OS实验记录

lab0

获取包信息

```
apt-cache searh <package name>
```

终端使用`qemu`调用`qemu-system-riscv64 \`.退出时`ctrl+a`送掉`ctrl`然后按`x`。

查找已经安装的包

```
dpkg -l | grep <pa>
```

查看程序返回值

首先运行程序

```
./your_program
```

然后输入

```
echo $?
```

查看某个文件夹内的用户访问权限

```
ls -l /path/to/folder
```

通过文件系统查看文件的授权信息。

使用`chmod`命令来更改文件夹的权限。你可以为所有者、所属组和其他用户分别授予不同的权限。以下是一些常见的权限标识符：

- `r`：读权限（4）
- `w`：写权限（2）
- `x`：执行权限（1）
- `-`：无权限（0）

给某一个文件授权

```
sudo chmod 777 /path/to/file
```

如果想直接给文件夹下的所有文件授权，添加`-R`参数循环迭代授权

```
sudo chmod 777 -R /path/to/folder
```

关于权限码

> 使用`chmod`的时候，用三位的权限码来分别表示文件所有者权限，文件所有者同组用户权限，其他用户权限。权限一共有三个：读、写、执行。
>
> 数字`7`表示`111`，也就是三个权限都授予。
>
> 数字`5`表示`101`，表示只授予读和执行权限。
>
> 其他以此类推。

`riscv64-unknown-elf-gcc`编译失败

具体表现为在源代码所在文件夹调用`riscv64-unknown-elf-gcc`编译任何文件都提示`Assembler Error`。

```
Assembler messages: Fatal error: invalid -march= option: `rv64imafdc'
```

本质原因是由于使用`sifive`预编译的`riscv64-unknown-elf-gcc`编译器去编译的时候，配套的汇编器没法进一步汇编`rv64imafdc`平台的机器指令，也就是汇编器选择出错。查看了预编译文件夹中的`as`工具，其大小为0，推测是这个原因。

要么选择`apt`安装

```
sudo apt install gcc-riscv64-unknown-elf
```

要么下载源文件手动重新编译

https://github.com/riscv-collab/riscv-gnu-toolchain

下载下来之后首先配置`configure`文件

```
./configure --prefix=/path/to/folder
```

这里使用`--prefix`参数指定编译的路径。指令执行完毕后会生成`Makefile`文件。

然后通过这个`Makefile`来完成编译指令

```
make linux
```

`make linux`表示生成目标为`linux`的可执行文件。

实际过程中碰到了

```
Makefile:999: *** multiple target patterns.  Stop.
```

的问题，暂时还没有解决。

编译`riscv64-unknown-elf-gdb`

如果想要在`riscv64-unknown-elf-gcc`编译器下使用`gdb`调试，必须使用配套的`riscv64-unknown-elf-gdb`来实现。

- 首先安装相应的依赖

```
sudo apt install libncurses5-dev python python-dev texinfo libreadline-dev
```

其中，`python`和`python-dev`并不是必须，如果后续配置的时候出问题，可以考虑先不装`python`和`python-dev`。

- 然后下载`gdb`的源代码，这里使用清华镜像源的`gdb-13.2`。解压并在该文件下打开终端

建立一个新的文件夹用于根据`configure`生成`Makefile`。

```
mkdir build
```

进入生成`Makefile`

```
../configure --prefix=/usr/local --with-python=/usr/bin/python --target=riscv64-unknown-elf --enable-tui=yes
```

这里`prefix`选项用于指定编译后的文件存放路径。

`with-python`和`enable-tui`都是用于支持后续安装一个可视化插件，并不是必须的内容，如果前面没装`python`，那么这里也不要指定这两个选项。

指令执行完毕后，可以看到我们所在的文件夹下有了一个`Makefile`文件，直接执行

```
make
```

如果使用`make -j`并发生成，可能会出现

```
gcc: fatal error: Killed signal terminated program cc1 compilation terminated.
```

等错误，多尝试几次，如果都是报的不一样的`fatal error`，很有可能是因为内存不足的原因。如果每次都是报同一个错误，可能真的是因为`cc1`或者`cc1plus`出了问题。这时候应该检查编译器的部件。

> 如果编译器用apt安装一般都没有问题，有问题也只能重装，因为我们并没有程序的源代码。
>
> 如果是编译器是通过源代码编译出来的，那么检查编译器源代码目录下的`libexec/gcc/_compiler_name_/_version_/`文件夹下是否有`cc1`或者`cc1plus`。
>
> 如果没有说明编译器的源码可能有问题，如果有，使用`ls -l .`确定是否具有所有文件的访问权限，授权之后再重新编译。

完成编译后记得

```
sudo make install
```

然后在终端检查版本信息，确认安装。

O(yuan)S(shen)，启动！

在启动之前，要首先将源代码的所有文件全部生成目标代码指令

```
make
```

直接使用`make`指令，默认是`make all`，将`Makefile`中的所有目标全部执行。

注意在这里我们首先会生成系统的镜像`ucore.img`，

```
UCOREIMG    := $(call totarget,ucore.img)


$(UCOREIMG): $(kernel)
    $(OBJCOPY) $(kernel) --strip-all -O binary $@

$(call create_target,ucore.img)
```

其路径会被记为`$UCOREIMG`，如果编译过程中这里报错，首先应该检查`ucore.img`是否已经存在于`bin`文件夹下，不存在的话说明生成有问题，反之应该检查权限等问题，也可以尝试将`$UCOREIMG`加入到`.bashrc`中。

然后

```
make qemu
```

`qemu`就会尝试执行`Makefile`中目标为`qemu`的指令脚本，运行`ucore`。如果看到`OpenSBI`的图形出现，并提示

```
(THU.CST) os is loading ...
```

说明系统就启动成功了。

使用GDB验证启动流程

首先定位到实验lab文件夹，在一个终端输入

```
qemu-system-riscv64 \
-machine virt \
-nographic \
-bios default \
-device loader,file=$(UCOREIMG),addr=0x80200000\
-s -S
```

这时终端处于等待状态。打开另一个终端，同样定位到实验lab文件夹，输入

```
riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'
```

这时`gdb`就会弹出启动信息和调试指令信息，按下回车正式开始`gdb`调试。

```
--Type <RET> for more, q to quit, c to continue without paging--
```

输入`c`之后，可以看到`qemu`运行的窗口出现`opensbi`的启动界面。模拟器启动成功。

练习1：使用gdb验证启动流程

- 众所周知，操作系统需要被加载到内存才能运行，但是操作系统不能自己把自己加载到内存。**如何将系统加载到内存呢**?

  在我们的日常使用的电脑上，机器上电后首先运行一个简单的输入输出的系统BIOS，BIOS是被写在固件(ROM)中的一段简单的程序代码，它的主要工作是初始化硬件，找到正确的引导设备（硬盘还是u盘还是软盘等）并且引导设备上的Bootloader。在我们的riscv64机器上，并没有BIOS这样的输入输出交互系统，引导Bootloader是由其他的方式完成的

  Bootloader是正儿八经的装载程序，负责将操作系统从外存中加载到内存运行。

那么riscv64机器从上电到系统初始化的过程是怎样的呢，这就是本节实验的目的。

如何启动程序的调试模式，以及如何启动gdb连接程序，上文都已经详细讲述，但是我们不希望每次都输入这么长的指令，这时候可以使用Makefile来直接运行指令脚本。

```
.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#   $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
    $(V)$(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000

debug: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
    $(V)$(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000\
        -s -S

gdb:
    riscv64-unknown-elf-gdb \
    -ex 'file bin/kernel' \
    -ex 'set arch riscv:rv64' \
    -ex 'target remote localhost:1234'
```

(Makefile当然不可能只有这么一点，这里摘出来的是我们要重点用到的一段)

在目标文件夹下，输入

```
make debug
```

ucore就会以调试模式启动。然后重新开启一个终端，同样进入对应的文件夹，输入

```
make gdb
```

gdb就会尝试执行上面的连接指令。

> gdb的一些常用调试指令：
>
> 1. **运行程序**:

- `gdb <可执行文件>`：启动GDB并加载指定的可执行文件。
- `r` 或 `run`：开始运行程序。

2. **设置断点**：
   - `b <函数名>`：在指定函数的开头设置断点。
   - `b <行号>`：在指定行号设置断点。
   - `b <文件名>:<行号>`：在指定文件和行号设置断点。
   - `delete <断点号>`：删除指定编号的断点。
   - `info breakpoints`：显示当前设置的断点列表。
3. **执行程序**：
   - `c` 或 `continue`：继续执行程序直到下一个断点。
   - `n` 或 `next`：执行下一行代码，如果是函数调用，则不进入函数内部。
   - `s` 或 `step`：执行下一行代码，如果是函数调用，则进入函数内部。
   - `finish`：执行当前函数的剩余部分并停止在返回处。
4. **查看变量和堆栈**：
   - `print <变量名>`：打印变量的值。
   - `info locals`：显示当前函数的局部变量。
   - `info args`：显示当前函数的参数。
   - `bt` 或 `backtrace`：显示函数调用堆栈。
   - `up` 和 `down`：在堆栈中上下导航。
   - `frame <帧号>`：切换到指定帧。
5. **修改变量值**：
   - `set variable <变量名> = <新值>`：修改变量的值。
6. **查看内存**：
   - `x/<格式> <地址>`：以指定格式查看内存内容，例如`x/x 0x12345678`以16进制格式查看内存地址0x12345678的内容。

     在GDB（GNU Debugger）中，`x` 命令用于查看内存中的数据。它的语法如下：

     ```
     x/<单位格式><数量格式><打印格式> 内存地址
     ```

     - `<单位格式>`：表示查看的内存单元的大小，可以是 `b`（字节，8位）、`h`（半字，16位，即两个字节）或者 `w`（字，32位，即四个字节）。
     - `<数量格式>`：表示要查看的内存单元的数量，可以是一个整数，也可以是一个寄存器（例如 `$eax`）。
     - `<打印格式>`：表示以何种格式显示内存中的数据，常见的格式有：
       - `x`：十六进制格式
       - `d`：十进制格式
         - `u`：无符号十进制格式
       - `o`：八进制格式
         - `t`：二进制格式
       - `c`：字符格式
       - `f`：浮点数格式
       - 等等。
7. **监视点**：

    - `watch <表达式>`：设置监视点，当表达式的值发生变化时停止程序执行。
8. **条件断点**：
    - `b <位置> if <条件>`：设置条件断点，仅当满足条件时触发断点。
9. **显示源代码**：
    - `list`：显示当前执行位置附近的源代码。
    - `list <函数名>`：显示指定函数的源代码。
10. **加载符号文件**：
    - `symbol-file <符号文件>`：加载指定的符号文件以获取更多调试信息。
10. **退出GDB**：
    - `q` 或 `quit`：退出GDB。
11. **GDB单指令执行一步**
    - `si`
12. **GDB显示某个地址之后的若干条汇编指令**
    - `x/10i 0x80000000`，显示`0x80000000`位置处的10条指令
13. **GDB显示指令寄存器`$pc`的指令**
    - `x/10i $pc`，显示`$pc`之后的10条指令。*注意`$pc`存的是下一条指令的地址，不是下一条指令的内容！！*
14. **GDB检查寄存器的值**
    - `info r t0`显示`t0`寄存器的值；`info register`检查所有寄存器的值。

`gdb`连接成功后，程序不会马上运行。根据上面的指南，查看当前`pc`地址的后10条指令如下

```
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      add     a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      .2byte  0x8000
   0x101c:      unimp
```

从这里我们可以看到一些现象：

- 我们所使用的`riscv`机器指令长度是32位的。**但是我们的`qemu`模拟出来的机器架构是64位的。**为了确定我们当前运行的操作系统到底是多少位的，在`gdb`调试台输入

  ```
  info file
  ```

  `gdb`调试器会返回当前运行的可执行文件信息

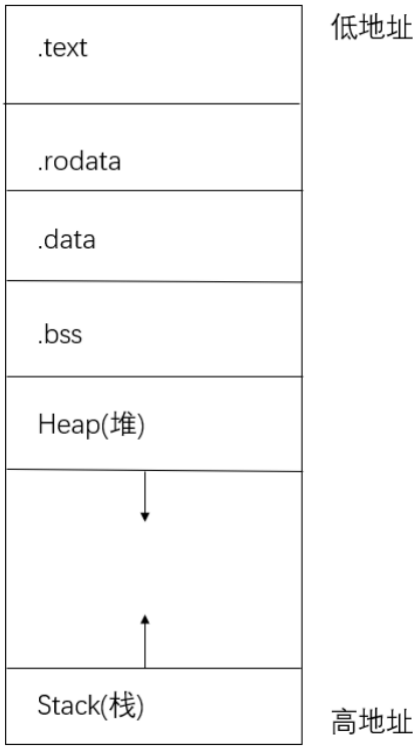
  ```
  Symbols from "/home/huangber/project/riscv_os/riscv64-ucore-
  labcodes/lab0/bin/kernel".
  Remote target using gdb-specific protocol:
          `/home/huangber/project/riscv_os/riscv64-ucore-
  labcodes/lab0/bin/kernel',
          file type elf64-littleriscv.
          Entry point: 0x80200000
  ```

```
        0x0000000080200000 - 0x00000000802004c8 is .text
        0x00000000802004c8 - 0x0000000080200738 is .rodata
        0x0000000080201000 - 0x0000000080203000 is .data
        0x0000000080203000 - 0x0000000080203008 is .sdata
        While running this, GDB does not access memory from...
Local exec file:
        `/home/huangber/project/riscv_os/riscv64-ucore-
labcodes/lab0/bin/kernel',
        file type elf64-littleriscv.
        Entry point: 0x80200000
        0x0000000080200000 - 0x00000000802004c8 is .text
        0x00000000802004c8 - 0x0000000080200738 is .rodata
        0x0000000080201000 - 0x0000000080203000 is .data
        0x0000000080203000 - 0x0000000080203008 is .sdata
```

这里的`filetype`提示了我们当前的运行的`kernel`是一个64位的`elf`文件。*注意`rv64i`指令集中的大多数指令和`rv32i`是一样的，长度也是32位。指令的长度和系统的位数没有必然关系*。见：

https://stackoverflow.com/questions/56874101/how-does-risc-v-variable-length-of-instruction-work-in-detail#:~:text=1%20Answer&text=RISC%2DV%20allows%20mixing%2016,registers%20are%2032%2Dbits%20wide.

> 上面的`.text`，`.rodata`这些又是什么呢，其实是一个程序的不同存储用途的不同空间的名字(内存布局)
>
> - `.text`段，即代码段，存放汇编代码
> - `.rodata`段，即只读数据段，顾名思义里面存放只读数据，通常是程序中 的常量
> - `.data`段，存放被初始化的可读写数据，通常保存程序中的全局变量
> - `.sdata`通常是用来存放静态数据（Static Data）的一个特定段。这些静态数据通常是全局或静态变量，它们在程序运行期间一直存在，不会被动态分配或释放，而是在程序启动时分配并在程序退出时释放。
> - `.bss`段，存放被初始化为 00 的 可读写数据，与`.data`段的不同之处在于我们知道它要被初始化为 00 ，因此在可执行文件中只需记录这个段 的大小以及所在位置即可，而不用记录里面的数据
> - `stack`，即栈，用来存储程序运行过程中的局部变量，以 及负责函数调用时的各种机制。它从高地址向低地址增长
> - `heap`，即堆，用来支持程序运行过程中内存的动 态分配，比如说你要读进来一个字符串，在你写程序的时候你也不知道它的长度究竟为多少，于是你只能在 运行过程中，知道了字符串的长度之后，再在堆中给这个字符串分配内存。



- `qemu` 模拟的机器上电后的复位地址是 `0x1000`。复位地址和程序的起始地址是两回事。上电后计算机首先执行复位程序，进行一些硬件初始化，**然后启动** `bootloader`。`bootloader` 会负责将操作系统加载到内存中。

  我们的 `bootloader` 被指定在 `0x80000000` 地址的位置。注意看其中几条指令

  ```
  0x100c:      ld      t0,24(t0)
  0x1010:      jr      t0
  ...
  0x1018:      unimp
  0x101a:      .2byte  0x8000
  ```

  这里如果在 `gdb` 中使用

  ```
  x/x 0x1018
  ```

  调试台会返回当前地址位置保存的值 `0x1018: 0x0000000080000000`。可以看到经过 `0x1010: jr t0` 之后，程序计数器会跳转到 `0x80000000` 的位置开始执行程序。

- `ucore` 的入口在 `0x80200000` 处。使用 `info file` 之后的 `Entry point: 0x80200000` 已经提示。至于为什么是在 `0x80200000`，这也是人为指定的。在源代码文件夹下的 `tools` 目录中有一个 `kernel.ld` 文件，这个文件是内核的链接文件，内核真正的进入地址在这个文件里面已经被规定了

  ```
  OUTPUT_ARCH(riscv)
  ENTRY(kern_entry)

  BASE_ADDRESS = 0x80200000;
  ```

  这时 `kernel.ld` 文件的头部信息，指定了内核编码输出的架构为 `riscv`，内核的入口是 `kern_entry`，内核存放的基址是 `0x80200000`。如果我们修改这个基址的值，对应的 `ucore` 的装载地址就变为修改后的地址。

lab1

lab1主要研究中断

中断主要有三种，并且都涉及到**用户态到内核态**的转换。

- 异常 (Exception)，指在执行一条指令的过程中发生了错误，此时我们通过中断来处理错误。最常见的异常包 括：访问无效内存地址、执行非法指令 (除零)、发生缺页等。他们有的可以恢复 (如缺页)，有的不可恢复 (如 除零)，只能终止程序执行。
- 陷入 (Trap)，指我们主动通过一条指令停下来，并跳转到处理函数。常见的形式有通过 ecall 进行系统调用 (syscall)，或通过 ebreak 进入断点 (breakpoint)。
- 外部中断 (Interrupt)，简称中断，指的是 CPU 的执行过程被外设发来的信号打断，此时我们必须先停下来对 该外设进行处理。典型的有定时器倒计时结束、串口收到数据等。

练习1：理解内核启动中的程序入口操作

为了说明这个问题，首先找到`kern/init`文件夹下的`entry.S`文件

```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

这里`.globl kern_entry`首先制定了一个全局的符号`kern_entry`，与上一节实验中的`ENTRY(kern_entry)`相照应。所谓的全局符号，其实就是声明了一段标签名为`kern_entry`的代码在这个地方。

`kern_entry`中只有两端指令：

- `la sp, bootstacktop`：这条指令使用伪指令`la`（load address）来将堆栈指针寄存器`sp`（Stack Pointer）的值设置为`bootstacktop`符号的地址。这意味着内核将使用`bootstacktop`符号所指示的位置作为初始堆栈顶部。

  > `bootstacktop`的地址是怎么来的呢？`0x80204000`又是怎么出来的呢？

- `tail kern_init`：这是一个跳转指令，它跳转到一个名为`kern_init`的函数。`tail`指令的特点是它会在跳转之前保存当前函数的返回地址，以便在跳转后返回。用于初始化操作系统内核。

  关于`kern_init`，这个其实在`kern_init.c`中就已经定义。机器会根据这个符号自动寻找到对应的代码所在位置

> 值得一提的是，上一节实验我们指出`ENTRY`是在`0x80200000`中，但是这只是装载的位置，而真正进入初始化函数`kern_init`的位置并不是这个

```
kern_entry:
 la sp, bootstacktop
 80200000:  00004117            auipc   sp,0x4
 80200004:  00010113            mv  sp,sp

 tail kern_init
 80200008:  a009                    j   8020000a <kern_init>

000000008020000a <kern_init>:
...
```

练习2：中断处理

完善`trap.c`中的中断处理函数。要求：

> 在对时钟中断进行处理的部分填写 kern/trap/trap.c 函数中处理时 钟中断的部分，使操作系统每遇到 100 次时钟中断后，调用 print_ticks 子程序，向屏幕上打印一行文字”100 ticks”，在打印完 10 行后调用 sbi.h 中的 shut_down() 函数关机。

源代码`trap.c`中，需要我们补全的位置

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_SOFT:
            cprintf("Supervisor software interrupt\n");
            break;
        case IRQ_H_SOFT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_SOFT:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_TIMER:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_TIMER:
            // "All bits besides SSIP and USIP in the sip register are
            // read-only." -- privileged spec1.9.1, 4.1.4, p59
            // In fact, Call sbi_set_timer will clear STIP, or you can clear it
            // directly.
            // cprintf("Supervisor timer interrupt\n");
             /* LAB1 EXERCISE2   YOUR CODE :  */
             clock_set_next_event();
             ticks++;
             if(ticks==100){
                print_ticks();
                ticks=0;
                num++;
```

```
            }
            if(num==10)
                sbi_shutdown();
            /*(1)设置下次时钟中断- clock_set_next_event()
             *(2)计数器（ticks）加一
             *(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断，同时打印次数（num）加一
            * (4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
            */
            break;
        case IRQ_H_TIMER:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_TIMER:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_EXT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_EXT:
            cprintf("Supervisor external interrupt\n");
            break;
        case IRQ_H_EXT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_EXT:
            cprintf("Machine software interrupt\n");
            break;
        default:
            print_trapframe(tf);
            break;
    }
}
```

关于`interrupt_handler()`，有一些值得注意的地方：

- 这里的`interrupt_handler()`函数是一个中断处理函数。在`trap()`函数中被调用

  ```
  void trap(struct trapframe *tf) { trap_dispatch(tf); }
  ```

  而`trap()`函数则是一个直接封装好的抛出中断的函数。

- 函数的参数`tf`是一个结构体，被定义在`trap.h`的头文件中

  ```
  struct trapframe {
      struct pushregs gpr;
      uintptr_t status;
      uintptr_t epc;
      uintptr_t badvaddr;
      uintptr_t cause;
  };
  ```

  这个结构体中包含了中断的各种状态参数。本代码中，通过`cause`成员来判断触发的中断类型

- `ticks`这个变量已经在其他文件中定义好，不需要再`trap.c`中重复定义，但是需要再函数头部声明。

  ```
  extern volatile size_t ticks;
  ```

代码的完善比较简单（见上）。直接运行`make qemu`的结果

```
OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...


Special kernel symbols:
  entry  0x000000008020000a (virtual)
  etext  0x00000000802009a8 (virtual)
  edata  0x0000000080204010 (virtual)
  end    0x0000000080204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

challenge 1：描述理解中断流程

> 描述 ucore 中处理中断异常的流程（从异常的产生开始），其中 mov a0，sp 的目的是什么？SAVE_ALL 中寄寄存器保存在栈中的位置是什么确定的？对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由。

首先确定，每次发生中断的时候必然会调用`trap_handler()`函数来处理中断。那么可以通过追踪该函数的调用堆栈来确定中断的起始位置。*但是实际上每次在*`gdb`*中调用*`bt`*的时候总是提示*`Backtrace stopped: frame did not save the PC`*然后看不到*

考虑直接在源文件中手动追踪。注意到在系统的入口文件中，调用了一些函数

```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();  // init the console
    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);
    print_kerninfo();

    // grade_backtrace();
    idt_init();  // init interrupt descriptor table

    // rdtime in mbare mode crashes
    clock_init();  // init clock interrupt

    intr_enable();  // enable irq interrupt
    while (1);
}
```

- 这里`idt_init()`函数是一个中断初始化的函数。它的定义位于`trap.c`。

  ```
  void idt_init(void) {
      extern void __alltraps(void);
      /* Set sscratch register to 0, indicating to exception vector that we
  are
       * presently executing in the kernel */
      write_csr(sscratch, 0);
      /* Set the exception vector address */
      write_csr(stvec, &__alltraps);
  }
  ```

  根据实验指导书知

  > 约定： 若中断前处于S态， `sscratch`为0
  >
  > 若中断前处于U态， `sscratch`存储内核栈地址
  >
  > 那么之后就可以通过`sscratch`的数值判断是内核态产生的中断还是用户态产生的中断
  >
  > 我们现在是内核态所以给`sscratch`置零

  `stvec`中存入了`__alltraps`符号的地址，对应我们中断向量，遇到中断的时候从寄存器中读取中断处理程序`__alltraps`的地址跳转执行。

  关于`__alltraps`中断处理程序，可以在`trapentry.S`的汇编文件中找到相应的内容。

```
#include <riscv.h>

    .macro SAVE_ALL

    csrw sscratch, sp

    addi sp, sp, -36 * REGBYTES
    # save x registers
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    STORE x3, 3*REGBYTES(sp)
    STORE x4, 4*REGBYTES(sp)
    STORE x5, 5*REGBYTES(sp)
    STORE x6, 6*REGBYTES(sp)
    STORE x7, 7*REGBYTES(sp)
    STORE x8, 8*REGBYTES(sp)
    STORE x9, 9*REGBYTES(sp)
    STORE x10, 10*REGBYTES(sp)
    STORE x11, 11*REGBYTES(sp)
    STORE x12, 12*REGBYTES(sp)
    STORE x13, 13*REGBYTES(sp)
    STORE x14, 14*REGBYTES(sp)
    STORE x15, 15*REGBYTES(sp)
    STORE x16, 16*REGBYTES(sp)
    STORE x17, 17*REGBYTES(sp)
    STORE x18, 18*REGBYTES(sp)
    STORE x19, 19*REGBYTES(sp)
    STORE x20, 20*REGBYTES(sp)
    STORE x21, 21*REGBYTES(sp)
    STORE x22, 22*REGBYTES(sp)
    STORE x23, 23*REGBYTES(sp)
    STORE x24, 24*REGBYTES(sp)
    STORE x25, 25*REGBYTES(sp)
    STORE x26, 26*REGBYTES(sp)
    STORE x27, 27*REGBYTES(sp)
    STORE x28, 28*REGBYTES(sp)
    STORE x29, 29*REGBYTES(sp)
    STORE x30, 30*REGBYTES(sp)
    STORE x31, 31*REGBYTES(sp)

    # get sr, epc, badvaddr, cause
    # Set sscratch register to 0, so that if a recursive exception
    # occurs, the exception vector knows it came from the kernel
    csrrw s0, sscratch, x0
    csrr s1, sstatus
    csrr s2, sepc
    csrr s3, sbadaddr
    csrr s4, scause

    STORE s0, 2*REGBYTES(sp)
    STORE s1, 32*REGBYTES(sp)
    STORE s2, 33*REGBYTES(sp)
    STORE s3, 34*REGBYTES(sp)
    STORE s4, 35*REGBYTES(sp)
    .endm
```

```
    .macro RESTORE_ALL

    LOAD s1, 32*REGBYTES(sp)
    LOAD s2, 33*REGBYTES(sp)

    csrw sstatus, s1
    csrw sepc, s2

    # restore x registers
    LOAD x1, 1*REGBYTES(sp)
    LOAD x3, 3*REGBYTES(sp)
    LOAD x4, 4*REGBYTES(sp)
    LOAD x5, 5*REGBYTES(sp)
    LOAD x6, 6*REGBYTES(sp)
    LOAD x7, 7*REGBYTES(sp)
    LOAD x8, 8*REGBYTES(sp)
    LOAD x9, 9*REGBYTES(sp)
    LOAD x10, 10*REGBYTES(sp)
    LOAD x11, 11*REGBYTES(sp)
    LOAD x12, 12*REGBYTES(sp)
    LOAD x13, 13*REGBYTES(sp)
    LOAD x14, 14*REGBYTES(sp)
    LOAD x15, 15*REGBYTES(sp)
    LOAD x16, 16*REGBYTES(sp)
    LOAD x17, 17*REGBYTES(sp)
    LOAD x18, 18*REGBYTES(sp)
    LOAD x19, 19*REGBYTES(sp)
    LOAD x20, 20*REGBYTES(sp)
    LOAD x21, 21*REGBYTES(sp)
    LOAD x22, 22*REGBYTES(sp)
    LOAD x23, 23*REGBYTES(sp)
    LOAD x24, 24*REGBYTES(sp)
    LOAD x25, 25*REGBYTES(sp)
    LOAD x26, 26*REGBYTES(sp)
    LOAD x27, 27*REGBYTES(sp)
    LOAD x28, 28*REGBYTES(sp)
    LOAD x29, 29*REGBYTES(sp)
    LOAD x30, 30*REGBYTES(sp)
    LOAD x31, 31*REGBYTES(sp)
    # restore sp last
    LOAD x2, 2*REGBYTES(sp)
    #addi sp, sp, 36 * REGBYTES
    .endm

    .globl __alltraps
.align(2)
__alltraps:
    SAVE_ALL

    move  a0, sp
    jal trap
    # sp should be the same as before "jal trap"

    .globl __trapret
```

```
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

这个程序主要定义了两个操作，一是`SAVE_ALL`，二是`RESTORE_ALL`。其中，`SAVE_ALL`的操作是将`x0`到`x31`，`s0`到`s4`寄存器的值保存到栈区。注意`s0`到`s4`寄存器保存的是状态字，`x0`到`x31`寄存器保存的是数据。中断之后需要将中断信息`sscratch`，`scause`这些东西写入内存中，不过由于`csr`（一种特殊的控制和状态寄存器）不能直接写到内存，只能先经过通用寄存器再送入内存。

> 这里这些状态寄存器指导书上也没说是干啥的，代码也没有注释有点逆天
>
> 1. `sscratch`（Scratch Register）：
>   - `sscratch`寄存器是一个通用的临时寄存器，通常用于在中断或异常处理时保存临时数据或上下文信息。它的内容可以在进入中断或异常处理程序之前被保存，并在恢复时重新加载。
> 2. `sstatus`（Status Register）：
>   - `sstatus`寄存器包含了一系列的状态位，用于控制和监视处理器的状态。其中一些重要的位包括：
>   - `SIE`：中断使能位，控制中断是否允许触发。
>   - `SPIE`：先前的中断使能位，用于在中断处理时暂时禁用中断。
>   - `SPP`：处理器模式位，指示处理器当前处于用户模式（0）还是超级用户/内核模式（1）。
>   - 其他位用于表示处理器状态和中断原因等信息。
> 3. `sepc`（Exception Program Counter）：
>   - `sepc`寄存器包含了在触发异常或中断时的程序计数器（PC）的值。当异常或中断发生时，处理器将当前 PC 的值保存到`sepc`寄存器，以便在异常处理程序完成后能够正确返回到程序的执行点。
> 4. `sbadaddr`（Bad Address）：
>   - `sbadaddr`寄存器包含了导致最近一次异常的指令的虚拟地址。这个寄存器在某些异常情况下非常有用，因为它可以帮助异常处理程序确定哪个地址或指令导致了异常。
> 5. `scause`（Cause Register）：
>   - `scause`寄存器包含了最近一次异常或中断的原因代码。它提供了异常的分类和信息，例如是中断、陷阱还是其他类型的异常，以及具体的异常原因代码。

而在`RESTORE_ALL`中，可以看到只有`x0`到`x31`的数据寄存器被恢复，而`s0`到`s4`的控制状态寄存器没有全部恢复，这是因为`s0`到`s4`寄存器本来也不是专门用来存储控制状态信息的，只是`csr`到内存的一个中转站。像`scratch`，`scause`这些状态信息，在处理完当前的中断之后其实就没有什么用处了，没必要恢复；`sepc`，`sstatus`分别需要用来帮助中断程序跳转回原来的程序以及恢复原来的状态（比如是否允许中断，优先级等），必须要恢复。

- `clock_init()`是时钟中断初始化函数。练习1中定时的时钟中断就是从这产生的。首先找到位于`clock.c`中`clock_init()`函数的定义

```
void clock_init(void) {
    // enable timer interrupt in sie
    set_csr(sie, MIP_STIP);
    // divided by 500 when using Spike(2MHz)
    // divided by 100 when using QEMU(10MHz)
    // timebase = sbi_timebase() / 500;
    clock_set_next_event();

    // initialize time counter 'ticks' to zero
    ticks = 0;

    cprintf("++ setup timer interrupts\n");
}
```

这个函数完成了两件重要的事：

1. 初始化了`SIE`状态字为`MIP_STIP`

   > `SIE`：中断使能位，控制中断是否允许触发。
   >
   > `MIP_STIP`的定义可以在`riscv.h`中找到
   >
   > ```
   > #define IRQ_S_TIMER  5
   > ...
   > #define MIP_STIP            (1 << IRQ_S_TIMER)
   > ...
   > #define SIP_STIP MIP_STIP
   > ```
   >
   > 不难看出，其实`sie`中存储的一个比特串的一个特定位置表示一个特定种类的中断。

2. 启动定时时钟中断`clock_set_next_event()`

   `clock_set_next_event()`函数的定义可以在`clock.c`找到定义

   ```
   void clock_set_next_event(void) { sbi_set_timer(get_cycles() +
   timebase); }
   ```

   而这里的`sbi_set_timer()`函数是一个与底层硬件交互的函数（带`sbi`的函数表示`Supervisor Binary Interface`的一种接口函数），在`sbi.c`中可以找到定义

   ```
   uint64_t SBI_SET_TIMER = 0;
   ...
   void sbi_set_timer(unsigned long long stime_value) {
       sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
   }
   ```

   这里的`sbi_call()`函数可以理解成一个通用的硬件接口函数，用于指定硬件执行某个特定的操作。它的定义同样在`sbi.c`中有定义

   ```
   uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1,
   uint64_t arg2) {
       uint64_t ret_val;
       __asm__ volatile (
           "mv x17, %[sbi_type]\n"
   ```

```
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n"
        "ecall\n"
        "mv %[ret_val], x10"
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r"
(arg1), [arg2] "r" (arg2)
        : "memory"
    );
    return ret_val;
}
```

`sbi_call()`函数可以直接执行一段汇编代码。在这段代码中，有一个汇编指令`ecall`，该指令可以触发一个异常事件。

> `ecall` 是 RISC-V 架构中的一种汇编指令，用于触发一个异常事件，通常用于进行系统调用（System Call）。RISC-V 是一种开放的指令集架构（ISA），具有可扩展性，广泛用于嵌入式系统和其他计算机架构。
>
> `ecall` 指令的主要作用是在用户程序（User Mode）中切换到特权模式（Privileged Mode），例如超级用户模式（Supervisor Mode）或机器模式（Machine Mode），以执行操作系统内核中的特权级别代码。通过执行 `ecall` 指令，用户程序可以请求执行特权级别代码来执行系统调用或其他需要特权级别权限的操作。
>
> 一般来说，`ecall` 指令会触发一个异常，然后处理器会跳转到预定义的异常处理程序地址。在异常处理程序中，操作系统内核会根据不同的系统调用号或其他条件来确定要执行的操作，然后执行相应的处理逻辑。系统调用通常用于执行一些敏感操作，例如文件 I/O、进程管理、内存管理等。

通过`gdb`单步调试的方法，可以找到`ecall`调用后系统执行的操作。`ecall`所引起的跳转是硬件级别的指令跳转，在源代码中是没有直接对应实现的。

```
(gdb) x/10i $pc
=> 0x802009ce <sbi_console_putchar+18>: ecall
   0x802009d2 <sbi_console_putchar+22>: mv      a5,a0
   0x802009d4 <sbi_console_putchar+24>: ret
   0x802009d6 <sbi_set_timer>:  li      a5,0
   0x802009d8 <sbi_set_timer+2>:        auipc   a4,0x3
   0x802009dc <sbi_set_timer+6>:        ld      a4,1608(a4)
   0x802009e0 <sbi_set_timer+10>:       mv      a7,a4
   0x802009e2 <sbi_set_timer+12>:       mv      a0,a0
   0x802009e4 <sbi_set_timer+14>:       mv      a1,a5
   0x802009e6 <sbi_set_timer+16>:       mv      a2,a5
(gdb) si
0x0000000080000474 in ?? ()
(gdb) x/55i $pc
=> 0x80000474:  sd      t0,64(tp) # 0x40
   0x80000478:  csrr    t0,mstatus
   0x8000047c:  srl     t0,t0,0xb
   0x80000480:  and     t0,t0,3
   0x80000484:  xor     t0,t0,3
   0x80000488:  beqz    t0,0x80000496
   0x8000048c:  add     t0,sp,zero
   0x80000490:  add     sp,tp,-272 # 0xfffffffffffffef0
```

```
   0x80000494:  j       0x8000049c
   0x80000496:  add     t0,sp,zero
   0x8000049a:  add     sp,sp,-272
   0x8000049c:  sd      t0,16(sp)
   0x8000049e:  ld      t0,64(tp) # 0x40
   0x800004a2:  sd      t0,40(sp)
   0x800004a4:  csrrw   tp,mscratch,tp
   0x800004a8:  csrr    t0,mepc
   0x800004ac:  sd      t0,256(sp)
   0x800004ae:  csrr    t0,mstatus
   0x800004b2:  sd      t0,264(sp)
   0x800004b4:  sd      zero,0(sp)
   0x800004b6:  sd      ra,8(sp)
   0x800004b8:  sd      gp,24(sp)
   0x800004ba:  sd      tp,32(sp)
   0x800004bc:  sd      t1,48(sp)
   0x800004be:  sd      t2,56(sp)
   0x800004c0:  sd      s0,64(sp)
   0x800004c2:  sd      s1,72(sp)
   0x800004c4:  sd      a0,80(sp)
   0x800004c6:  sd      a1,88(sp)
   0x800004c8:  sd      a2,96(sp)
   0x800004ca:  sd      a3,104(sp)
   0x800004cc:  sd      a4,112(sp)
   0x800004ce:  sd      a5,120(sp)
   0x800004d0:  sd      a6,128(sp)
   0x800004d2:  sd      a7,136(sp)
   0x800004d4:  sd      s2,144(sp)
   0x800004d6:  sd      s3,152(sp)
   0x800004d8:  sd      s4,160(sp)
   0x800004da:  sd      s5,168(sp)
   0x800004dc:  sd      s6,176(sp)
   0x800004de:  sd      s7,184(sp)
   0x800004e0:  sd      s8,192(sp)
   0x800004e2:  sd      s9,200(sp)
   0x800004e4:  sd      s10,208(sp)
   0x800004e6:  sd      s11,216(sp)
   0x800004e8:  sd      t3,224(sp)
   0x800004ea:  sd      t4,232(sp)
   0x800004ec:  sd      t5,240(sp)
   0x800004ee:  sd      t6,248(sp)
   0x800004f0:  add     a0,sp,zero
   0x800004f4:  csrr    a1,mscratch
   0x800004f8:  jal     0x80001cfa
   0x800004fc:  ld      ra,8(sp)
   0x800004fe:  ld      gp,24(sp)
   0x80000500:  ld      tp,32(sp)
(gdb) list
12      uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;
13      uint64_t SBI_SHUTDOWN = 8;
14
15      uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t
arg1, uint64_t arg2) {
16          uint64_t ret_val;
17          __asm__ volatile (
```

```
18              "mv x17, %[sbi_type]\n"
19              "mv x10, %[arg0]\n"
20              "mv x11, %[arg1]\n"
21              "mv x12, %[arg2]\n"
```

这里我们其实就已经可以看出一些端倪，`pc`指令寄存器存储地址的后面一大段都是在`sd`，也就是保存寄存器。但是注意这里中断并不是我们程序的中断实现，因为此时的地址是在`0x80200000`之前，还没到我们的程序段。

```
(gdb) info address __alltraps
```

`gdb`就会返回这个符号（标签）的地址：

```
Symbol "__alltraps" is at 0x802004a4 in a file compiled without
debugging.
```

这里我们就不细究`ecall`跳转之后是在干什么了，中间的步骤非常复杂。继续执行。根据`trapentry.s`的代码，`__alltraps`应该是所有中断都要进入的一个公共中断服务程序，执行完公共部分之后，程序会跳转到`trap`处

```
void trap(struct trapframe *tf) { trap_dispatch(tf); }
```

这里的`trap`函数是一个中断分配函数，根据`tf`的属性，将不同原因的中断进行划分，然后调用不同的中断处理

```
static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}
```

这里的`interrupt_handler()`就是分门别类处理不同中断的一个函数。

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_SOFT:
            cprintf("Supervisor software interrupt\n");
            break;
        case IRQ_H_SOFT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_SOFT:
            cprintf("Machine software interrupt\n");
            break;
```

```
        case IRQ_U_TIMER:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_TIMER:
            // "All bits besides SSIP and USIP in the sip register are
            // read-only." -- privileged spec1.9.1, 4.1.4, p59
            // In fact, Call sbi_set_timer will clear STIP, or you can
clear it
            // directly.
            // cprintf("Supervisor timer interrupt\n");
             /* LAB1 EXERCISE2   YOUR CODE :  */
             clock_set_next_event();
             ticks++;
             if(ticks==100){
                print_ticks();
                ticks=0;
                num++;
             }
            if(num==3){
                asm volatile (".word 0xEEEEFFFF" ); //非法未定义指令
                break;
            }

            if(num==5){
                asm volatile ("ebreak"); //从S态触发的异常断点指令
                break;
            }
            if(num==10)
                sbi_shutdown();
            /*(1)设置下次时钟中断- clock_set_next_event()
             *(2)计数器（ticks）加一
             *(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了
100次时钟中断，同时打印次数（num）加一
            * (4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
            */
            break;
        case IRQ_H_TIMER:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_TIMER:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_EXT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_EXT:
            cprintf("Supervisor external interrupt\n");
            break;
        case IRQ_H_EXT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_EXT:
            cprintf("Machine software interrupt\n");
            break;
        default:
```

```
            print_trapframe(tf);
            break;
    }
}
```

> 值得注意的是，如果这个`tf`不属于任何一个`case`那么就会被归为异常去处理。从这里也可以看出异常Exception和中断Interrupt之间的关系。

中断服务程序处理完毕之后，会回到`__trapret`处，这一部分的代码也在`trapentry.s`中定义

```
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

这里经过`RESTORE_ALL`之后还会继续返回到原来发生中断的地方，继续执行工作代码。

```
__trapret () at kern/trap/trapentry.S:114
114         sret
(gdb) info register
ra             0x8020004e       0x8020004e <kern_init+68>
sp             0x80203ff0       0x80203ff0
...
```

> 关于`tf`
>
> 这是一个由寄存器值和一些控制信息组成的结构体
>
> ```
> struct pushregs {
>  uintptr_t zero;  // Hard-wired zero
>  uintptr_t ra;    // Return address
>  uintptr_t sp;    // Stack pointer
>  uintptr_t gp;    // Global pointer
>  uintptr_t tp;    // Thread pointer
>  uintptr_t t0;    // Temporary
>  uintptr_t t1;    // Temporary
>  uintptr_t t2;    // Temporary
>  uintptr_t s0;    // Saved register/frame pointer
>  uintptr_t s1;    // Saved register
>  uintptr_t a0;    // Function argument/return value
>  uintptr_t a1;    // Function argument/return value
>  uintptr_t a2;    // Function argument
>  uintptr_t a3;    // Function argument
>  uintptr_t a4;    // Function argument
>  uintptr_t a5;    // Function argument
>  uintptr_t a6;    // Function argument
>  uintptr_t a7;    // Function argument
>  uintptr_t s2;    // Saved register
>  uintptr_t s3;    // Saved register
>  uintptr_t s4;    // Saved register
>  uintptr_t s5;    // Saved register
>  uintptr_t s6;    // Saved register
>  uintptr_t s7;    // Saved register
> ```

```
 uintptr_t s8;    // Saved register
 uintptr_t s9;    // Saved register
 uintptr_t s10;   // Saved register
 uintptr_t s11;   // Saved register
 uintptr_t t3;    // Temporary
 uintptr_t t4;    // Temporary
 uintptr_t t5;    // Temporary
 uintptr_t t6;    // Temporary
};

struct trapframe {
 struct pushregs gpr;
 uintptr_t status;
 uintptr_t epc;
 uintptr_t badvaddr;
 uintptr_t cause;
};
```

Lab2

本次实验主要内容是物理内存和页表，以及讨论不同的内存系统的内存分配算法。

本次实验的基础是在页式内存管理的基础上进行的，在讨论算法之前，首先需要了解我们的系统是如何对物理内存进行分页以及虚拟化的。

经过上面的前置实验，我们知道`ucore`系统的入口在`kern_init()`函数位置。本次实验在这个函数中，我们新添加了一个初始化内存的内容

```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();  // init the console
    const char *message = "(THU.CST) os is loading ...\0";
    //cprintf("%s\n\n", message);
    cputs(message);

    print_kerninfo();

    // grade_backtrace();
    idt_init();  // init interrupt descriptor table

    pmm_init();  // init physical memory management

    idt_init();  // init interrupt descriptor table

    clock_init();   // init clock interrupt
    intr_enable();  // enable irq interrupt



    /* do nothing */
    while (1)
        ;
```

```
}
```

这里的`pmm_init()`函数就是我们的内存管理系统的初始化。

如何组织一个高效的内存管理系统，一是如何管理物理连续内存，二是如何合理分配。这里我们首先介绍如何使用页表来管理物理连续内存。在`pmm.c`中，内存系统初始化函数被定义为

```
void pmm_init(void) {
    // We need to alloc/free the physical memory (granularity is 4KB or other
size).
    // So a framework of physical memory manager (struct pmm_manager)is defined
in pmm.h
    // First we should init a physical memory manager(pmm) based on the
framework.
    // Then pmm can alloc/free the physical memory.
    // Now the first_fit/best_fit/worst_fit/buddy_system pmm are available.
    init_pmm_manager();

    // detect physical memory space, reserve already used memory,
    // then use pmm->init_memmap to create free page list
    page_init();

    // use pmm->check to verify the correctness of the alloc/free function in a
pmm
    check_alloc_page();

    extern char boot_page_table_sv39[];
    satp_virtual = (pte_t*)boot_page_table_sv39;
    satp_physical = PADDR(satp_virtual);
    cprintf("satp virtual address: 0x%016lx\nsatp physical address: 0x%016lx\n",
satp_virtual, satp_physical);
}
```

这里的`page_init()`就是我们的目标，它同样被定义在`pmm.c`中

```
static void page_init(void) {
    va_pa_offset = PHYSICAL_MEMORY_OFFSET;

    uint64_t mem_begin = KERNEL_BEGIN_PADDR;
    uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;
    uint64_t mem_end = PHYSICAL_MEMORY_END; //硬编码取代 sbi_query_memory()接口

    cprintf("physcial memory map:\n");
    cprintf("  memory: 0x%016lx, [0x%016lx, 0x%016lx].\n", mem_size, mem_begin,
            mem_end - 1);

    uint64_t maxpa = mem_end;

    if (maxpa > KERNTOP) {
        maxpa = KERNTOP;
    }

    extern char end[];
```

```
    npage = maxpa / PGSIZE;
    //kernel在end[]结束, pages是剩下的页的开始
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);

    for (size_t i = 0; i < npage - nbase; i++) {
        SetPageReserved(pages + i);
    }

    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * (npage -
nbase));

    mem_begin = ROUNDUP(freemem, PGSIZE);
    mem_end = ROUNDDOWN(mem_end, PGSIZE);
    if (freemem < mem_end) {
        init_memmap(pa2page(mem_begin), (mem_end - mem_begin) / PGSIZE);
    }
}

...
static void init_memmap(struct Page *base, size_t n) {
    pmm_manager->init_memmap(base, n);
}
```

分页初始化函数就干了一件事：确定内存边界，然后将这一大段内存等分成`npage`页。分页之后要考虑我们内核所占用的页，这部分页是不能用于分配的。这里利用`end`变量作为`kernel`末尾的指针，向上取整到`PGSIZE`的倍数，然后保留这部分页。

剩下的页就是用户可以调用的页，也就是`mem_begin`到`mem_end`的部分。这部分页面进入页表。对应`init_memmap()`函数。页表交由`pmm_manager`来管理。每当需要分配内存的时候，就由`pmm_manager`从页表中选择一个空闲的页面返回。

> 这里的函数都是`static`函数，只在当前文件的作用域可以被使用。

关于`pmm_manager`，这是一个管理内存分配的结构体，定义在`pmm.h`中

```
struct pmm_manager {
    const char *name;  // XXX_pmm_manager's name
    void (*init)(
        void);  // initialize internal description&management data structure
                // (free block list, number of free block) of XXX_pmm_manager
    void (*init_memmap)(
        struct Page *base,
        size_t n);  // setup description&management data structcure according to
                    // the initial free physical memory space
    struct Page *(*alloc_pages)(
        size_t n);  // allocate >=n pages, depend on the allocation algorithm
    void (*free_pages)(struct Page *base, size_t n);  // free >=n pages with
                                                      // "base" addr of Page
                                                      // descriptor
                                                      // structures(memlayout.h)
    size_t (*nr_free_pages)(void);  // return the number of free pages
    void (*check)(void);            // check the correctness of XXX_pmm_manager
};
```

`pmm_manager`主要包括几个函数指针（并没有实际的成员变量，都是指针）。这几个函数指针是为了匹配不同内存分配方式的函数的不同实现（实际装载不同`pmm_manager`时，就是将函数指针指向对应的实现）

例如，如果要使用`default_pmm_manager`，在`default_pmm.c`中定义了

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

那么回到刚刚的问题，我们的内存被分页后调用了`init_memmap()`函数，将页表交给了对应的`pmm_manager`，这里以`default_pmm_manager`为例，在`default_pmm.c`中，这个函数是这么实现的

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

这个函数只了两件事：接受了第一页的页地址以及页数，一页一页向后检查页面的属性并且置位。然后将这些**全部的连续的页**看成一大块内存（订在一起看成一个本子？！），然后把这个本子挂到空闲列表`free_list`中。本子的地址就是第一页的地址，一共有多少页也被记录在了第一页的`property`属性中。

关于空闲列表，这是一个双向链表。它的定义在`default_pmm.c`的头部

```
//default_pmm.c
extern free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)
```

```
//memlayout.h
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // number of free pages in this free list
} free_area_t;
```

```
//list.h
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct list_entry list_entry_t;
```

```
//memlayout.h
struct Page {
    int ref;                        // page frame's reference counter
    uint64_t flags;                 // array of flags that describe the status
of the page frame
    unsigned int property;          // the num of free block, used in first fit
pm manager
    list_entry_t page_link;         // free list link
};
```

注意这里，我们的空闲链表中的元素都是`list_entry`，而不是`page`。当我们的程序去向系统申请内存的时候，返回的也是`list_entry`。`list_entry`从代码上看，没有数据域，**但是由于`C`的结构体特性，只要我们在`page`中包含`list_entry`项，那么就可以直接根据该`list_entry`的地址，找到整个`page`结构体的地址，因为结构体内成员的排布式连续的。**

那么要怎么根据`page`来找到对应的物理地址空间呢。这里我们可以在`pmm.h`中找到一个叫`page2pa()`的函数。

```
extern struct Page *pages;
extern size_t npage;
extern const size_t nbase;
extern uint64_t va_pa_offset;

static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }
static inline uintptr_t page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
```

> 必须要指出的是，我们使用页式模式来管理内存，是说我们每次进行内存操作都是页大小的整数倍的内存。虽然内存被分成一页一页，但是并没有真的为每一页内存都建立了一个数据结构来对应。比如在`first fit`的分配函数中，我们可以指定分配若干页
>
> ```
> 
> ```

```
default_alloc_pages(size_t n) {
 assert(n > 0);
 if (n > nr_free) {
     return NULL;
 }
 struct Page *page = NULL;
 list_entry_t *le = &free_list;
 while ((le = list_next(le)) != &free_list) {
     struct Page *p = le2page(le, page_link);
     if (p->property >= n) {
         page = p;
         break;
     }
 }
 if (page != NULL) {
     list_entry_t* prev = list_prev(&(page->page_link));
     list_del(&(page->page_link));
     if (page->property > n) {
         struct Page *p = page + n;
         p->property = page->property - n;
         SetPageProperty(p);
         list_add(prev, &(p->page_link));
     }
     nr_free -= n;
     ClearPageProperty(page);
 }
 return page;
}
```

这段代码干了两件事，一是找到一个大小合适的“本子”，二是将这个本子中你需要的这么多页内存摘出来一个小本子返回。假如我们调用`default_alloc_pages(2)`，那么后面的`struct Page *p = page + n`这里就会把2页内存划出来，第一步找到的这个本子它的`property`减掉2（页数减2），划出来的这部分的新的小本子封面注上“这段内存已经被分配”（`flag`的`PG_property`位被清0）。

注意划出来两页的过程中，我们并没有给这个两页显式分配一个独立的数据结构`Page`来管理，这是因为这一段内存初始化时已经被看作了一堆连续的`Page`。

```
default_init_memmap(struct Page *base, size_t n) {
 assert(n > 0);
 struct Page *p = base;
 for (; p != base + n; p ++) {
     assert(PageReserved(p));
     p->flags = p->property = 0;
     set_page_ref(p, 0);
 }
 ...
```

`base`是用户页内存的起始地址，每隔一个`Page`的大小，就把这一段内存的`flag`置位，`property`初始化。虽然没有明说，但是相当于就是把这些设置的内存看作一个`Page`结构体在操作。

使用`gdb`调试`default_init_memmap`函数

```
(gdb) info args
base = 0xffffffffc020f318
n = 31929
```

练习1 深入理解First Fit内存分配算法的实现

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

经过上面的介绍，不难看出`first fit`算法就是在`free_list`中找到第一个大小符合条件的本子，然后将本子的第一页返回出去。

这里需要补充一下如何释放内存，不论是`first fit`还是`best fit`算法，内存的释放与回收都是一样的，这里我们搬出`first fit`中的内存释放回收函数。

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
```

```
    SetPageProperty(base);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}
```

内存的释放干了三件事

- 首先将准备释放的内存的页面状态位全部重置。也就是第一个`for`循环干的事情。然后把第一页的`property`先重新写上`n`，也就是回收的内存页数。
- 然后去查看空闲链表`free_list`。如果链表上什么都没有了，就把这部分回收的内存订成本子，挂到空闲链表上面去。反之，如果空闲链表上有其他的本子，那么遍历上面的本子，找到的第一个地址比当前本子地址大的项，就把当前回收的这个本子插入到它的前面去。如果找遍了也没找到，那么就放到最后去。
- 最后分别检查这个本子的前面和后面，如果本子和本子之间的内存是连续的，那么把这两个本子合订到一起。

练习2 完成Best fit

相比于`first fit`，`best fit`就只改动了三行

```
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
     /*LAB2 EXERCISE 2: YOUR CODE*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 遍历空闲链表，查找满足需求的空闲页框
    // 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量
    int min_property=999999;//add a variable to record best property
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_property) {//find a new
available page which has a less property
            min_property=p->property;//maintain variable
            page = p;
            //break;
            //remove break
        }
    }

    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

内存的回收与释放与`first fit`中一模一样。