



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验报告

实验一：操作系统的启动

姓 名： 刘家祥
学 号： 23336152
教学班号： 计科二班
专 业： 计算机科学与技术
院 系： 计算机学院

2024~2025 学年第二学期

操作系统的启动

一. 编译内核 ELF

首先在 ysos 下创建一个 0x01 的目录，将实验 0x00 文件中的 assets 文件夹复制到 0x01 目录下，接下来在 pkg/kernel 目录下执行 `cargo build --release` 命令，得到如下错误：

```
camellia@LAPTOP-Camellia: ~ × + v
Compiling uefi v0.34.1
Compiling ysos_boot v0.1.0 (/home/camellia/ysos/0x01/pkg/boot)
Compiling ysos_kernel v0.1.0 (/home/camellia/ysos/0x01/pkg/kernel)
error[E0425]: cannot find value `SERIAL` in this scope
--> pkg/kernel/src/utils/macros.rs:70:14
70 |         unsafe { SERIAL.get().unwrap().force_unlock() };
    |                   ^^^^^^ not found in this scope
help: consider importing this static
1  + use crate::drivers::serial::SERIAL;
    |
warning: unused variable: `record`
--> pkg/kernel/src/utils/logger.rs:19:19
19 |         fn log(&self, record: &Record) {
    |                   ^^^^^^ help: if this is intentional, prefix it with an underscore: `_record`
    = note: `#[warn(unused_variables)]` on by default
warning: unused variable: `port`
--> pkg/kernel/src/drivers/uart16550.rs:7:22
7 |         pub const fn new(port: u16) -> Self {
    |                   ^^^^^ help: if this is intentional, prefix it with an underscore: `_port`
warning: unused variable: `data`
--> pkg/kernel/src/drivers/uart16550.rs:17:28
```

根据报错信息提示，在 macros.rs 文件开头加入 `use crate::drivers::serial::SERIAL;`，再次执行 `cargo build --release` 命令，解决了之前的 error 问题，编译成功，得到如下结果：

```

|          ^^^^^^ help: if this is intentional, prefix it with an underscore: `'_record`
= note: `#[warn(unused_variables)]` on by default
warning: unused variable: `port`
--> pkg/kernel/src/drivers/uart16550.rs:7:22
7 |         pub const fn new(port: u16) -> Self {
|         ^^^^^^ help: if this is intentional, prefix it with an underscore: `'_port`
warning: unused variable: `data`
--> pkg/kernel/src/drivers/uart16550.rs:17:28
17 |         pub fn send(&mut self, data: u8) {
|         ^^^^^^ help: if this is intentional, prefix it with an underscore: `'_data`
warning: method `receive` is never used
--> pkg/kernel/src/drivers/uart16550.rs:22:12
6 |     impl SerialPort {
|     ----- method in this implementation
...
22 |         pub fn receive(&mut self) -> Option<u8> {
|         ^^^^^^
= note: `#[warn(dead_code)]` on by default
warning: `ysos_kernel` (lib) generated 4 warnings
Finished `release` profile [optimized] target(s) in 0.93s

```

随后执行命令：

```
find ~/ysos/0x01/target/x86_64-unknown-none/release/ -type f -name "*.elf" -o -
name "libysos_kernel.a" -o -name "ysos_kernel"
```

```

camellia@LAPTOP-Camellia:~/ysos/0x01/pkg/kernel$ find ~/ysos/0x01/target/x86_64-unknown-none/release/ -type f -name "*.e
lf" -o -name "libysos_kernel.a" -o -name "ysos_kernel"
/home/camellia/ysos/0x01/target/x86_64-unknown-none/release/ysos_kernel

```

找到编译产物 `ysos_kernel` 并执行 `readelf -h` 命令，得到如下结果：

```
readelf -h /home/camellia/ysos/0x01/target/x86_64-unknown-none/release/
ysos_kernel
```

```

camellia@LAPTOP-Camellia:~/ysos/0x01/pkg/kernel$ readelf -h /home/camellia/ysos/0x01/target/x86_64-unknown-none/release/
ysos_kernel
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                               2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                             UNIX - System V
  ABI Version:                        0
  Type:                               EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:                 0xffffffff0000020d0
  Start of program headers:            64 (bytes into file)
  Start of section headers:            51296 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           7
  Size of section headers:             64 (bytes)
  Number of section headers:           12
  Section header string table index:   10

```

1. 请查看编译产物的架构相关信息，与配置文件中的描述是否一致？

根据 `readelf -h` 指令得到的信息，可以看出编译产物是 X86_64 架构，64 位，小端序，与配置文件中的描述一致。

2. 找出内核的入口点，它是被如何控制的？结合源码、链接、加载的过程，谈谈你的理解。

从 `readelf -h` 指令得到的信息可看出，内核的入口地址为 `0xffffffff00000020d0`。JSON 配置文件、`kernel.ld` 链接脚本和 `src/main.rs` 源码文件共同控制了内核的入口点。JSON 配置文件中强制链接器使用 `kernel.ld` 脚本 `-Tpkg/kernel/config/kernel.ld` 确保入口点和内存布局按脚本进行。链接器使用 `kernel.ld` 脚本，脚本中定义了内核的入口点(`ENTRY(_start)`)明确了入口函数符号为 `_start`，又使用了 `KERNEL_BEGIN = 0xffffffff0000000000` 定义了内核的虚拟地址起始位置。接着在 `main.rs` 中使用了 `boot::entry_point!(kernel_main)` 宏，通过在 `_start` 函数下定义了 `call kernel_main` 来跳转到 rust 入口函数，作为程序的入口点。最后编译得到的 ELF 文件中，可以找到 `Entry point address: 0xffffffff0000000000 + _start`，在这里是 `0xffffffff00000020d0`。

3. 请找出编译产物的 segments 的数量，并且用表格的形式说明每一个 segments 的权限、是否对齐等信息。

```
camellia@LAPTOP-Camellia:~/ysos/0x01/pkg/kernel$ readelf -l /home/camellia/ysos/0x01/target/x86_64-unknown-none/release/ysos_kernel
Elf file type is EXEC (Executable file)
Entry point 0xffffffff00000020d0
There are 7 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
FileSiz        MemSiz             Flags             Align
LOAD           0x0000000000000100 0xffffffff00000000 0xffffffff00000000
0x0000000000000126 0x0000000000000126 R                  0x1000
LOAD           0x0000000000000300 0xffffffff0000002000 0xffffffff0000002000
0x000000000000036b 0x000000000000036b R E               0x1000
LOAD           0x0000000000000700 0xffffffff0000006000 0xffffffff0000006000
0x00000000000001018 0x00000000000001018 RW                0x1000
LOAD           0x0000000000000900 0xffffffff0000008000 0xffffffff0000008000
0x0000000000000000 0x0000000000000020 RW                0x1000
GNU_RELRO     0x0000000000000800 0xffffffff0000007000 0xffffffff0000007000
0x0000000000000018 0x0000000000000018 R                  0x1
GNU_EH_FRAME  0x0000000000000244 0xffffffff0000001244 0xffffffff0000001244
0x000000000000000c 0x000000000000000c R                  0x4
GNU_STACK     0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 RW                0x0

Section to Segment mapping:
Segment Sections...
00 .rodata .eh_frame_hdr .eh_frame
01 .text
02 .data .got
03 .bss
04 .got
05 .eh_frame_hdr
06
```

从 `readelf -l` 指令的信息中可以看出，编译产物有[7 个]Segments,信息如下：

段类型	虚拟地址	权限	对齐
LOAD	0xffffffff0000000000	只读 (R)	0x1000 (4KB)
LOAD	0xffffffff0000002000	可读可执行 (R E)	0x1000 (4KB)
LOAD	0xffffffff0000006000	可读可写 (RW)	0x1000 (4KB)
LOAD	0xffffffff0000008000	可读可写 (RW)	0x1000 (4KB)
GNU_RELRO	0xffffffff0000007000	只读 (R)	0x1 (1 字节)
GNU_EH_FRAME	0xffffffff0000001244	只读 (R)	0x4 (4 字节)
GNU_STACK	0x0	可读可写 (RW)	0x0 (无对齐)

表 1 编译产物的 Segments 分析 - 权限和对齐信息

二. 在 UEFI 中加载内核

1. 加载相关文件

1.1. 加载配置文件

加载配置文件，解析其中的内核栈大小、内核栈地址等内容。

```
// 1. Load config
let config = {
    // 加载配置文件并解析
    let mut file = open_file(CONFIG_PATH);
    let buf = load_file(&mut file);
    config::Config::parse(buf)
};
```

1.2. 加载内核 ELF

根据配置文件中的信息，加载内核 ELF 文件到内存中，并关将其加载为 `ElfFile` 以便进行后续的操作。

```
// 2. Load ELF files
let elf = {
    // 从配置中获取内核路径并加载ELF文件
    let mut file = open_file(config.kernel_path);
    let buf = load_file(&mut file);
    ElfFile::new(buf).expect("Failed to parse ELF")
};
```

2. 更新控制寄存器以禁用写保护功能

```
// FIXME: root page table is readonly, disable write protect (Cr0)
unsafe {
    Cr0::update(|flags| {
        flags.remove(Cr0Flags::WRITE_PROTECT);
    });
}
```

3. 映射内核文件

在成功加载内核，并禁用根页表写保护后，需要将内核的代码段、数据段、BSS 段等映射到虚拟地址空间中。

```
// FIXME: map physical memory to specific virtual address offset
map_physical_memory(
    config.physical_memory_offset,
    max_phys_addr,
    &mut page_table,
    &mut UEFIFrameAllocator,
);
// FIXME: load and map the kernel elf file
load_elf(
    &elf,
    config.physical_memory_offset,
    &mut page_table,
    &mut UEFIFrameAllocator,
).expect("Failed to load kernel ELF");

// FIXME: map kernel stack
let stack_pages = map_range(
    config.kernel_stack_address,
    config.kernel_stack_size,
    &mut page_table,
    &mut UEFIFrameAllocator,
).expect("Failed to map kernel stack");
```

pkg/elf/src/lib.rs 中的 load_segment 函数补全。

```

// 处理段标志并设置相应的页表标志
if segment.flags().is_write() {
    page_table_flags |= PageTableFlags::WRITABLE;
}

// 如果段不是可执行的，设置不可执行标志（NX位）
if !segment.flags().is_execute() {
    page_table_flags |= PageTableFlags::NO_EXECUTE;
}

// 如果段是用户空间的代码/数据，设置用户访问权限
if segment.flags().is_read() {
    page_table_flags |= PageTableFlags::USER_ACCESSIBLE;
}

```

4. 调试内核

首先使用 `python3 ysos.py build -p debug` 命令编译内核，然后使用 `python3 ysos.py launch -d` 命令启动 QEMU 并进入调试模式，在 0x01 目录下创建一个 `.gdbinit` 文件，内容如下：

```

file esp/KERNEL.ELF
gef-remote localhost 1234
tmux-setup
b ysos_kernel::init

```

打开另一个终端，在 0x01 目录下执行 `gdb -q` 命令进入 GDB 调试模式，接着使用 `c` 命令继续执行，得到如下输出：

```

camellia@LAPTOP-Camellia: ~ X + v
[ Legend: Modified register | Code | Heap | Stack | String ]

registers -----
$rax : 0xffffffff0000020d0 → <_start+0000> push rax
$rbx : 0x0000000005f0df78 → 0x0000000005c01000 → 0x0000000005c02023 → 0xc080030000000005
$rcx : 0x0000000005f0dfd0 → 0x00000000059ee018 → "IBI SYSTF"
$rdx : 0xffffffff01001ffff8 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$rsp : 0xffffffff01001ffed8 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$rbp : 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$rsi : 0x0000000005f0dec8 → 0x00000000043a1018 → 0x0000000000000003 → 0x0000000000000000 → 0x0000
000000000000 → [loop detected]
$rdi : 0x0000000005f0dfd0 → 0x00000000059ee018 → "IBI SYSTF"
$rip : 0xffffffff000002615 → <_sysos_kernel::init+0005> mov rdi, QWORD PTR [rdi]
$r8 : 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r9 : 0x0000000000000501 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r10 : 0x0000000005ae8860 → 0x7707309600000000
$r11 : 0x0000000000000870 → 0x0000000000000000 → 0x0000000000000000 → [loop detected]
$r12 : 0xffffffff1ffaaffc0
$r13 : 0x0000000004e7abe0 → 0x11d293ca8be4df61
$r14 : 0x00000000043b048 → 0x0000000000000003 → 0x0000000000000000 → 0x0000000000000000 → [loop
detected]
$r15 : 0x00000000043b2ec8 → 0x00000000043a4940 → 0xc000000004e3be9
$eflags: [zero carry PARITY adjust SIGN trap interrupt direction overflow resume virtualx86 identification]
$cs: 0x38 $ss: 0x30 $ds: 0x30 $es: 0x30 $fs: 0x30 $gs: 0x30

--- stack ---
0xffffffff01001ffed8 +0x0000: 0x0000000000000000 → 0x0000000000000000 → [loop detected] ← $rsp
0xffffffff01001ffed0 +0x0008: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001ffed8 +0x0010: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001ffef0 +0x0018: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001ffef8 +0x0020: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff00 +0x0028: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff08 +0x0030: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff01001fff10 +0x0038: 0x0000000000000000 → 0x0000000000000000 → [loop detected]

e:x86:64 -----
0xffffffff00000260f int3

```

```

camellia@LAPTOP-Camellia: ~ X + v
0xffffffff00000260f int3
0xffffffff000002610 <_sysos_kernel::init+0000> push rbx
0xffffffff000002611 <_sysos_kernel::init+0001> sub rsp, 0x70
→ 0xffffffff000002615 <_sysos_kernel::init+0005> mov rdi, QWORD PTR [rdi]
0xffffffff000002618 <_sysos_kernel::init+0008> call 0xffffffff0000028c0 <_ZN4uefi5table16set_system_table1
7h70e263032aae6e7aE>
0xffffffff00000261d <_sysos_kernel::init+000d> movzx eax, BYTE PTR [rip+0x59dc] # 0xffffffff00000800
0 <_ZN11sysos_kernel7drivers6serial6SERIAL17h1a01fe5ce975ebfeE>
0xffffffff000002624 <_sysos_kernel::init+0014> cmp al, 0x2
0xffffffff000002626 <_sysos_kernel::init+0016> jne 0xffffffff00000283e <_ZN11sysos_kernel4init17hd9199288d
cfcaf6fE+558>
0xffffffff00000262c <_sysos_kernel::init+001c> call 0xffffffff000002880 <_ZN11sysos_kernel7drivers6serial19
get_serial_for_sure17h543de7a63bbf1899E>
source:pkg/kernel/src/l
ib.rs+22 -----
17 use uefi::{runtime::ResetType, Status};
18
19 pub fn init(boot_info: &'static BootInfo) {
20     unsafe {
21         // set uefi system table
→ 22         uefi::table::set_system_table(boot_info.system_table.cast().as_ptr());
23     }
24
25     drivers::serial::init(); // init serial output
26     logger::init(); // init logger system
27

threads -----
[#0] Id 1, stopped 0xffffffff000002615 in _sysos_kernel::init (), reason: BREAKPOINT

--- trace ---
[#0] 0xffffffff000002615 → _sysos_kernel::init(boot_info=0x5f0dfd0)
[#1] 0xffffffff000002013 → _sysos_kernel::kernel_main(boot_info=0x5f0dfd0)
[#2] 0xffffffff0000020d6 → _sysos_kernel::__impl_start(boot_info=0x5f0dfd0)

```

注意：这里使用 gdb15.0 会出现一些非必要的错误信息，又因 Ubuntu24.04 无 gdb12.1 安装包，遂使用下载源码的方式更换为 gdb12.1

4.1. 汇编分析

从测试会话中，可以看出断点在 `ysos_kernel::init` 函数的开始处已经正确命中。当前断点处的汇编代码显示：

→ `0xffffffff0000002615 <ysos_kernel::init+0005> mov rdi, QWORD PTR [rdi]`，这表明程序停在了 `ysos_kernel::init` 函数的第 5 个字节处。正在执行一条 `mov` 指令，将 `rdi` 寄存器中的值加载到 `rdi` 寄存器中。这是在解引用 `boot_info` 参数，访问其中的系统表。前后的相关指令为：

```
0xffffffff0000002610 <ysos_kernel::init+0000> push    rbx
0xffffffff0000002611 <ysos_kernel::init+0001> sub     rsp, 0x70
0xffffffff0000002615 <ysos_kernel::init+0005> mov     rdi, QWORD PTR [rdi]
0xffffffff0000002618 <ysos_kernel::init+0008> call   0xffffffff00000028c0
```

这是函数序言部分（prologue），设置栈帧并准备访问参数。

4.2. 符号信息分析

根据测试会话中的下列内容：

```
— trace —
[#0] 0xffffffff0000002615 → ysos_kernel::init(boot_info=0x5f0dfd0)
[#1] 0xffffffff0000002013 → ysos_kernel::kernel_main(boot_info=0x5f0dfd0)
[#2] 0xffffffff00000020d6 → ysos_kernel::__impl_start(boot_info=0x5f0dfd0)
```

显示符号信息正确加载：函数名正确解析：`ysos_kernel::init`、`ysos_kernel::kernel_main` 和 `ysos_kernel::__impl_start`，参数名和值可见：`boot_info=0x5f0dfd0` 并且源代码位置正确显示：`source:pkg/kernel/src/lib.rs+22`，这表明符号信息已正确加载并能与源代码对应。

4.3. 使用 vmmap 查看内存映射

```
gef> vmmap
[ Legend: Code | Stack | Heap ]
Start      End      Offset    Perm Path
0x0000000000000000 0x00000000004c0000 0x00000000004c0000 rw-
0x00000000004c0000 0x00000000004e0000 0x0000000000200000 r--
0x00000000004e0000 0x00000000005c0000 0x0000000000e00000 rw-
0x00000000005c0000 0x00000000005e0000 0x0000000000200000 r--
0x00000000005e0000 0x0000000100000000 0x00000000ffa20000 rw-
0xfffff80000000000 0xfffff80010020000 0x0000000010020000 rw-
0xfffffff000000000 0xfffffff0000006000 0x0000000000006000 r--
0xfffffff0000006000 0xfffffff0000009000 0x0000000000003000 rw-
0xfffffff010000000 0xfffffff010020000 0x0000000000200000 rw-
```

代码段 (r-): 0xffffffff0000000000 - 0xffffffff00000006000 数据段 (rw-): 0xffffffff00000006000 - 0xffffffff00000009000 说明内核加载在高地址空间，内核时一个“higher half kernel”。内核栈区域: 0xffffffff0100000000 - 0xffffffff0100200000 (rw-) 从调试会话看到 \$rsp = 0xffffffff01001ffed8，确认栈指针位于此区域内的高地址(栈向下增长) 物理内存映射: 包含多个区域如 0x0000000000000000 - 0x000000010000000000 和 0xffff80000000000000 - 0xffff80010020000000 这些可能是通过 map_physical_memory 函数映射的物理内存区域。

4.4. 使用 readelf 查看 ELF 文件

```
camellia@LAPTOP-Camellia:~/ysos/0x01$ readelf -a esp/KERNEL.ELF
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:           ELF64
  Data:            2's complement, little endian
  Version:         1 (current)
  OS/ABI:          UNIX - System V
  ABI Version:     0
  Type:            EXEC (Executable file)
  Machine:         Advanced Micro Devices X86-64
  Version:         0x1
  Entry point address: 0xffffffff00000020d0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 102176 (bytes into file)
  Flags:           0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 7
  Size of section headers: 64 (bytes)
  Number of section headers: 20
  Section header string table index: 18

Section Headers:
 [Nr] Name              Type              Address            Offset
     Size              EntSize          Flags  Link  Info  Align
 [ 0]                      NULL              0000000000000000  00000000
     0000000000000000  0000000000000000          0     0     0
 [ 1] .rodata              PROGBITS          ffffffff0000000000 00001000
```

可以看出 ELF 文件的入口地址为 0xffffffff00000020d0，与之前的分析一致，与符号表中的 _start 函数对应。 ffffffff00000020d0 6 FUNC GLOBAL DEFAULT 4 _start。以及段加载信息。

5. 实验任务问题分析

5.1. set_entry 函数做了什么？为什么它是 unsafe 的？

set_entry 函数的目的是设置内核的入口点地址：

```
pub fn set_entry(entry: usize) {
    unsafe {
        ENTRY = entry;
    }
}
```

这个函数是 `unsafe` 的，因为它直接修改了一个全局变量，而这个全局变量是一个静态变量，可能会被其他线程访问，因此需要保证线程安全。

5.2. `jump_to_entry` 函数做了什么？要传递给内核的参数位于哪里？查询 `call` 指令的行为和 x86_64 架构的调用约定，借助调试器进行说明。

`jump_to_entry` 函数负责从引导加载程序跳转到内核的入口点：

```
pub fn jump_to_entry(bootinfo: *const BootInfo, stacktop: u64) → ! {
    unsafe {
        assert!(ENTRY ≠ 0, "ENTRY is not set");
        core::arch::asm!("mov rsp, {}; call {}", in(reg) stacktop, in(reg) ENTRY,
in("rdi") bootinfo);
    }
    unreachable!()
}
```

这个函数设置了新的栈指针 `rsp` 为参数 `stacktop`，然后调用内核的入口点 `ENTRY`，并将 `bootinfo` 作为参数传递给内核。参数位于 `rdi` 寄存器中，根据 x86_64 架构的调用约定，前 6 个参数分别存放在 `rdi`、`rsi`、`rdx`、`rcx`、`r8`、`r9` 寄存器中，超过 6 个参数的参数会被压入栈中。在调试器中，可以看到 `call` 指令的行为，它会将当前的 `rip` 压入栈中，然后跳转到指定的地址。

5.3. `entry_point!` 宏做了什么？内核为什么需要使用它声明自己的入口点？

`entry_point!` 宏用于声明内核的入口点，它会生成一个 `_start` 函数，这个函数会调用内核的入口函数 `kernel_main`，并传递 `BootInfo` 作为参数。内核需要使用这个宏声明自己的入口点，因为内核需要在启动时进行一些初始化工作，比如设置栈帧、初始化内存分配器等，这些工作需要在内核的入口点之前完成。

5.4. 如何为内核提供直接访问物理内存的能力？你知道几种方式？代码中所采用的是哪一种？

为内核提供直接访问物理内存的能力有几种常见方式：

- 恒等映射 (Identity Mapping)：物理地址直接映射到相同的虚拟地址
- 偏移映射 (Offset Mapping)：所有物理地址都映射到虚拟地址空间的特定偏移处
- 按需映射：根据需要动态地映射物理内存区域
- 硬件特定机制：如 x86 的 PAT (Page Attribute Table) 或 MTRR (Memory Type Range Register)

本代码采用的是偏移映射方式，从 `map_physical_memory` 函数可以看出：

```
pub fn map_physical_memory(
    offset: u64,
    max_addr: u64,
    page_table: &mut impl Mapper<Size2MiB>,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
)
```

此函数将物理地址范围 `[0, max_addr)` 映射到虚拟地址空间的 `[offset, offset + max_addr)`。具体在配置中，`physical_memory_offset` 默认值为 `0xFFFF_8000_0000_0000`，这是高地址空间的一部分，避免与内核和用户空间代码冲突。通过这种映射方式，内核可以通过简单地将物理地址加上偏移量来访问任何物理内存位置。

5.5. 为什么 ELF 文件中不描述栈的相关内容？栈是如何被初始化的？它可以被任意放置吗？

5.5.1. ELF 文件中不描述栈的相关内容，因为：

ELF 主要描述程序的代码和数据段，这些是预加载并映射到内存中的。栈是执行环境的一部分，而不是程序本身的一部分。栈的位置和大小通常由加载器或操作系统决定。在不同执行环境中，栈的位置和大小可能需要不同的配置。

5.5.2. 在本代码中，栈是通过以下步骤初始化的：

在 `config.rs` 中定义了栈的配置：

```
kernel_stack_address: 0xFFFF_FF01_0000_0000,
kernel_stack_size: 512, // 页数
```

在 `main.rs` 中，使用 `map_range` 函数为栈分配物理内存并映射到虚拟地址：

```
let stack_pages = map_range(
    config.kernel_stack_address,
    config.kernel_stack_size,
    &mut page_table,
    &mut UEFIFrameAllocator,
)
```

在跳转到内核时，通过以下代码设置栈顶指针：

```
let stacktop = config.kernel_stack_address + config.kernel_stack_size * 0x1000 - 8;
jump_to_entry(&bootinfo, stacktop);
```

5.5.3. 栈不能被任意放置，因为：

栈需要在内核可访问的虚拟地址空间中 栈地址通常需要适当对齐（如 8 字节对齐）栈应避免与其他内存区域（如内核代码、数据段、堆等）重叠 在某些架构上，栈可能有特定的地址要求或限制。

5.6. 请解释指令 `layout asm` 的功能。倘若想找到当前运行内核所对应的 Rust 源码，应该使用什么 GDB 指令？

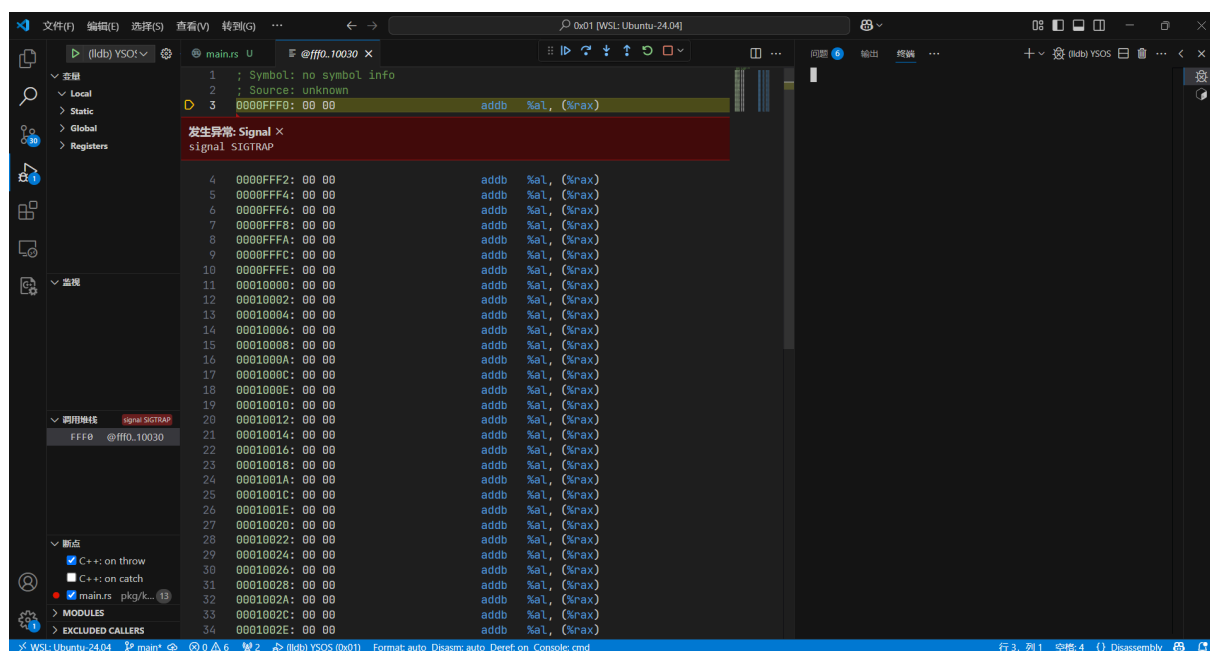
`layout asm` 指令用于在 GDB 中显示当前正在执行的代码的汇编指令，可以帮助调试者理解程序的执行流程。如果想找到当前运行内核所对应的 Rust 源码，可以使用 `list` 指令，它会显示当前正在执行的代码的 Rust 源码。

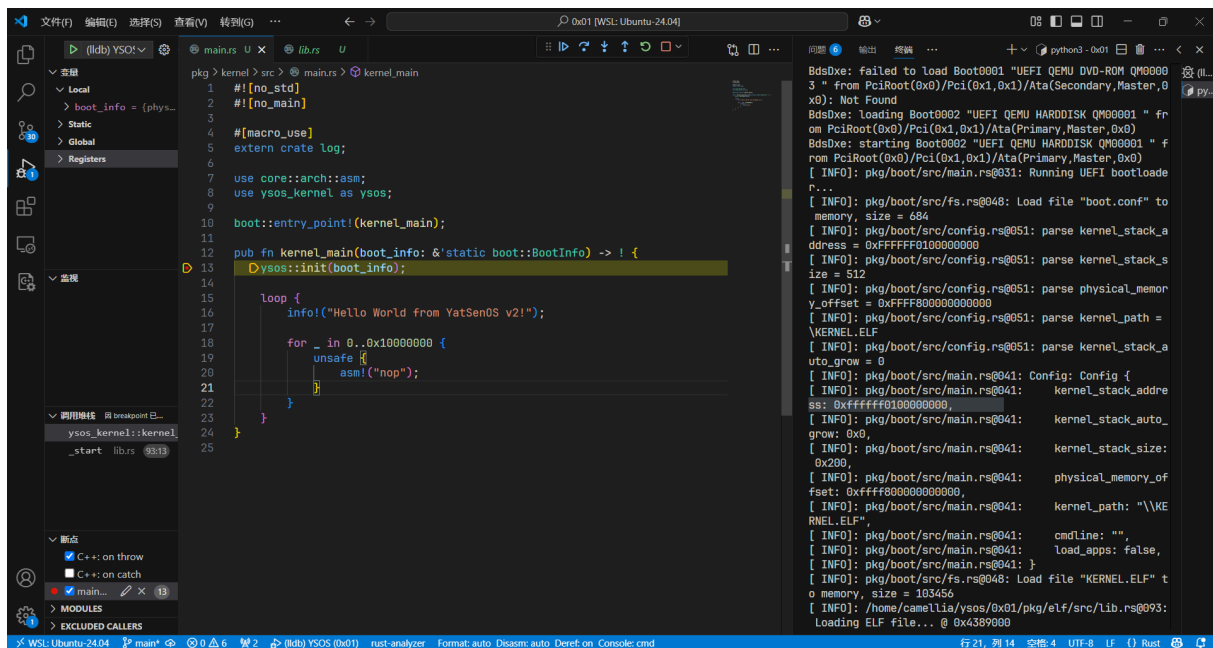
5.7. 没有 `DBG_INFO=true` 的影响

没有调试信息将导致无法查看源代码与机器码的对应关系，无法显示变量名、函数名、文件名和行号，调试将局限于汇编级别，无法设置基于源代码行号的断点。

5.8. 你如何选择了你的调试环境？截图说明你在调试界面（TUI 或 GUI）上可以获取到哪些信息？

我选择了 VSCode + LLDB 的调试环境，截图如下：





在调试界面上，可以获取到源代码、汇编代码、寄存器值、变量值、断点信息等信息。

三. UCRT 与日志输出

1. 串口驱动

1.1. 初始化

```
/// Initializes the serial port.
pub fn init(&self) {
    // 禁用所有中断
    let mut interrupt_enable = PortWriteOnly::new(BASE_ADDR + 1);
    unsafe {
        interrupt_enable.write(0x00_u8);
    }
    // 启用 DLAB (设置波特率除数)
    let mut line_control = PortWriteOnly::new(BASE_ADDR + 3);
    unsafe {
        line_control.write(0x80_u8);
    }
}
```

```

// 设置除数为 3（低字节）38400 波特率
let mut divisor_lo = PortWriteOnly::new(BASE_ADDR + 0);
unsafe {
    divisor_lo.write(0x03_u8);
}

// 高字节设置为 0
let mut divisor_hi = PortWriteOnly::new(BASE_ADDR + 1);
unsafe {
    divisor_hi.write(0x00_u8);
}

```

注意： 这里代码块全部放在同一页会导致排版混乱，无法正常显示，因而分开显示。

```

// 8 位数据，无奇偶校验，一个停止位
unsafe {
    line_control.write(0x03_u8);
}
// 启用 FIFO，清空，使用 14 字节阈值
let mut fifo_control = PortWriteOnly::new(BASE_ADDR + 2);
unsafe {
    fifo_control.write(0xC7_u8);
}
// 启用 IRQ，设置 RTS/DSR
let mut modem_control = PortWriteOnly::new(BASE_ADDR + 4);
unsafe {
    modem_control.write(0x0B_u8);
}
// 设置回环模式，测试串行芯片
unsafe {
    modem_control.write(0x1E_u8);
}
// 测试串行芯片（发送 0xAE 字节并检查返回值）
let mut data = Port::new(BASE_ADDR);
unsafe {
    data.write(0xAE_u8);
}

```

```

// 检查串口是否有故障（返回字节是否与发送字节相同）
unsafe {
    if data.read() != 0xAE_u8 {
        panic!("Serial port initialization failed");
    }
}

// 设置为正常操作模式（非回环模式，IRQ 启用，OUT#1 和 OUT#2 位启用）
unsafe {
    modem_control.write(0x0F_u8);
}
}

```

1.2. 发送数据

```

/// Sends a byte on the serial port.
pub fn send(&mut self, data: u8) {
    // 等待发送缓冲区为空
    let mut line_status = PortReadOnly::<u8>::new(BASE_ADDR + 5);
    // 检查 Line Status Register 的 Transmitter Holding Register Empty 位(第5
位)

    while unsafe { (line_status.read() & 0x20) == 0 } {
        // 忙等待直到发送缓冲区为空
    }

    // 发送字节
    let mut data_port = Port::new(BASE_ADDR);
    unsafe {
        data_port.write(data);
    }
}
}

```

1.3. 接收数据

```

/// Receives a byte on the serial port no wait.
pub fn receive(&mut self) → Option<u8> {
    let mut line_status = PortReadOnly::<u8>::new(BASE_ADDR + 5);
    // 检查 Line Status Register 的 Data Ready 位(第0位)

```



```

    if unsafe { (line_status.read() & 0x01) == 0 } {
        // 无数据可读
        None
    } else {
        // 有数据可读，读取数据
        let mut data_port = Port::new(BASE_ADDR);
        unsafe {
            Some(data_port.read())
        }
    }
}
}
}

```

1.4. 测试串口初始化

首先使用 `rm -rf target` 命令清除之前的编译产物，然后使用 `python3 ysos.py build run` 命令编译内核并运行，得到如下结果：

```

[ INFO]: pkg/boot/src/main.rs@041: physical_memory_of
fset: 0xffff800000000000,
[ INFO]: pkg/boot/src/main.rs@041: kernel_path: "\\KE
RNEL.ELF",
[ INFO]: pkg/boot/src/main.rs@041: cmdline: "",
[ INFO]: pkg/boot/src/main.rs@041: load_apps: false,
[ INFO]: pkg/boot/src/main.rs@041: }
[ INFO]: pkg/boot/src/fs.rs@048: Load file "KERNEL.ELF" t
o memory, size = 52280
[ INFO]: /home/camellia/ysos/0x01/pkg/elf/src/lib.rs@093:
Loading ELF file... @ 0x43c2000
[ INFO]: pkg/boot/src/fs.rs@063: Free ELF file, pages = 1
3, addr = 0x00000000043c2000
[ INFO]: pkg/boot/src/main.rs@115: Exiting boot services.
..

v0.1.0

[+] Serial Initialized.
[INFO ] pkg/kernel/src/utlis/logger.rs:13 - Logger Initia
lized.
[INFO ] pkg/kernel/src/lib.rs:28 - YatsenOS initialized.
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from Yats
enOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from Yats
enOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from Yats
enOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from Yats
enOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from Yats
enOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from Yats
enOS v2!

```

可以看到串口初始化成功，能够正常看到 `[+] Serial Initialized.` 的输出。

1.5. 日志输出

在 `pkg/kernel/src/utlis/logger.rs` 中找到日志输出的相关代码，你需要完成其中的 `init` 函数和 `log` 函数。

```
pub fn init() {
    static LOGGER: Logger = Logger;
    log::set_logger(&LOGGER).unwrap();

    // 配置日志级别过滤器
    // 这里设置为 Trace，表示所有级别的日志都会被输出
    // 可以根据需求调整为 Debug, Info, Warn 或 Error
    log::set_max_level(LevelFilter::Trace);

    info!("Logger Initialized.");
}
```

```
fn log(&self, record: &Record) {
    // 只处理启用的日志记录
    if self.enabled(record.metadata()) {
        // 根据日志级别添加不同的颜色和前缀
        let (color_code, level_str) = match record.level() {
            Level::Error => ("\x1b[31m", "ERROR"), // 红色
            Level::Warn => ("\x1b[33m", "WARN "), // 黄色
            Level::Info => ("\x1b[32m", "INFO "), // 绿色
            Level::Debug => ("\x1b[36m", "DEBUG"), // 青色
            Level::Trace => ("\x1b[90m", "TRACE"), // 灰色
        };

        // 格式化输出: [级别] 文件:行 - 消息
        if let (Some(file), Some(line)) = (record.file(), record.line()) {
            println!(
                "{}[{}]\x1b[0m {}:{} - {}",
                color_code, level_str, file, line, record.args()
            );
        } else {
            // 如果没有文件和行号信息，则只输出消息
            println!(
                "{}[{}]\x1b[0m {}",
                color_code, level_str, record.args()
            );
        }
    }
}
```

[测试日志输出]

注意：测试时直接使用 cargo build 会产生编译错误，应使用 python 编译脚本，此处将错误日志截图作为日志输出展示。

认 `feature` 是为了避免什么问题？请结合 `pkg/boot` 的 `Cargo.toml` 谈谈你的理解。

- 避免引导程序(boot)特有的依赖：

从 `Cargo.toml` 可以看到，`default = ["boot"]` 意味着默认会启用 `boot feature`，而 `boot feature` 启用了 `"uefi/alloc"`，`"uefi/logger"`，`"uefi/panic_handler"`，`"uefi/global_allocator"` 等依赖特性。

- 防止冲突产生：

`uefi/panic_handler` - 这会提供一个 panic 处理器，但内核肯定有自己的 panic 处理实现
`uefi/global_allocator` - 这会设置全局内存分配器，而内核通常也需要自己的内存分配器

- 避免重复的功能：

`uefi/logger` - 内核已经实现了自己的日志系统，不需要 UEFI 的日志功能

- 阶段性需求区分：

这些功能在引导阶段是必要的，但一旦控制权移交给内核后，内核会有自己的实现来替代它们。禁用默认 features 可以让内核只使用 boot 包中的必要部分，而不包括那些在内核环境中不适用或会造成冲突的功能。

2. 在 `pkg/boot/src/main.rs` 中参考相关代码，聊聊 `max_phys_addr` 是如何计算的，为什么要这么做？

- 首先通过 `mmap.entries()` 获取 UEFI 内存映射中的所有内存区域条目。
- 对每个条目计算其物理内存结束地址：`m.phys_start + m.page_count * 0x1000`，`m.phys_start` 是内存区域的起始物理地址，`m.page_count` 是该区域包含的页面数量，`0x1000` (4KB) 是页面大小
- 通过 `max()` 找出所有内存区域中最大的结束地址
- 最后与 `0x1_0000_0000` (4GB) 取最大值，确保地址至少达到 4GB

3. 串口驱动是在进入内核后启用的，那么在进入内核之前，显示的内容是如何输出的？

在进入内核之前 (UEFI 引导阶段)，输出是通过 UEFI 提供的标准输出服务实现的。UEFI 提供了多种标准输出服务：

- 控制台输出协议 (ConOut)：用于文本模式的屏幕输出
- 简单文本输出协议 (SimpleTextOutput)：提供简化的文本输出接口
- 图形输出协议 (GOP)：用于图形模式显示

3.1. UEFI 输出的关键实现

在引导加载程序中可以看到以下关键步骤：

代码	功能
<code>uefi::helpers::init()</code>	初始化 UEFI 辅助功能，包括日志系统
<code>log::set_max_level()</code>	设置日志输出级别
<code>extern crate log</code>	导入通用日志接口

3.2. 具体实现机制

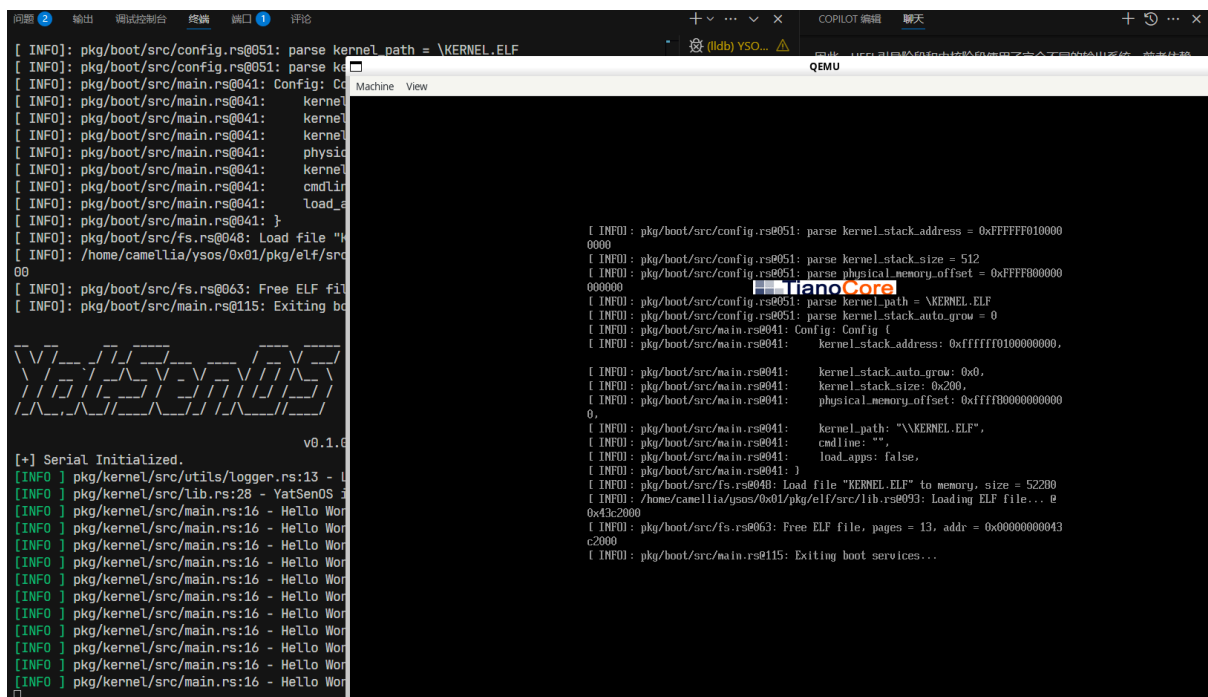
在 UEFI 环境中：

- log 宏被重定向到 UEFI 的输出服务
- 使用 uefi_services 包实现全局日志记录
- 通过 SystemTable 的 stdout() 服务处理实际输出
- UEFI 固件负责将输出显示在屏幕或通过调试通道输出

4. 在 QEMU 中，我们通过指定 `-nographic` 参数来禁用图形界面，这样 QEMU 会默认将串口输出重定向到主机的标准输出。

- 假如我们将 Makefile 中取消该选项，QEMU 的输出窗口会发生什么变化？请观察指令 `make run QEMU_OUTPUT=` 的输出，结合截图分析对应现象。
- 在移除 `-nographic` 的情况下，如何依然将串口重定向到主机的标准输入输出？请尝试自行构造命令行参数，并查阅 QEMU 的文档，进行实验。
- 如果你使用 `ysos.py` 来启动 qemu，可以尝试修改 `-o` 选项来实现上述功能。

[解答]相比于原来使用 `python3 ysos.py launch -d` 命令启动 QEMU 并进入调试模式，此处使用 `python3 ysos.py run -o "-serial stdio"` 命令启动 QEMU 图形界面并将串口输出重定向到终端，得到如下结果：



可以看到成功启动 QEMU 图形界面，串口输出重定向到终端。

五. 加分项

1. 😊 线控寄存器的每一比特都有特定的含义，尝试使用 bitflags 宏来定义这些标志位，并在 uart16550 驱动中使用它们。

1.1. 定义寄存器端口偏移

```
/// 定义寄存器端口偏移
#[repr(u16)]
enum UartRegister {
    Data = 0,                // 数据寄存器
    InterruptEnable = 1,     // 中断使能寄存器
    FifoControl = 2,         // FIFO控制寄存器
    LineControl = 3,         // 线控寄存器
    ModemControl = 4,       // 调制解调器控制寄存器
    LineStatus = 5,         // 线状态寄存器
    ModemStatus = 6,        // 调制解调器状态寄存器
    Scratch = 7,            // 临时寄存器
}
```

1.2. 使用 bitflags 宏定义线控寄存器的标志位

```
bitflags! {  
    /// 线控寄存器标志位 (LCR)  
    pub struct LineControlFlags: u8 {  
        /// 除数锁存访问位 (设置波特率)  
        const DLAB = 0x80;  
        /// 停止位 (0=1位, 1=2位)  
        const STOP_BITS_SET = 0x04;  
        /// 无奇偶校验  
        const NO_PARITY = 0x00;  
        /// 奇校验  
        const ODD_PARITY = 0x08;  
        /// 偶校验  
        const EVEN_PARITY = 0x18;  
        /// 5位数据长度  
        const WORD_LENGTH_5 = 0x00;  
        /// 6位数据长度  
        const WORD_LENGTH_6 = 0x01;  
        /// 7位数据长度  
        const WORD_LENGTH_7 = 0x02;  
        /// 8位数据长度  
        const WORD_LENGTH_8 = 0x03;  
    }  
  
    /// FIFO控制寄存器标志位 (FCR)  
    pub struct FifoControlFlags: u8 {  
        /// 启用FIFO  
        const ENABLE = 0x01;  
        /// 清空接收FIFO  
        const CLEAR_RECEIVE = 0x02;  
        /// 清空发送FIFO  
        const CLEAR_TRANSMIT = 0x04;  
        /// 使用DMA模式  
        const DMA_MODE = 0x08;  
        /// 触发级别1字节  
        const TRIGGER_LEVEL_1 = 0x00;  
        /// 触发级别4字节  
        const TRIGGER_LEVEL_4 = 0x40;  
        /// 触发级别8字节  
        const TRIGGER_LEVEL_8 = 0x80;  
        /// 触发级别14字节  
        const TRIGGER_LEVEL_14 = 0xC0;  
    }  
}
```

```

    /// 调制解调器控制寄存器标志位 (MCR)
    pub struct ModemControlFlags: u8 {
        /// Data Terminal Ready
        const DTR = 0x01;
        /// Request To Send
        const RTS = 0x02;
        /// 辅助输出1
        const AUX_OUTPUT1 = 0x04;
        /// 辅助输出2 (通常用于启用中断)
        const AUX_OUTPUT2 = 0x08;
        /// 回环模式
        const LOOPBACK = 0x10;
    }
    /// 线状态寄存器标志位 (LSR)
    pub struct LineStatusFlags: u8 {
        /// 数据就绪
        const DATA_READY = 0x01;
        /// 溢出错误
        const OVERRUN_ERROR = 0x02;
        /// 奇偶校验错误
        const PARITY_ERROR = 0x04;
        /// 帧错误
        const FRAMING_ERROR = 0x08;
        /// 断行信号
        const BREAK_SIGNAL = 0x10;
        /// 发送保持寄存器空
        const THR_EMPTY = 0x20;
        /// 发送器空
        const TRANSMITTER_EMPTY = 0x40;
        /// FIFO中有错误
        const FIFO_ERROR = 0x80;
    }
}

```

注意：考虑到报告篇幅以及实验重点，此处不再给出剩余部分需要细微调整的代码，仅展示测试结果。

1.3. 总结：主要改进点

1.3.1. 定义寄存器偏移量枚举：使用枚举来定义各个寄存器的偏移量，避免在代码中使用魔法数字。

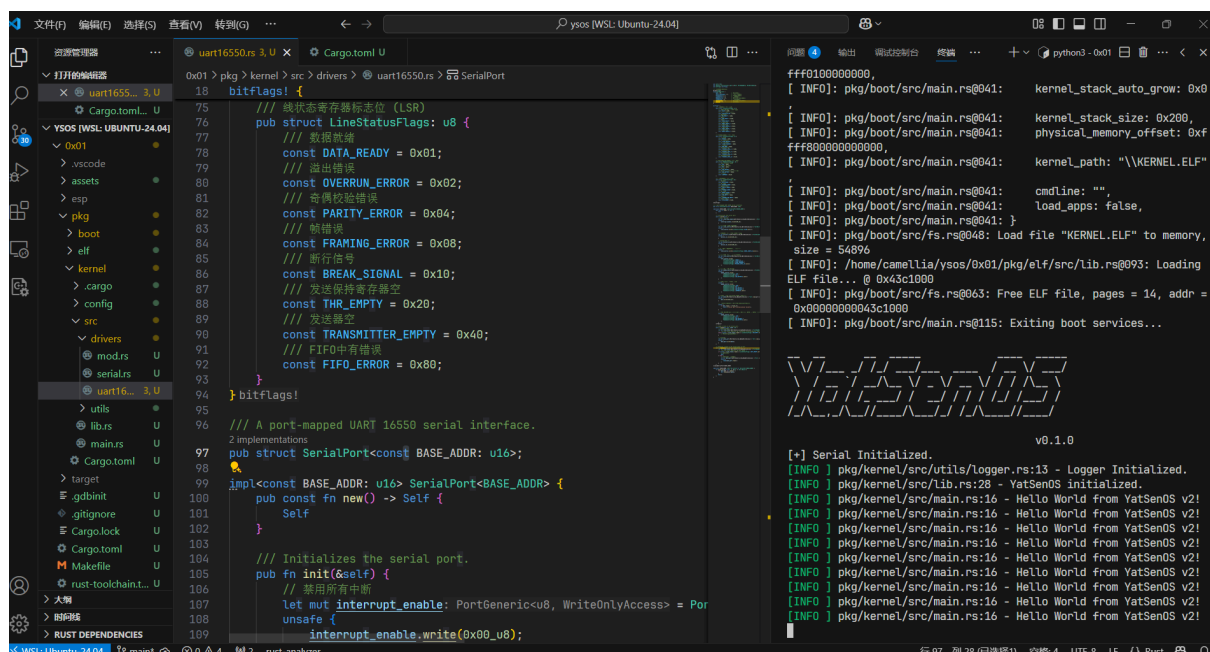
1.3.2. 使用 bitflags 定义标志位：为各个寄存器定义了对应的标志位结构体，使得代码更加清晰和自文档化。

- LineControlFlags: 线控寄存器的标志位
- FifoControlFlags: FIFO 控制寄存器的标志位
- ModemControlFlags: 调制解调器控制寄存器的标志位
- LineStatusFlags: 线状态寄存器的标志位

1.3.3. 使用组合标志位：通过位运算符组合多个标志位，例如在 FIFO 控制寄存器中同时启用 FIFO、清空接收和发送缓冲区，以及设置触发级别。

1.3.4. 代码可读性增强：使用有意义的常量名称替代魔法数字，使得代码更易于理解和维护。

注意：此处需要确保在项目的 Cargo.toml 文件中添加了 bitflags 依赖，否则无法编译通过。



The screenshot shows a code editor with a Rust project. The left sidebar shows the project structure with files like Cargo.toml, src, and drivers. The main editor displays the file `uart16550.rs` in the `drivers` directory. The code defines a `SerialPort` struct and implements a UART driver for a port-mapped UART 16550 serial interface. It includes a `bitflags!` macro call and a `pub struct SerialPort` definition. The right sidebar shows the terminal output, which includes boot logs and a stylized ASCII art logo for 'YatSenOS'.

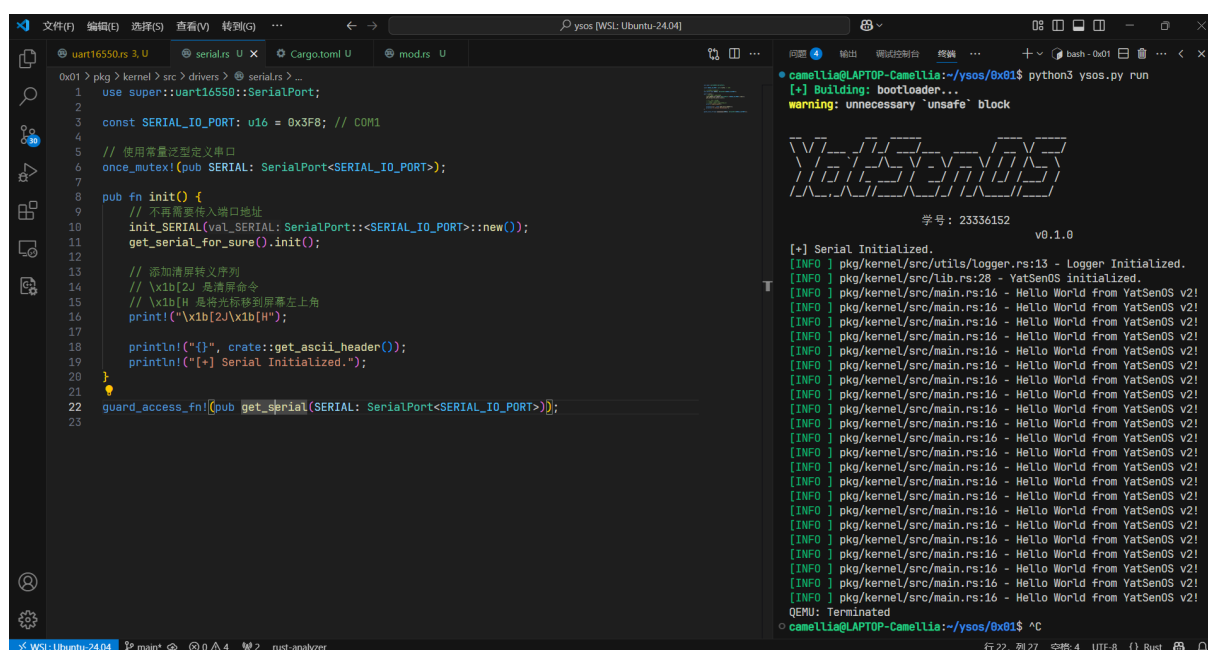
可以看到修改后的代码能够正常编译运行，完成串口的初始化等。

2. 😊 尝试在进入内核并初始化串口驱动后，使用 `escape sequence` 来清屏，并编辑 `get_ascii_header()` 中的字符串常量，输出你的学号信息。

2.1. ASCII 标题添加学号信息之后修改 `pkg/kernel/src/drivers/serial.rs` 中的串口初始化函数，添加清屏功能

```
// 添加清屏转义序列
// \x1b[2J 是清屏命令
// \x1b[H 是将光标移到屏幕左上角
print!("\x1b[2J\x1b[H");
```

重新执行 `python3 ysos.py run` 命令, 可以看到学号输出和清屏效果的实现: 没有了之前的终端输出。



3. 🤔 尝试添加字符串型启动配置变量 `log_level`, 并修改 `logger` 的初始化函数, 使得内核能够根据启动参数进行日志输出。

3.1. 在 pkg/kernel/src/config.rs 中添加 log_level 配置变量

```
/// Config for the bootloader
#[derive(Debug)]
pub struct Config<'a> {
    // ...
    /// Log level for kernel logger
    pub log_level: &'a str,
}
```

```
const DEFAULT_CONFIG: Config = Config {
    // ...
    log_level: "info",
};
```

```
fn process(&mut self, key: &str, value: &'a str) {
    // ...
    match key {
        // ...
        "log_level" => self.log_level = value,
        _ => warn!("undefined config key: {}", key),
    }
}
```

3.2. 修改 pkg/boot/src/lib.rs 中的 BootInfo 结构体, 添加 log_level 字段

```
pub struct BootInfo {
    // ...
    /// The log level for kernel logger
    pub log_level: &'static str,
}
```

3.3. 修改 bootloader 的 main.rs (pkg/boot/src/main.rs), 在 BootInfo 中添加 log_level 字段

```
// construct BootInfo
let bootinfo = BootInfo {
    // ...
    log_level: config.log_level,
};
```

3.4. 修改内核的 lib.rs (pkg/boot/src/lib.rs), 将日志级别传递给 logger

```
pub fn init(boot_info: &'static BootInfo) {
    // ...
    logger::init(boot_info.log_level); // 传递日志级别给logger
    // ...
}
```

3.5. 修改 `longger.rs` (`pkg/kernel/src/utils/logger.rs`), 支持日志级别配置

注意: 在 `no_std` 环境下, `&str` 类型没有 `to_lowercase()` 方法, 因为这个方法需要内存分配。需要修改日志级别解析代码, 使用不依赖内存分配的方式: 使用 `eq_ignore_ascii_case` 方法替代 `to_lowercase`。

```
/// 解析日志级别字符串, 返回对应的 LevelFilter
fn parse_log_level(level: &str) → LevelFilter {
    // 使用 eq_ignore_ascii_case 替代 to_lowercase
    let level = level.trim();

    if level.eq_ignore_ascii_case("off") {
        LevelFilter::Off
    } else if level.eq_ignore_ascii_case("error") {
        LevelFilter::Error
    } else if level.eq_ignore_ascii_case("warn") {
        LevelFilter::Warn
    } else if level.eq_ignore_ascii_case("info") {
        LevelFilter::Info
    } else if level.eq_ignore_ascii_case("debug") {
        LevelFilter::Debug
    } else if level.eq_ignore_ascii_case("trace") {
        LevelFilter::Trace
    } else {
        // 如果无法解析, 默认使用 Info 级别, 并打印警告
        println!("\x1b[33m[WARN ]\x1b[0m Unknown log level: {}, using 'info'",
level);
        LevelFilter::Info
    }
}
```

```
pub fn init(log_level: &str) {
    static LOGGER: Logger = Logger;
    log::set_logger(&LOGGER).unwrap();

    // 根据启动配置参数设置日志级别
    let level = parse_log_level(log_level);
    log::set_max_level(level);

    info!("Logger Initialized with level: {}", log_level);
}
```

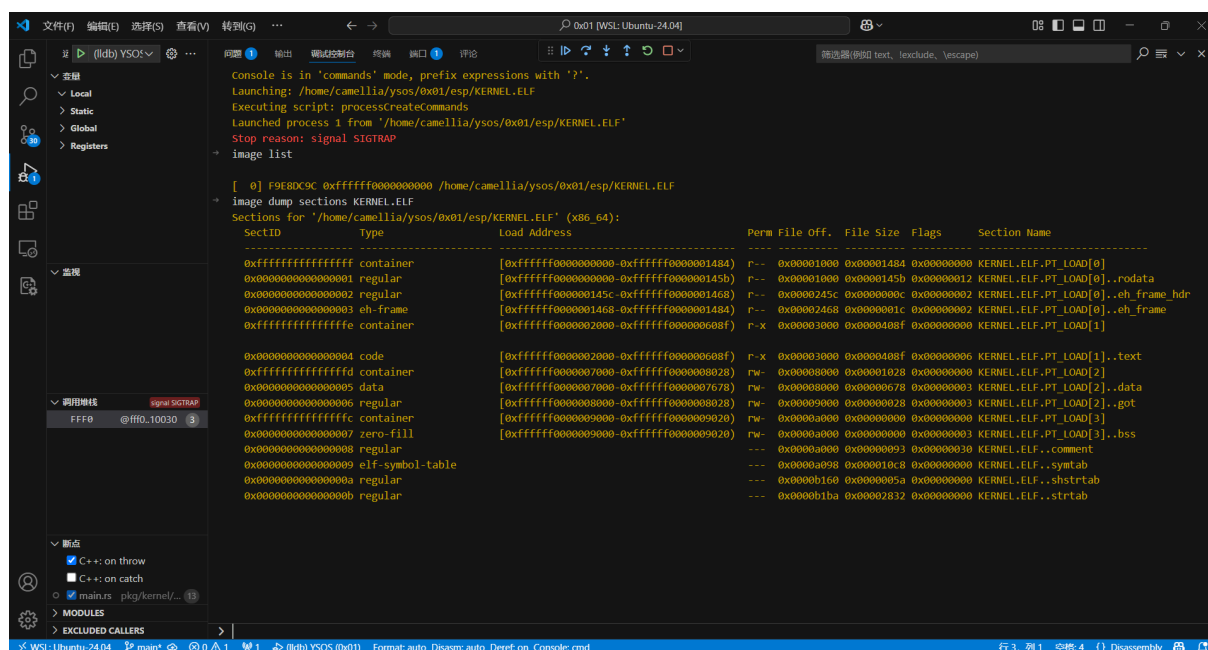
3.6. 最后在 `boot.conf (pkg/kernel/config/boot.conf)` 中添加 `log_level` 配置

```
# Log level for kernel: off, error, warn, info, debug, trace
log_level=info
```

4. 🤔 尝试使用调试器，在内核初始化之后（`ysos::init` 调用结束后）下断点，查看、记录并解释如下的信息：

- 内核的栈指针、栈帧指针、指令指针等寄存器的值。
- 内核的代码段、数据段、BSS 段等在内存中的位置。

首先在 0x01 下执行 `python3 ysos.py launch -d` 命令启动 QEMU 并进入调试模式，然后使用 LLDB 进行调试，在调试控制台输入 `image list` 查看内核的加载地址：
[0] F9E8DC9C 0xffffffff0000000000 /home/camellia/ysos/0x01/esp/KERNEL.ELF，可以看到内核加载地址为 `0xffffffff0000000000`。随后执行 `image dump sections KERNEL.ELF` 查看内核的段信息：



可以看到内核的代码段、数据段、BSS段等在内存中的位置，代码段为 `0xffffffff0000002000-0xffffffff000000608f`，数据段的起始地址为 `0xffffffff0000007000-0xffffffff0000007678`，BSS段的起始地址为 `0xffffffff0000009000-0xffffffff0000009020`。

注意：做的太慢了，第五个加分项来不及做了🤔。