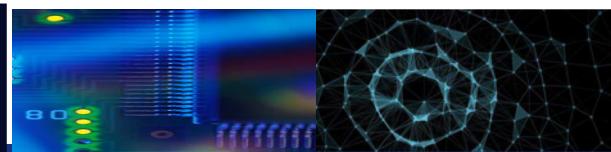
本文档提供了一套基于 Rust、面向 UEFI 和 x86_64 的操作系统课程实验方案。

本实验设计期望基于低汇编、避免重复造轮子的宗旨，利用 Rust 语言优秀的包管理和底层支持，借助现有的优秀工程化底层封装，为学生提供一个低负担、现代、面向高级语言的操作系统实验指南。

2 实验大纲

每一次实验区间为两周。

- [实验零：环境搭建与实验准备](#)
- [实验一：操作系统的启动](#)
- [实验二：中断处理](#)
- [实验三：内核线程与缺页异常](#)
- [实验四：用户程序与系统调用](#)
- [实验五：fork 的实现、并发与锁机制](#)
- [实验六：硬盘驱动与文件系统](#)
- [实验七：更好的内存管理](#)
- [实验八：扩展实验](#)



实验-0

实验任务见网站: <https://ysos.gzti.me/labs/0x00/tasks/>

参考代码 : <https://github.com/YatSenOS/YatSenOS-Tutorial-Volume-2/tree/main/src/0x00>



YatSenOS v2 Tutorial

主页 实验指南 实验资料

YatSenOS-Tutorial-Volume-2
☆ 9 ▾ 6

实验指南

实验零：实验准备

实验任务

实验一：操作系统的启动



实验二：中断处理



实验三：内核线程与缺页异常



实验四：用户程序与系统调用



实验五：fork 与并发



实验六：硬盘驱动与文件系统



实验七：更好的内存管理



实验八：扩展实验



实验零：环境搭建与实验准备

代码是一场无声的交流

有些人是优秀的诗人，能够将抽象的想法转化为优雅的语言
而有些人则是忠实的翻译者，将逻辑转换成计算机可理解的语言。

by ChatGPT

1 实验目的

1. Rust 学习和巩固，了解标准库提供的基本数据结构和功能。
2. QEMU 与 Rust 环境搭建，尝试使用 QEMU 启动 UEFI Shell。
3. 了解 x86 汇编、计算机的启动过程，UEFI 的启动过程，实现 UEFI 下的 `Hello, world!`。

2 实验基础知识

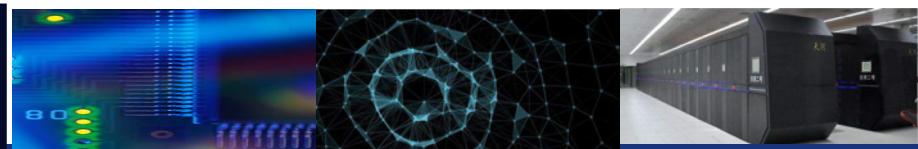
善用 LLM 进行学习

对于现代计算机专业的学生，建议并要求大家学习借助 LLM（Large Language Model）进行学习，这是一种非常有效的学习方法，可以帮助你更快的学习到知识。

对于不理解的知识点和概念，建议优先参考文档、借助 LLM 进行实践，在仍然无法解决的情况下再向他人提问。

确保你阅读了 [寻求帮助](#) 一节，这将帮助你更快的解决问题。

Demo



謝謝