Lab1 实验报告

零、实验目的

1. 从操作系统角度理解 MIPS 体系结构
2. 掌握操作系统启动的基本流程
3. 掌握 ELF 文件的结构和功能
4. 完成 `printk` 函数的编写

在本章中，需要阅读并填写部分代码，使得 MOS 操作系统可以正常的运行起来。这一章节的难度较为简单。

一、思考题

Thingking 1.1

> 在阅读 附录中的编译链接详解 以及本章内容后，尝试分别使用实验环境中的原生 x86 工具链（`gcc`、`ld`、`readelf`、`objdump` 等）和 MIPS 交叉编译工具链（带有 `mips-linux-gnu-` 前缀，如 `mips-linux-gnu-gcc`、`mips-linux-gnu-ld`），重复其中的编译和解析过程，观察相应的结果，并解释其中向 `objdump` 传入的参数的含义。

编写程序 `hello.c`：

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

x86 工具链：

预处理不编译：执行指令 `gcc -E hello.c`。得到了 800 多行的代码，这里展示一部分：

```c
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
```

```c
typedef unsigned long int __uint64_t;

extern struct _IO_FILE *stdin;
extern struct _IO_FILE *stdout;
extern struct _IO_FILE *stderr;

extern int printf (const char *__restrict __format, ...);
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

编译不链接：执行指令 `gcc -c hello.c`，产生目标文件 `hello.o`

反汇编：执行指令 `objdump -DS hello.o`，产生 96 行的汇编代码：

```
hello.o: file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
0:  55                  push %rbp
1:  48 89 e5            mov %rsp,%rbp
4:  bf 00 00 00 00      mov $0x0,%edi
9:  e8 00 00 00 00      callq e <main+0xe>
                        #本该填写 printf 地址的位置上被填写了一串 0。
e:  b8 00 00 00 00      mov $0x0,%eax
13: 5d                  pop %rbp
14: c3                  retq
```

编译链接：执行 `gcc -o hello hello.c`，得到可执行文件 `hello`。
反汇编：执行 `objdump -DS hello`：

```
hello: file format elf64-x86-64


Disassembly of section .init:

00000000004003a8 <_init>:
4003a8: 48 83 ec 08             sub $0x8,%rsp
4003ac: 48 8b 05 0d 05 20 00    mov 0x20050d(%rip),%rax
4003b3: 48 85 c0                test %rax,%rax
4003b6: 74 05                   je 4003bd <_init+0x15>
4003b8: e8 43 00 00 00          callq 400400 <__gmon_start__@plt>
                                #填写printf地址处填写了puts@plt标记的位置
4003bd: 48 83 c4 08             add $0x8,%rsp
4003c1: c3                      retq
```

```
Disassembly of section .plt:

00000000004003d0 <puts@plt-0x10>:
4003d0: ff 35 fa 04 20 00       pushq 0x2004fa(%rip)
4003d6: ff 25 fc 04 20 00       jmpq *0x2004fc(%rip)
4003dc: 0f 1f 40 00             nopl 0x0(%rax)
#......
```

可以推断，`printf` 的实现在“链接”这一步被插入可执行文件，链接器会将所有编译好的目标文件链接在一起，填入地址信息，形成最终的可执行文件。

MIPS 交叉编译工具链

预处理：执行 `mips-linux-gnu-gcc -E hello.c`，依然是头文件被展开，但代码有所不同。也是 800 多行。
编译不链接+反汇编：执行 `mips-linux-gnu-gcc -c hello.c` 以及 `mips-linux-gnu-objdump -DS hello.o`

```
 7 00000000 <main>:
 8    0:   27bdffe0        addiu   sp,sp,-32
 9    4:   afbf001c        sw      ra,28(sp)
10    8:   afbe0018        sw      s8,24(sp)
11    c:   03a0f025        move    s8,sp
12   10:   3c1c0000        lui     gp,0x0
13   14:   279c0000        addiu   gp,gp,0
14   18:   afbc0010        sw      gp,16(sp)
15   1c:   3c020000        lui     v0,0x0
16   20:   24440000        addiu   a0,v0,0
17   24:   8f820000        lw      v0,0(gp)
18   28:   0040c825        move    t9,v0
19   2c:   0320f809        jalr    t9
20   30:   00000000        nop
21   34:   8fdc0010        lw      gp,16(s8)
22   38:   00001025        move    v0,zero
23   3c:   03c0e825        move    sp,s8
24   40:   8fbf001c        lw      ra,28(sp)
25   44:   8fbe0018        lw      s8,24(sp)
26   48:   27bd0020        addiu   sp,sp,32
27   4c:   03e00008        jr      ra
28   50:   00000000        nop
29         ...
```

是计组课学过的 MIPS 汇编语言。（把行号也复制进来啦，实在不知道 vim 编辑器怎么和系统剪切板交互……）
编译链接+反汇编：执行 `mips-linux-gnu-gcc -o hello hello.c` 以及 `mips-linux-gnu-objdump -DS hello`

```
345 00400650 <main>:
346   400650:       27bdffe0        addiu   sp,sp,-32
347   400654:       afbf001c        sw      ra,28(sp)
348   400658:       afbe0018        sw      s8,24(sp)
349   40065c:       03a0f025        move    s8,sp
350   400660:       3c1c0043        lui     gp,0x43
351   400664:       279c8010        addiu   gp,gp,-32752
352   400668:       afbc0010        sw      gp,16(sp)
353   40066c:       3c020040        lui     v0,0x40
354   400670:       24440720        addiu   a0,v0,1824
355   400674:       8f828024        lw      v0,-32732(gp)
356   400678:       0040c825        move    t9,v0
357   40067c:       0320f809        jalr    t9
358   400680:       00000000        nop
359   400684:       8fdc0010        lw      gp,16(s8)
360   400688:       00001025        move    v0,zero
361   40068c:       03c0e825        move    sp,s8
362   400690:       8fbf001c        lw      ra,28(sp)
363   400694:       8fbe0018        lw      s8,24(sp)
364   400698:       27bd0020        addiu   sp,sp,32
365   40069c:       03e00008        jr      ra
366   4006a0:       00000000        nop
367         ...
```

发现不链接的时候一些立即数为 0 的位置被填上数字啦，说明所有目标文件都被链接在一起了。

`objdump` 传入参数的含义：

- `-D`：反汇编所有的 section；
- `-d`：反汇编特定指令机器码的 section；
- `-S`：尽可能反汇编出源代码；
- `-s`：显示指定 section 的完整内容。

Thinking 1.2

> 思考下述问题：
> • 尝试使用我们编写的 `readelf` 程序，解析之前在 `target` 目录下生成的内核 ELF 文件。

在 `tools/readelf` 目录下，执行命令：

```
./readelf ../../target/mos
```

解析内核 ELF 文件，结果如下：

```
0:0x0
1:0x80400000
2:0x804016f0
3:0x80401708
4:0x80401720
5:0x0
```

```
// ...
17:0x0
```

> • 也许你会发现我们编写的 `readelf` 程序是不能解析 `readelf` 文件本身的，而我们刚才介绍的系统工具 `readelf` 则可以解析，这是为什么呢？（提示：尝试使用 `readelf -h`，并阅读 `tools/readelf` 目录下的 `Makefile`，观察 `readelf` 与 `hello` 的不同）

解析我们编写的 `readelf`： `readelf -h readelf`（参数说明：`-h --file-header Display the ELF file header`）

```
ELF 头：
  Magic:    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              DYN (Position-Independent Executable file)
  系统架构:                          Advanced Micro Devices X86-64
  版本:                              0x1
  入口点地址:               0x1180
  程序头起点:          64 (bytes into file)
  Start of section headers:          14488 (bytes into file)
  标志:             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

解析 `hello`： `readelf -h hello`

```
ELF 头：
  Magic:    7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  类别:                              ELF32
  数据:                              2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          Intel 80386
  版本:                              0x1
  入口点地址:               0x8049750
  程序头起点:          52 (bytes into file)
  Start of section headers:          707128 (bytes into file)
  标志:             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
```

```
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

用我们编写的 `readelf` 解析 `readelf`： `./readelf readelf`，没有输出。
观察 Makefile：

```
 6 readelf: main.o readelf.o
 7         $(CC) $^ -o $@
 8
 9 hello: hello.c
10         $(CC) $^ -o $@ -m32 -static -g
```

> 看不懂这个 Makefile，喂给 DeepSeek 看看！

- `$(CC)` 是 Makefile 的预定义变量，默认值为 `cc` （为 `gcc` 或 `clang` 的别名）
- `$^` 是自动变量，表示所有的依赖项（`main.o readelf.o`）
- `$@` 是自动变量，表示当前目标（`readelf`）

其实就是 `gcc main.o readelf.o -o readelf` 。我勒个豆啊，装什么啊（x）。
那么下面的额外参数是什么？

- `-m32` 表示生成 32 位程序（默认 64 位）
- `-static` 静态链接（将所有依赖库打包到可执行文件中，文件会更大，但无需动态库即可执行（看不懂））
- `-g` 包含调试信息（方便 `gdb` 调试）

在上面用系统 `readelf` 的解析结果中，发现 `readelf` 的类别是 ELF64，而 `hello` 的类别是 ELF32。在 Makefile 中，使用了 `-m32` 参数来生成 32 位的 `hello` 可执行文件，而 `readelf` 没有这个参数。说明我们编写的 `readelf` 是 64 位的，不能解析； `hello` 是 32 位的，可以解析。（也有博客说是静态链接的原因）

Thinking 1.3

> 在理论课上我们了解到，MIPS 体系结构上电时，启动入口地址为 `0xBFC00000` （其实启动入口地址是根据具体型号而定的，由硬件逻辑确定，也有可能不是这个地址，但一定是一个确定的地址），但实验操作系统的内核入口并没有放在上电启动地址，而是按照内存布局图放置。思考为什么这样放置内核还能保证内核入口被正确跳转到？
> （提示：思考实验中启动过程的两阶段分别由谁执行。）

操作系统设计人员一般会将硬件初始化工作作为 bootloader 程序放在非易失存储器，因此启动流程被简化为加载内核到内存，跳转到内核的入口。启动流程分为两个阶段：

- stage1，进行初始化硬件设备，加载 stage2 到 RAM 空间，并设置堆栈最后跳转到 stage2 的入口函数。
- stage2，初始化这一阶段需要用到的硬件设备及其他功能，将内核镜像从存储器读到 RAM，为内核设置启动参数，最后将 CPU 指令存储器的内容设置为内核入口函数的地址，将控制权转交给系统内核，保证了内核地址被正确跳转到。

我们的 QEMU 中，stage2 中，`kernel.lds` 中设置了程序各个生成地址，使得 section 的位置被调整到了我们指定的地址上。通过 lds 文件控制各段（包括内核）被加载到我们预期的位置。把内核入口定为 `_start` 函数。

在 `init/start.S` 中通过对 `_start` 函数的设置，就可以正确的跳转到 `mips_init` 函数。

```
// init/start.S
  5 EXPORT(_start)
  6 .set at
  7 .set reorder
  8 /* Lab 1 Key Code "enter-kernel" */
  9         /* clear .bss segment */
 10         la      v0, bss_start
 11         la      v1, bss_end
 12 clear_bss_loop:
 13         beq     v0, v1, clear_bss_done
 14         sb      zero, 0(v0)
 15         addiu   v0, v0, 1
 16         j       clear_bss_loop
 17 /* End of Key Code "enter-kernel" */
// ...
 19 clear_bss_done:
 20         /* disable interrupts */
 21         mtc0    zero, CP0_STATUS
 22
 23         /* hint: you can refer to the memory layout in include/mmu.h */
 24         /* set up the kernel stack */
// ...
```

二、难点分析

项目结构

```
.
├── include // 存放系统头文件
│   ├── elf.h // 内含 ELF 段节
│   ├── mmu.h // 内含 MIPS 内存布局
│   ├── print.h
│   ├── printk.h
│   ├── string.h
├── include.mk
├── init // 内核初始化相关代码
│   ├── init.c
│   ├── init.o
│   ├── Makefile
│   ├── start.o
│   └── start.S
├── kern // 存放内核的主体代码
│   ├── include.mk
│   ├── machine.c
│   ├── Makefile
│   ├── panic.c
│   ├── printk.c
├── kernel.lds
├── lib // 存放一些常用的库函数，包括 vprintfmt
├── Makefile // 用于编译 MOS 内核的 Makefile 文件
├── target // 存放编译的产物
│   └── mos // 最终的 MOS 的可执行文件
├── tests // 存放测试程序
```

```
└── tools // 存放一些实用工具，包括 readelf
    ├── include.mk
    ├── Makefile
    ├── readelf
    │   ├── elf.h
    │   ├── hello.c
    │   ├── main.c
    │   ├── Makefile
    │   └── readelf.c
    └── run_bg.sh
```

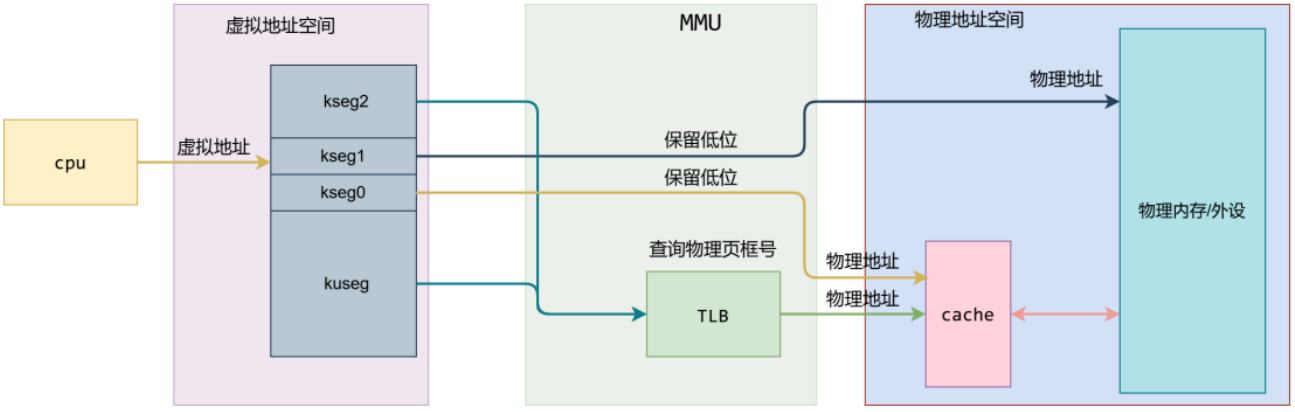
QEMU 模拟器

在本课程实验中，编写代码的环境是 Linux 系统，进行实验的硬件仿真平台是 QEMU 模拟器。

QEMU 中的启动流程：QEMU 模拟器直接加载 ELF 格式的内核，启动流程被简化为加载内核到内存，之后跳转到内核的入口，启动完成。因此我们需要**将内核编译成正确的 ELF 可执行文件**（我们的目标）

MIPS 内存布局——内核运行的正确位置

地址空间关系



`printk`

在 `kern/printk.c` 中：

```
void printk(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    vprintfmt(outputk, NULL, fmt, ap);
    va_end(ap);
}
```

变长参数：函数参数列表末尾有省略号，有至少一个固定参数，变长参数在参数表的末尾。在 `stdarg.h` 头文件中有如下定义：

- `va_list`：变长参数表的变量类型

- `va_start(va_list ap, lastarg)`：初始化变长参数表的宏
- `va_arg(va_list ap, 类型)`：取变长参数表下一个参数的宏
- `va_end(va_list ap)`：结束变长参数表的宏

在带变长参数表的函数内使用变长参数表前，需声明变长参数表类型，并进行初始化：

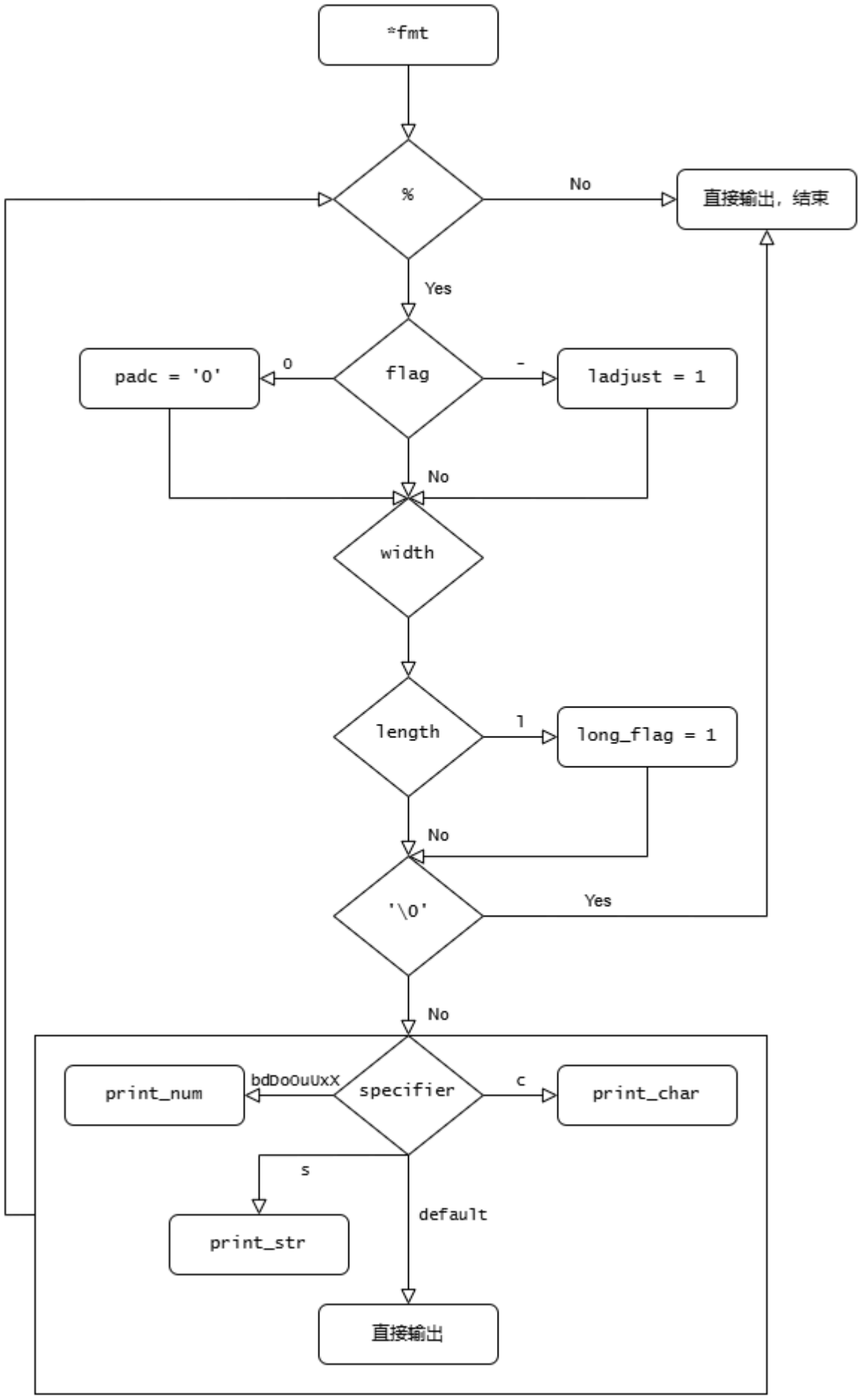
```
va_list ap;
va_start(ap, lastarg);
```

`lastarg` 是该函数最后一个命名的形式参数。初始化后，每次用 `va_arg` 取一个形式参数，如：

```
int num;
num = va_arg(ap, int);
```

退出函数前，调用 `va_end` 宏来结束变长参数表的使用。

`vprintfmt` **函数**：



三、实验体会

一上来就处理这么复杂的项目是有些难以把握的，并且不懂 Makefile 的高级语法也是困难之一。在实验过程中，为了能尽可能理解指导书想告诉我的，我一边阅读指导书一边写了很多笔记。但是限于实验报告篇幅，我把它们全删了，换成了更多可视化。

在完成实验题目的时候，我将我的思考过程尽可能完整地写下来了，但也不想放在实验报告里面（不然太臃肿了），也全删了。

四、原创说明

该报告参考了以下资料或博客：

1. BUAA-OS-lab1 | YannaのBlog
2. os-lab1实验报告 | hugo
3. OS Lab1 - 内核，启动与printf - Alkaid
4. OS*Lab1*实验报告 | 水货不水
5. https://www.deepseek.com/