# 目录

# lab0

# Lab 0: RV64 内核调试

## 1 实验简介

通过在QEMU上运行Linux来熟悉如何从源代码开始将内核运行在QEMU模拟器上，并且掌握使用GDB跟QEMU进行联合调试，为后续实验打下基础。

## 2 实验环境

- Ubuntu 18.04.5 LTS
- Docker 下载地址 备用地址（内网访问）

## 3 实验基础知识介绍

### 3.1 Linux 使用基础

在Linux环境下，人们通常使用命令行接口来完成与计算机的交互。终端（Terminal）是用于处理该过程的一个应用程序，通过终端你可以运行各种程序以及在自己的计算机上处理文件。在类Unix的操作系统上，终端可以为你完成一切你所需要的操作。
下面我们仅对实验中涉及的一些概念进行介绍，你可以通过下面的链接来对命令行的使用进行学习：

1. The Missing Semester of Your CS Education >>Video<<
2. GNU/Linux Command-Line Tools Summary
3. Basics of UNIX

#### 环境变量

当我们在终端输入命令时，终端会找到对应的程序来运行。我们可以通过 `which` 命令来做一些小的实验：

```
$ which gcc
/usr/bin/gcc
$ ls -l /usr/bin/gcc
lrwxrwxrwx 1 root root 5 5月  21  2019 /usr/bin/gcc -> gcc-7
```

可以看到，当我们在输入 `gcc` 命令时，终端实际执行的程序是 `/usr/bin/gcc` 。实际上，终端在执行命令时，会从 `PATH` 环境变量所包含的地址中查找对应的程序来执行。我们可以将 `PATH` 变量打印出来来检查一下其是否包含 `/usr/bin` 。

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/phantom/.local/bin
```

在后面的实验中，如果你想直接访问 `riscv64-unknown-linux-gnu-gcc` 、 `qemu-system-riscv64` 等程序，那么你需要把他们所在的目录添加到目录中。

```
$ export PATH=$PATH:/opt/riscv/bin
```

### 3.2 Docker 使用基础

#### Docker 基本介绍

Docker 是一种利用容器（container）来进行创建、部署和运行应用的工具。Docker把一个应用程序运行需要的二进制文件、运行需要的库以及其他依赖文件打包为一个包（package），然后通过该包创建容器并运行，由此被打包的应用便成功运行在了Docker容器中。之所以要把应用程序打包，并以容器的方式运行，主要是因为在生产开发环境中，常常会遇到应用程序和系统环境变量以及一些依赖的库文件不匹配，导致应用无法正常运行的问题。Docker带来的好处是只要我们将应用程序打包完成（组装成为Docker imgae），在任意安装了Docker的机器上，都可以通过运行容器的方式来运行该应用程序，因而将依赖、环境变量等带来的应用部署问题解决了。
Docker和虚拟机功能上有共同点，但是和虚拟机不同，Docker不需要创建整个操作系统，只需要将应用程序的二进制和有关的依赖文件打包，因而容器内的应用程序实际上使用的是容器外Host的操作系统内核。这种共享内核的方式使得Docker的移植和启动非常的迅速，同时由于不需要创建新的OS，Docker对于容器物理资源的管理也更加的灵活，Docker用户可以根据需要动态的调整容器使用的计算资源（通过cgroups）。

#### Docker 安装

如果你在 Ubuntu 发行版上安装 Docker，请参考这里，其余平台请自行查找。 你可以从2中获得实验所需的环境，我们已经为你准备好了RISC-V工具链，以及QEMU模拟器，使用方法请参见4.1。

## 3.3 QEMU 使用基础

### 什么是QEMU

QEMU最开始是由法国程序员Fabrice Bellard开发的模拟器。QEMU能够完成用户程序模拟和系统虚拟化模拟。用户程序模拟指的是QEMU能够将为一个平台编译的二进制文件运行在另一个不同的平台，如一个ARM指令集的二进制程序，通过QEMU的TCG（Tiny Code Generator）引擎的处理之后，ARM指令被转化为TCG中间代码，然后再转化为目标平台（比如Intel x86）的代码。系统虚拟化模拟指的是QEMU能够模拟一个完整的系统虚拟机，该虚拟机有自己的虚拟CPU，芯片组，虚拟内存以及各种虚拟外部设备，能够为虚拟机中运行的操作系统和应用软件呈现出与物理计算机完全一致的硬件视图。

### 如何使用 QEMU（常见参数介绍）

以以下命令为例，我们简单介绍QEMU的参数所代表的含义

```
$ qemu-system-riscv64 -nographic -machine virt -kernel build/linux/arch/riscv/boot/Image  \
 -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"  \
 -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
 -netdev user,id=net0 -device virtio-net-device,netdev=net0 -S -s
```

**-nographic**: 不使用图形窗口，使用命令行
**-machine**: 指定要emulate的机器，可以通过命令 `qemu-system-riscv64 -machine help` 查看可选择的机器选项
**-kernel**: 指定内核image
**-append cmdline**: 使用cmdline作为内核的命令行
**-device**: 指定要模拟的设备，可以通过命令 `qemu-system-riscv64 -device help` 查看可选择的设备，通过命令 `qemu-system-riscv64 -device <具体的设备>,help` 查看某个设备的命令选项
**-drive, file=<file_name>**: 使用'file'作为文件系统
**-netdev user,id=str**: 指定user mode的虚拟网卡, 指定ID为str
**-S**: 启动时暂停CPU执行(使用'c'启动执行)
**-s**: -gdb tcp::1234 的简写
**-bios default**: 使用默认的OpenSBI firmware作为bootloader

更多参数信息可以参考这里

## 3.4 GDB 使用基础

### 什么是 GDB

GNU调试器（英语：GNU Debugger，缩写：gdb）是一个由GNU开源组织发布的、UNIX/LINUX操作系统下的、基于命令行的、功能强大的程序调试工具。借助调试器，我们能够查看另一个程序在执行时实际在做什么（比如访问哪些内存、寄存器），在其他程序崩溃的时候可以比较快速地了解导致程序崩溃的原因。 被调试的程序可以是和gdb在同一台机器上（本地调试，or native debug），也可以是不同机器上（远程调试， or remote debug）。

总的来说，gdb可以有以下4个功能

- 启动程序，并指定可能影响其行为的所有内容
- 使程序在指定条件下停止
- 检查程序停止时发生了什么
- 更改程序中的内容，以便纠正一个bug的影响

### GDB 基本命令介绍

(gdb) start：单步执行，运行程序，停在第一执行语句
(gdb) next：单步调试（逐过程，函数直接执行）,简写n
(gdb) run：重新开始运行文件（run-text：加载文本文件，run-bin：加载二进制文件），简写r
(gdb) backtrace：查看函数的调用的栈帧和层级关系，简写bt
(gdb) break 设置断点。比如断在具体的函数就break func；断在某一行break filename:num
(gdb) finish：结束当前函数，返回到函数调用点
(gdb) frame：切换函数的栈帧，简写f
(gdb) print：打印值及地址，简写p
(gdb) info：查看函数内部局部变量的数值，简写i；查看寄存器的值i register xxx
(gdb) display：追踪查看具体变量值

更多命令可以参考100个gdb小技巧

#### 3.4.3 GDB 插件使用（不做要求）

单纯使用gdb比较繁琐不是很方便，我们可以使用gdb插件让调试过程更有效率。推荐各位同学使用gef，由于当前工具链还不支持 `python3`，请使用旧版本的gef-legacy。
该仓库中已经取消的原有的安装脚本，同学们可以把 `gef.py` 脚本拷贝下来，直接在 `.gdbinit` 中引导，感兴趣的同学可以参考这篇文章（内网访问）。

### 3.5 LINUX 内核编译基础

#### 交叉编译

交叉编译指的是在一个平台上编译可以在另一个平台运行的程序，例如在x86机器上编译可以在arm平台运行的程序，交叉编译需要交叉编译工具链的支持。

#### 内核配置

内核配置是用于配置是否启用内核的各项特性，内核会提供一个名为 `defconfig` (即default configuration) 的默认配置，该配置文件位于各个架构目录的 `configs` 文件夹下，例如对于RISC-V而言，其默认配置文件为 `arch/riscv/configs/defconfig` 。使用 `make ARCH=riscv defconfig` 命令可以在内核根目录下生成一个名为 `.config` 的文件，包含了内核完整的配置，内核在编译时会根据 `.config` 进行编译。配置之间存在相互的依赖关系，直接修改defconfig文件或者 `.config` 有时候并不能达到想要的效果。因此如果需要修改配置一般采用 `make ARCH=riscv menuconfig` 的方式对内核进行配置。

#### 常见参数

**ARCH** 指定架构，可选的值包括arch目录下的文件夹名，如x86,arm,arm64等，不同于arm和arm64，32位和64位的RISC-V共用 `arch/riscv` 目录，通过使用不同的config可以编译32位或64位的内核。

**CROSS_COMPILE** 指定使用的交叉编译工具链，例如指定 `CROSS_COMPILE=aarch64-linux-gnu-` ，则编译时会采用 `aarch64-linux-gnu-gcc` 作为编译器，编译可以在arm64平台上运行的kernel。

**CC** 指定编译器，通常指定该变量是为了使用clang编译而不是用gcc编译，Linux内核在逐步提供对clang编译的支持，arm64和x86已经能够很好的使用clang进行编译。

#### 常用编译选项

```
$ make defconfig        ### 使用当前平台的默认配置，在x86机器上会使用x86的默认配置
$ make -j$(nproc)       ### 编译当前平台的内核，-j$(nproc)为以机器硬件线程数进行多线程编译

$ make ARCH=riscv defconfig ### 使用RISC-V平台的默认配置
$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j$(nproc)     ### 编译RISC-V平台内核

$ make clean              ### 清除所有编译好的object文件
$ make mrproper               ### 清除编译的配置文件，中间文件和结果文件

$ make init/main.o        ### 编译当前平台的单个object文件init/main.o（会同时编译依赖的文件）
```

## 4 实验步骤

通常情况下，`$` 提示符表示当前运行的用户为普通用户，`#` 代表当前运行的用户为特权用户。
但注意，在下文的示例中，以 `###` 开头的行代表注释，`$` 开头的行代表在你的宿主机/虚拟机上运行的命令，`#` 开头的行代表在 `docker` 中运行的命令，`(gdb)` 开头的行代表在 `gdb` 中运行的命令。
**在执行每一条命令前，请你对将要进行的操作进行思考，给出的命令不需要全部执行，并且不是所有的命令都可以无条件执行，请不要直接复制粘贴命令去执行。**

### 4.1 搭建docker环境

```
### docker安装可使用下面命令，或者参考给出的官方链接
$ curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun

### 将用户加入docker组，免 sudo
$ sudo usermod -aG docker $USER   ### 注销后重新登陆生效
$ sudo chmod a+rw /home/$USER/.docker/config.json
```

```
### 导入docker镜像
$ cat oslab.tar | docker import - oslab:2020
### 执行命令后若出现以下错误提示
### ERROR: Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock
### 可以使用下面命令为该文件添加权限来解决
### $ sudo chmod a+rw /var/run/docker.sock

### 查看docker镜像
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
oslab               2020                d7046ea68221        5 seconds ago       2.89GB

### 从镜像创建一个容器
$ docker run -it oslab:2020 /bin/bash    ### -i:交互式操作 -t:终端
root@368c4cc44221:/#                     ### 提示符变为 '#' 表明成功进入容器 后面的字符串根据容器而生成，为容器id
root@368c4cc44221:/# exit (或者CTRL+D）    ### 从容器中退出 此时运行docker ps，运行容器的列表为空

### 查看当前运行的容器
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
### 查看所有存在的容器
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
368c4cc44221        oslab:2020          "/bin/bash"         54 seconds ago      Exited (0) 30 seconds ago                       relaxed_agnesi

### 启动处于停止状态的容器
$ docker start 368c     ### 368c 为容器id的前四位，id开头的几位便可标识一个容器
$ docker ps             ### 可看到容器已经启动
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
368c4cc44221        oslab:2020          "/bin/bash"         About a minute ago   Up 16 seconds                       relaxed_agnesi

### 进入已经运行的容器 oslab的密码为2020
$ docker exec -it -u oslab -w /home/oslab 36 /bin/bash
oslab@368c4cc44221:~$

### 进入docker后，按4.2-4.5指导进行下一步实验
```

## 4.2 编译 linux 内核

```
### 进入实验目录并设置环境变量
# pwd
/home/oslab
# cd lab0
# export TOP=`pwd`
# export RISCV=/opt/riscv
# export PATH=$PATH:$RISCV/bin

# mkdir -p build/linux
# make -C linux O=$TOP/build/linux \
        CROSS_COMPILE=riscv64-unknown-linux-gnu- \
        ARCH=riscv CONFIG_DEBUG_INFO=y \
        defconfig all -j$(nproc)
```

## 4.3 使用QEMU运行内核

```
### 用户名root，没有密码
# qemu-system-riscv64 -nographic -machine virt -kernel build/linux/arch/riscv/boot/Image  \
 -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"   \
 -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
 -netdev user,id=net0 -device virtio-net-device,netdev=net0
```

## 4.4 使用 gdb 对内核进行调试

```
### Terminal 1
# qemu-system-riscv64 -nographic -machine virt -kernel build/linux/arch/riscv/boot/Image  \
```

```
 -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"   \
 -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
 -netdev user,id=net0 -device virtio-net-device,netdev=net0 -S -s

### Terminal 2
# riscv64-unknown-linux-gnu-gdb build/linux/vmlinux
(gdb) target remote localhost:1234  ### 连接 qemu
(gdb) b start_kernel                ### 设置断点
(gdb) continue                      ### 继续执行
(gdb) quit                        ### 退出 gdb
```

## 5 实验任务与要求

- 请各位同学独立完成作业，任何抄袭行为都将使本次作业判为0分。
- 编译内核并用 gdb + QEMU 调试，在内核初始化过程中（用户登录之前）设置断点，对内核的启动过程进行跟踪，并尝试使用gdb的各项命令（如backtrace、finish、frame、info、break、display、next等）。
- 在学在浙大中提交pdf格式的实验报告，记录实验过程并截图（4.1-4.4），对每一步的命令以及结果进行必要的解释，记录遇到的问题和心得体会。

**本文贡献者以及负责答疑内容**

> 周侠 (gdb qemu riscv-toolchain)
> 管章辉 (docker)
> 徐金焱 (gdb qemu riscv-toolchain)
> 张文龙 (gdb qemu riscv-toolchain)
> 刘强 孙家栋 周天昱

# lab1

# Lab 1: RV64 内核引导

## 1. 实验简介

学习RISC-V相关知识，Makefile相关知识，编写head.S实现bootloader的功能，并利用Makefile来完成对整个工程的管理。

## 2. 实验环境

- Docker in Lab0

## 3. 实验基础知识介绍

### 3.1 Bootloader介绍

BootLoader是系统加电后运行的第一段代码，它在操作系统内核运行之前运行，可以分为Booter和Loader，Booter是初始化系统硬件使之能够运行起来；Loader是建立内存空间映射图，将操作系统镜像加载到内存中，并跳转过去运行。经过Bootloader的引导加载后，系统就会退出bootloader程序，启动并运行操作系统，此后交由Linux内核接管。

Bootloader启动可分为两个阶段：

第一阶段主要包含依赖于CPU的体系结构硬件初始化的代码，通常都用汇编语言来实现。这个阶段的任务为基本的硬件设备初始化（屏蔽所有的中断、关闭处理器内部指令/数据Cache等）、设置堆栈、跳转到第二阶段的C程序入口点。本实验中对应为head.S的编写，操作相关寄存器，实现模式切换、中断关闭、程序跳转等。

第二阶段通常用C语言完成，以便实现更复杂的功能，也使程序有更好的可读性和可移植性。这个阶段的任务有：初始化本阶段要使用到的硬件设备、检测系统内存映射、为内核设置启动参数等，它为内核的运行完成所需的初始化和准备工作。

### 3.2 寄存器介绍

寄存器是计算机中最基本的概念，是中央处理器用来存放数据，地址以及指令的部件。一般而言，寄存器都有各自的功能，比如数据寄存器一般用来保存操作数和运算结果等，指针寄存器一般用来存放堆栈内存存储的偏移量等。而RISC-V中寄存器的种类以及功能详细的信息可以参考RISC-V手册以及The RISC-V Instruction Set Manual。接下来我们对实验中需要用到的几个寄存器进行介绍。

#### mstatus寄存器

mstatus寄存器，即Machine Status Register，是一种CSRs(Control and Status Registers)，其中m代表machine mode（参考3.6 Mode介绍），此寄存器中保持跟踪以及控制hart(hardware thread)的运算状态。比如mie和mip都对应mstatus上的某些bit位，所以通过对mstatus进行一下按位运算，可以实现对不同bit位的设置，从而控制不同运算状态（具体的mstatus位的布局请参考The RISC-V Instruction Set Manual 3.1.6节）。

同时注意此处的mie和mip指的是mstatus上的状态位。

#### mie以及mip寄存器

mie以及mip寄存器是Machine Interrup Registers，用来保存中断相关的一些信息，通过mstatus上mie以及mip位的设置，以及mie和mip本身两个寄存器的设置可以实现对硬件中断的控制。mie以及mip具体的布局及各个位的解释请参考The RISC-V Instruction Set Manual 3.1.9节。

#### mtvec寄存器

mtvec(即Machine Trap-Vector Base-Address Register)寄存器，主要保存machine mode下的trap vector（可理解为中断向量）的设置，其包含一个基地址以及一个mode。具体布局以及介绍请参考The RISC-V Instruction Set Manual 3.1.7节。

#### stvec寄存器

stvec(即Supervisor Trap Vector Bass Address Register)寄存器，其作用与mtvec类似，区别是保存的是supervisor mode对应的base和mode。具体布局以及介绍请参考The RISC-V Instruction Set Manual 4.1.2节。

#### sp寄存器

sp寄存器即栈顶指针寄存器，栈是程序运行中一个基本概念，栈顶指针寄存器是用来保存当前栈的栈顶地址的寄存器。

### 3.3 Makefile介绍

Makefile是一种实现关系整个工程编译规则的文件，在Lab0中我们已经使用了make工具利用Makefile文件来管理整个工程，那么什么是编译，以及Makefile的作用是什么，如何编写Makefile，请参考Makefile介绍进行学习，为之后实验步骤打好基础。

## 3.4 Linux Basic

仿照Linux Kernel的结构，下图是本次实验需要遵循的文件结构。除了3.3介绍的Makefile，图中head.S是本实验最主要的部分，承担了BootLoader的工作；main.c中包含了一个 `start_kernel` 函数，是程序最终到达的函数；test.h、test.c中包含了一个 `os_test` 函数，它是本实验的测试函数，会被 `start_kernel` 函数调用。本部分的Linux基础知识将围绕下图中其他的文件展开。

```
lab1
├── arch
│   └── riscv
│       ├── boot
│       │   └── Image
│       ├── include
│       ├── kernel
│       │   ├── head.S
│       │   ├── Makefile
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── Makefile
├── System.map
└── vmlinux
```

### 什么是vmlinux

vmlinux通常指Linux Kernel编译出的可执行文件（Executable and Linkable Format， ELF），特点是未压缩的，带调试信息和符号表的。在本实验中，vmlinux通常指将你的代码进行编译，链接后生成的可供QEMU运行的RISC-V 64-bit架构程序。如果对vmlinux使用**file**命令，你将看到如下信息：

```
$ file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, not stripped
```

### 什么是System.map

System.map是内核符号表（Kernel Symbol Table）文件，是存储了所有内核符号及其地址的一个列表。“符号”通常指的是函数名，全局变量名等等。使用 `nm vmlinux` 命令即可打印vmlinux的符号表，符号表的样例如下：

```
0000000000000800 A __vdso_rt_sigreturn
ffffffe000000000 T __init_begin
ffffffe000000000 T _sinittext
ffffffe000000000 T _start
ffffffe000000040 T _start_kernel
ffffffe000000076 t clear_bss
ffffffe000000080 t clear_bss_done
ffffffe0000000c0 t relocate
ffffffe00000017c t set_reset_devices
ffffffe000000190 t debug_kernel
```

使用System.map可以方便地读出函数或变量的地址，为Debug提供了方便。

### 什么是vmlinux.lds

GNU ld即链接器，用于将*.o文件（和库文件）链接成可执行文件。在操作系统开发中，为了指定程序的内存布局，ld使用链接脚本（Linker Script）来控制，在Linux Kernel中链接脚本被命名为vmlinux.lds。更多关于ld的介绍可以使用 `man ld` 命令。

下面给出一个vmlinux.lds的例子：

```
/* 目标架构 */
```

```
OUTPUT_ARCH( "riscv" )
/* 程序入口 */
ENTRY( _start )
/* 程序起始地址 */
BASE_ADDR = 0x80000000;
SECTIONS
{
 /* . 代表当前地址 */
 . = BASE_ADDR;
 /* code 段 */
 .text : { *(.text) }
 /* data 段 */
 .rodata : { *(.rodata) }
 .data : { *(.data) }
 .bss : { *(.bss) }
 . += 0x8000;
 /* 栈顶 */
 stack_top = .;
 /* 程序结束地址 */
 _end = .;
}
```

首先我们使用OUTPUT_ARCH指定了架构为RISC-V，之后使用ENTRY指定程序入口点为`_start`函数，程序入口点即程序启动时运行的函数，经过这样的指定后在head.S中需要编写`_start`函数，程序才能正常运行。

链接脚本中有`.` `*`两个重要的符号。单独的`.`在链接脚本代表当前地址，它有赋值、被赋值、自增等操作。而`*`有两种用法，其一是`*()`在大括号中表示将所有文件中符合括号内要求的段放置在当前位置，其二是作为通配符。

链接脚本的主体是SECTIONS部分，在这里链接脚本的工作是将程序的各个段按顺序放在各个地址上，在例子中就是从0x80000000地址开始放置了.text，.rodata，.data和.bss段。各个段的作用可以简要概括成：

| 段名 | 主要作用 |
| --- | --- |
| .text | 通常存放程序执行代码 |
| .rodata | 通常存放常量等只读数据 |
| .data | 通常存放已初始化的全局变量、静态变量 |
| .bss | 通常存放未初始化的全局变量、静态变量 |

在链接脚本中可以自定义符号，例如stack_top与_end都是我们自己定义的，其中stack_top与程序调用栈有关。

更多有关链接脚本语法可以参考这里。

#### 什么是Image

在Linux Kernel开发中，为了对vmlinux进行精简，通常使用objcopy丢弃调试信息与符号表并生成二进制文件，这就是Image。Lab0中QEMU正是使用了Image而不是vmlinux运行。

```
$ objcopy -O binary vmlinux Image --strip-all
```

此时再对Image使用**file**命令时：

```
$ file Image
image: data
```

## 3.5 Driver介绍

本部分主要是介绍QEMU模拟的 RISC-V Virt 计算机访问外部设备的方法。首先需要介绍MMIO与DTS的概念。

#### MMIO（Memory Mapped I/O）

内存映射（Memory Mapped I/O, MMIO）是RISC-V架构使用的CPU与外部设备交互的方式，与之相对的是端口映射（Port Mapped I/O, PMIO）。MMIO使用了相同的地址总线来处理内存与外部I/O设备，外部I/O设备的内存与寄存器被映射到内存空间特定的地址。当CPU需要访问某个外部设备时，MMIO使其只需要访问某个地址即可，即使用内存访问指令（sd，ld等）可以读写外部设备。本实验的NS16550A串口和之后实验的时钟寄存器都需要使用MMIO访问。

#### DTS（Device Tree Source）

设备树源码（Device Tree Source, DTS），Device Tree是一种描述硬件的数据结构，由一系列被命名的结点（node）和属性（property）组成，其中结点本身可包含子结点，属性是成对的name和value。简单说，Device Tree是一棵电路板上CPU、总线、设备组成的树，Bootloader会将这棵树传递给内核，然后内核可以识别这棵树。给定正确的设备树，相同的编译内核可以支持更广泛的体系结构系列中的不同硬件配置。

#### NS16550A

简要的说，RISC-V使用MMIO（Memory Mapped I/O）技术，将外设映射到各个物理地址，通过访问物理地址来操作外设。在virt.c的virt_memmap[]中有各个外设的物理地址，位于设备树上。本次实验只需要关注**0x10000000**地址即NS16550A串口，通过向NS16550A串口写入字符我们可以将它们打印出来。

在test.c中有一个`putChar`函数使用了串口，打印所需的字符串。这只是一个最简单的版本，最终我们可以利用这个串口实现`printf`函数。

### 3.6 Mode介绍

RISC-V有三个特权模式：U（user）模式、S（supervisor）模式和M（machine）模式。它通过设置不同的特权级别模式来管理系统资源的使用。其中M模式是最高级别，该模式下的操作被认为是安全可信的，主要为对硬件的操作；U模式是最低级别，该模式主要执行用户程序，操作系统中对应于用户态；S模式介于M模式和U模式之间，操作系统中对应于内核态，当用户需要内核资源时，向内核申请，并切换到内核态进行处理。

| Level | Encoding | Name | Abbreviation |
|---|---|---|---|
| 0 | 00 | User/Application | U |
| 1 | 01 | Supervisor | S |
| 2 | 10 | Reserved | |
| 3 | 11 | Machine | M |

## 4. 实验步骤

### 4.1 搭建实验环境

实验环境仍为docker镜像，请下载最新镜像，按以下方法创建新的容器，并建立volume映射(参考资料)，在本地编写代码，并在容器内进行编译检查。

```
### 首先新建自己本地的工作目录(比如lab1)并进入
$ mkdir lab1
$ cd lab1
$ pwd
~/.../lab1

### 查看docker已有镜像(与lab0同一个image)
$ docker image ls
REPOSITORY        TAG             IMAGE ID         CREATED          SIZE
oslab             2020            678605140682     46 hours ago     2.89GB

### 创建新的容器，同时建立volume映射
$ docker run -it -v `pwd`:/home/oslab/lab1 -u oslab -w /home/oslab 6786 /bin/bash
oslab@3c1da3906541:~$


### 测试映射关系是否成功，新开一个shell并进入之前~/.../lab1目录下
$ touch testfile
$ ls
testfile

### 在docker中确认是否挂载成功
oslab@3c1da3906541:~$ pwd
/home/oslab
oslab@3c1da3906541:~$ cd lab1
oslab@3c1da3906541:~$ ls
testfile

### 确认映射关系建立成功后在本地lab1目录下继续实验
```

## 4.2 编写Makefile

在了解了make的使用以及Makefile的基本知识之后（参考**3.3** Makefile介绍），并参照**3.4**节文件结构介绍。实验给出以下文件(https://gitee.com/zjuicsr/lab20fall-stu)

```
main.c
test.h
test.c
vmlinux.lds
```

### 4.2.1 组织文件结构

请参照3.4节文件结构，并组织形成对应的目录结构。其中main.c,test.c,test.h以及vmlinux.lds已经给出，在对应目录下添加空的Makefile文件，并添加空的head.S文件，形成如下目录结构，并确认容器中对应目录下同步成功。

```
lab1
├── arch
│   └── riscv
│       ├── boot
│       ├── include
│       ├── kernel
│       │   ├── head.S
│       │   ├── Makefile
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
└── Makefile
```

### 4.2.2 编写各级目录Makefile

需要你书写各级目录下的Makefile，用来控制整个工程，需要实现的功能有：

- 能通过lab1目录（即工程中顶级目录）来实现Makefile的层级调用。
- 能够通过(main.c，vmlinux.lds，以及head.S)来生成vmlinux, System.map, Image等文件到相应位置。
- 实现make clean来清除文件，实现make run来直接运行(参考**4.4**)。

基本思路如下（为了描述方便，文档中Makefile_dir表示dir目录下的Makefile）：首先清楚Makefile的层级调用关系如下，其中顶级目录下的Makefile_lab1控制Makefile_init和Makefile_riscv，Makefile_riscv控制Makefile_kernel。

```
lab1
├── arch
│   └── riscv
│       ├── kernel
│       │   ├── Makefile_kernel
│       └── Makefile_riscv
├── init
│   ├── Makefile_init
└── Makefile_lab1
```

接下来每个Makefile需要实现的功能如下：

- Makefile_lab1：设置编译需要的变量，并利用make -C调用其需要控制的其他Makefile
- Makefile_init：利用main.c和test.c生成main.o和test.o
- Makefile_kernel：利用head.S生成head.o
- Makefile_riscv：使用ld并将main.o，test.o，head.o等目标文件以及lds文件（利用ld的-T选项）生成vmlinux文件到指定目录下，并使用OBJCOPY利用vmlinux生成Image文件到指定目录。

其中顶层Makefile中的变量可以参考以下设置：

```
export
CROSS_= riscv64-unknown-elf-
AR=${CROSS_}ar
GCC=${CROSS_}gcc
LD=${CROSS_}ld
OBJCOPY=${CROSS_}objcopy

ISA ?= rv64imafd
ABI ?= lp64

INCLUDE = -I ../include
CF =  -O3 -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nr
CFLAG = ${CF} ${INCLUDE}
```

### 4.3 编写head.S

#### 4.3.1 学习汇编指令

请参考RISC-V手册以及The RISC-V Instruction Set Manual来学习RISC-V的基础指令，为编写head.S文件打好基础。

#### 4.3.2 学习mode切换

请参考RISC-V手册第十章，学习如何设置mstatus,mepc并利用mret指令可以使得CPU从machine mode切换到supervisor mode。

#### 4.3.3 完成head.S

编写head.S，并且实现以下功能：

- 关闭machine mode中断
- 设置machine mode异常处理地址
- 使cpu从machine mode切换到supervisor mode
- 设置supervisor mode异常处理地址
- 设置c语言调用栈环境
- 跳转到main.c中给出的start_kernel函数

注：需要实现从supervisor mode跳转到start_kernel函数。

### 4.4 编译及测试

利用make调用Makefile文件完成整个工程的编译过程，利用qemu启动kernel（参考lab0），如果kernel能正常运行，并成功执行了start_kernel中的函数（即打印字符串），那么实验成功。

```
### 此时应提前设置好path等环境变量（export PATH=$PATH:/opt/riscv/bin）
# QEMU RUN (即make run)
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
# QEMU DEBUG (可选配置为make debug)
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -S -s
```

预期的实验结果：

```
oslab@3c1da3906541:~/lab1$ make run
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid breakages when this happe
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
Hello RISC-V!
```

## 5. 实验任务与要求

请学习基础知识，并按照实验步骤指导完成实验，撰写实验报告。实验报告的要求：

- 各实验步骤的截图以及结果分析
- 实验结束后的心得体会
- 对实验指导的建议（可选）

### 本文贡献者以及负责答疑内容

> 庄阿得(bootloader,Mode)
> 王琨(寄存器,Makefile,实验步骤)
> 沈韬立(Linux Basic,Driver)

# lab2

# Lab 2: RV64 时钟中断处理

## 1 实验简介

学习在RISC-V上的异常处理相关机制，以时钟中断为例，编写时钟中断处理函数。

## 2 实验环境

- Docker Image

## 3 背景知识

### 3.1 什么是异常(trap)

异常(trap)是指是不寻常的运行时事件，由硬件或软件产生，当异常产生时控制权将会转移至异常处理程序。异常是操作系统最基础的概念，一个没有异常的操作系统无法进行正常交互。

RISC-V将异常分为两类。一类是硬件中断(interrupt)，它是与指令流异步的外部事件，比如鼠标的单击。另外一类是同步异常(exception)，这类异常在指令执行期间产生，如访问了无效的存储器地址或执行了具有无效操作码的指令时。

这里我们用异常(trap)作为硬件中断(interrupt)和同步异常(exception)的集合，另外trap指的是发生硬件中断或者同步异常时控制权转移到handler的过程。

> 后文统一用异常指代trap，中断/硬件中断指代interrupt，同步异常指代exception

### 3.2 Machine Mode下的异常

#### 3.2.1 异常需要的寄存器

Machine mode异常需要使用的寄存器首先有lab1提到的mstatus，mip，mie，mtvec寄存器，这些寄存器需要我们操作；剩下还有mepc，mcause寄存器，这些寄存器在异常发生时硬件会自动置位，它们的功能如下：

- mepc（Machine Exception Program Counter）：通常指向异常处理后应该恢复执行的位置
- mcause（Machine Cause Register）：保存异常的种类，具体可以查看请参考The RISC-V Instruction Set Manual 3.1.16节（P.40）

事实上，异常还与mideleg和medeleg两个寄存器密切相关，它们的功能将在**Supervisor mode下的异常**部分讲解。

与时钟中断相关的还有mtime和mtimecmp寄存器，它们的功能如下：

- mtime（Machine Time Register）：保存时钟计数，这个值会由硬件自增
- mtimecmp（Machine Time Compare Register）：保存需要比较的时钟计数，当mtime的值大于或等于mtimecmp的值时，时钟中断触发

需要注意的是，mtime和mtimecmp寄存器需要用MMIO的方式即使用内存访问指令（sd，ld等）的方式交互，可以将它们理解为Machine mode下的一个外设。

#### 3.2.2 硬件中断的处理（以时钟中断为例）

简单地来说，中断处理经过了三个流程，中断触发，判断可以处理还是忽略，可以处理时调用处理函数。

1. 中断触发：

   时钟中断的触发条件是这个hart（硬件线程）的时间比较器mtimecmp小于实数计数器mtime。

2. 判断是否可处理：

   当时钟中断触发时，并不一定会响应中断信号。Machine mode只有在全局中断使能位mstatus[mie]置位时才会产生中断，如果在Supervisor mode下触发了Machine mode的中断，此时无视mstatus[mie]直接响应，即运行在低权限模式下，高权限模式的全局中断使能位一直是enable状态。此外，每个中断在控制状态寄存器mie中都有自己的使能位，对于特定中断来说，需要考虑自己对应的使能位，而控制状态寄存器mip中又指示目前待处理的中断。以时钟中断为例，只有当mstatus[mie]=1，mie[mtie]=1，且mip[mtip]=1时，才可以处理机器的时钟中断。其中mstatus[mie]以及mie[mtie]需要我们自己设置，而mip[mtip]在中断触发时会被硬件自动置位。

3. 调用处理函数：

当满足对应中断的处理条件时，硬件首先会发生一些状态转换（参考中文手册10.3节），并跳转到对应的异常处理函数中，在异常处理函数中我们可以通过分析异常产生的原因判断具体为哪一种，然后执行对应的处理。为了处理异常结束后不影响hart正常的运行状态，我们首先需要保存当前的状态即上下文切换。我们可以先用栈上的一段空间来把**全部**寄存器保存，保存完之后执行到我们编写的异常处理函数主体，结束后退出。函数调用规范可以参考中文开源手册3.2节。

#### 3.2.3 同步异常的处理

同步异常的触发条件是当前指令执行了未经定义的行为，比如Illegal instruction，也没有判断可以处理还是忽略的步骤，硬件会直接经历一些状态转换，然后跳到对应的异常处理函数。又比如环境调用同步异常ecall主要是用于低权限的mode需要高权限的mode的相关操作时使用的，比如U-mode call S-mode 通常用于使用系统调用，而S-mode call M-mode通常用于在S-mode操作某些硬件，这在本实验会用到。

需要注意的是，不管是中断还是同步异常，都会经历相似的硬件状态转换，并跳到同一个异常处理地址（由mtvec/stvec指定），再由处理函数分析异常出现原因并进行不同的处理。

### 3.3 Supervisor Mode下的异常

由于hart位于Supervisor mode，我们需要在Supervisor mode下处理异常。这时首先要提到委托（delegation）机制。

#### 3.3.1 委托（delegation）

**RISC-V架构所有mode的异常在默认情况下都跳转到Machine mode处理**。为了提高性能，RISC-V支持将低权限mode产生的异常委托给对应mode处理。这里使用了mideleg和medeleg两个寄存器，它们的功能如下：（更具体的介绍参考The RISC-V Instruction Set Manual 3.1.8节）

- mideleg（Machine Interrupt Delegation）：控制将哪些中断委托给对应mode处理，它的结构可以参考mip寄存器，比如将mip中stip对应的位置位会将Supervisor mode的时钟中断委托给Supervisor mode处理
- medeleg（Machine Exception Delegation）：控制将哪些同步异常委托给对应mode处理，它的各个位对应mcause寄存器的返回值

#### 3.3.2 Supervisor Mode下时钟中断处理流程

事实上，虽然在mideleg中设置了将Supervisor mode产生的时钟中断委托给Supervisor mode，委托并没有完成。因为硬件产生的时钟中断仍会发到Machine mode（mtime寄存器是Machine mode的设备），所以我们需要手动触发Supervisor mode下的时钟中断。

此前，假设设置好[m|s]status以及[m|s]ie，即我们已经满足了时钟中断在两种mode下触发的使能条件。接下来一个时钟中断的委托流程如下：

1. 当mtimecmp小于mtime时，触发时钟中断并且硬件自动置位mip[mtip]。
2. 此时mstatus[mie]=1，mie[mtie]=1，且mip[mtip]=1 表示可以处理machine mode的时钟中断。
3. 此时hart发生了异常，硬件会自动经历状态转换，其中pc被设置被mtvec，即跳转到我们设置好的machine mode处理函数入口。
4. machine mode处理函数分析异常原因，判断为时钟中断，为了将时钟中断委托给supervisor mode，于是将mip[stip]置位，并且为了防止在supervisor mode处理时钟中断时继续触发machine mode时钟中断，于是同时将mie[mtie]清零。
5. machine mode处理函数处理完成并退出，此时sstatus[sie]=1，sie[stie]=1，且sip[stip]=1(由于sip是mip的子集，所以第4步中令mip[stip]置位等同于将sip[stip]置位)，于是触发supervisor mode的时钟中断。
6. 此时hart发生了异常，硬件自动经历状态转换，其中pc被设置为stvec，即跳转到我们设置好的supervisor mode处理函数入口。
7. supervisor mode处理函数分析异常原因，判断为时钟中断，于是进行相应的操作，然后利用ecall触发异常，跳转到machine mode的异常处理函数进行最后的收尾。
8. machine mode异常处理函数分析异常原因，发现为ecall from S-mode，于是设置mtimecmp+=100000，将mip[stip]清零，表示supervisor mode时钟中断处理完毕，并且设置mie[mtie]恢复machine mode的中断使能，保证下一次时钟中断可以触发。
9. 函数逐级返回，整个委托的时钟中断处理完毕。

## 4 实验步骤

### 4.1 环境搭建

#### 4.1.1 建立映射

同lab1的文件夹映射方法，目录名为lab2。

#### 4.1.2 组织文件结构

文件结构如下：

```
lab2
├── arch
│   └── riscv
│       ├── include
```

```
│   │   └── put.h
│   ├── kernel
│   │   ├── entry.S
│   │   ├── head.S
│   │   ├── Makefile
│   │   ├── strap.c
│   │   └── vmlinux.lds
│   └── Makefile
├── include
│   ├── put.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── put.c
└── Makefile
```

其中put.h,put.c,test.h,test.c,main.c会提供给同学们。将lab1中实现的Makefile放置到对应目录下，并在lib目录新建Makefile，将lib目录下的文件纳入到整个编译的工程管理中。

#### 4.1.3 文件必要修改

由于裸机程序需要在.text段起始位置执行，所以需要利用vmlinux.lds中.text段的定义来确保head.S中的.text段被放置在其他.text段之前。具体的做法如下：

1. 首先修改head.S中的.text命名为.text.init

```
<<<<< before
.section .text
============
.section .text.init
>>>>> after
```

2. 修改entry.S中的.text命名为.text.entry

```
<<<<< before
.section .text
============
.section .text.entry
>>>>> after
```

3. 修改lds文件中的.text展开方式

```
<<<<< before
.text : { *(.text) }
============
.text : {
      *(.text.init)
      *(.text.entry)
      *(.text)
}
>>>>> after
```

### 4.2 head.S模式切换前添加功能

#### 4.2.1 初始化bss段

确认vmlinux.lds中有对bss段的定义，在head.S模式切换之前初始化bss段的内存。

> 思考题：观察vmlinux和image，解释为什么要初始化bss段。

#### 4.2.2 初始化mtimecmp寄存器

将mtimecmp寄存器的值初始化为mtime+1000000。其中mtime对应的映射地址是0x200bff8，mtimecmp对应的映射地址是0x2004000。参考这里（内网访问）

#### 4.2.3 设置时钟中断委托

首先需要设置mideleg对应位来打开时钟中断的委托，除此以外需要设置[m|s]status以及[m|s]ie来打开时钟中断的使能（具体为设置mstatus[mie],sstatus[sie],mie[mtie],sie[stie]）。

### 4.3 编写machine mode的异常处理代码

#### 4.3.1 上下文切换

lab1中我们设置mtvec指向一个空的trap_m函数。在此trap_m函数中，首先我们需要利用sp开辟一段空间，并保存所有寄存器以及必要的CSRs的值，并且在mret之前恢复所有寄存器的值，并设置sp回收空间。

#### 4.3.2 编写处理代码

保存寄存器之后，我们需要编写对应的异常处理代码，这里我们只需处理两种异常。一种是时钟中断，另外一种是ecall from S-mode。

在处理时钟中断时，需要完成以下功能：

- disable mie[mtie]，即禁用时钟中断，避免之后Supervisor mode处理时钟中断同时继续触发时钟中断。
- enable sip[stip]，设置Supervisor mode timer interrupt的pending位，为之后触发Supervisor mode下的时钟中断做准备。

在处理ecall from S-mode时，需要完成以下功能：

- set mtimecmp += 100000 此时设置mtimecmp同时硬件会clear mip[mtip]
- enable mie[mtie] 恢复machine mode的时钟中断使能
- disable mip[stip] 清除Supervisor mode下的时钟中断Pending位

> 思考题：当处理同步异常时应该在退出前给mepc+4，当处理中断时则不需要，请解释为什么要这样做。

### 4.4 编写Supervisor mode的异常处理代码entry.S

#### 4.4.1 上下文切换

将head.S中的trap_s函数（即Supervisor mode的异常处理函数）移动到entry.S中，并首先保存所有寄存器以及必要的CSRs的值。

#### 4.4.2 异常处理

对异常进行处理，仅需实现对时钟中断的处理，功能是输出已经产生的时钟中断的个数。（为了方便测试，请每产生100个时钟中断输出一行显示信息）

### 4.5 编译及测试

仿照lab1进行调试，预期的实验结果如下：(请对test.c做修改，确保输出自己的组号，例第4组修改XX为04)

```
ZJU OS LAB 2           GROUP-XX
[S] Supervisor Mode Timer Interrupt 0
[S] Supervisor Mode Timer Interrupt 1
[S] Supervisor Mode Timer Interrupt 2
[S] Supervisor Mode Timer Interrupt 3
[S] Supervisor Mode Timer Interrupt 4
[S] Supervisor Mode Timer Interrupt 5
[S] Supervisor Mode Timer Interrupt 6
[S] Supervisor Mode Timer Interrupt 7
[S] Supervisor Mode Timer Interrupt 8
[S] Supervisor Mode Timer Interrupt 9
[S] Supervisor Mode Timer Interrupt 10
[S] Supervisor Mode Timer Interrupt 11
[S] Supervisor Mode Timer Interrupt 12
[S] Supervisor Mode Timer Interrupt 13
[S] Supervisor Mode Timer Interrupt 14
[S] Supervisor Mode Timer Interrupt 15
[S] Supervisor Mode Timer Interrupt 16
[S] Supervisor Mode Timer Interrupt 17
[S] Supervisor Mode Timer Interrupt 18
```

```
[S] Supervisor Mode Timer Interrupt 19
[S] Supervisor Mode Timer Interrupt 20
[S] Supervisor Mode Timer Interrupt 21
[S] Supervisor Mode Timer Interrupt 22
[S] Supervisor Mode Timer Interrupt 23
[S] Supervisor Mode Timer Interrupt 24
```

> Hint：如果觉得直接完成实验困难，可以先完成machine mode下的时钟中断处理，再完成supervisor mode下的时钟中断处理。

## 5 实验任务与要求

请仔细阅读背景知识，确保理解RISC-V异常委托与异常处理机制，并按照实验步骤完成实验，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 实验报告：
  - 各实验步骤截图以及结果分析
  - 回答思考题
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（确保make clean）
- 最终目录
  - 将Lab2_319010XXXX目录压缩并打包（若分组，则一组只需要一位同学提交）

```
Lab2_319010XXXX
├── report
│   └── 319010XXXX.pdf（若分组，所有组员学号在报告内部声明）
└── source
    ├── arch
    │   └── riscv
    │       ├── include
    │       │   └── put.h
    │       ├── kernel
    │       │   ├── entry.S
    │       │   ├── head.S
    │       │   ├── Makefile
    │       │   ├── strap.c
    │       │   └── vmlinux.lds
    │       └── Makefile
    ├── include
    │   ├── put.h
    │   └── test.h
    ├── init
    │   ├── main.c
    │   ├── Makefile
    │   └── test.c
    ├── lib
    │   ├── Makefile
    │   └── put.c
    └── Makefile
```

**本文贡献者**

> 王琨（背景知识，实验步骤）
> 沈韬立（背景知识，实验步骤）

# lab3

# 实验 3：RISC-V64简单的进程调度实现

## 1. 实验简介

- 结合课堂所学习的相关内容，在上一实验实现中断的基础上进一步地实现简单的进程调度

## 2. 实验环境

- Docker Image

## 3. 背景知识

## 3.1 什么是进程

`源代码` 经编译器一系列处理（编译、链接、优化等）后得到的可执行文件，我们称之为 `程序（Program）` 。而通俗地说， `进程（Process）` 就是 `正在运行并使用计算机资源` 的程序。 `进程` 与 `程序` 的不同之处在于，进程是一个动态的概念，其不仅需要将其运行的程序的代码/数据等加载到内存空间中，还需要拥有自己的 `运行栈` 。

## 3.2 进程的表示

在不同的操作系统中，为每个进程所保存的信息都不同。在这里，我们提供一种基础的实现，每个进程会包括：

- `进程ID` ：用于唯一确认一个进程。
- `运行时间片` ：为每个进程分配的运行时间。
- `优先级` ：在调度时，配合调度算法，来选出下一个执行的进程。
- `运行栈` ：每个进程都必须有一个独立的运行栈，保存运行时的数据。
- `执行上下文` ：当进程不在执行状态时，我们需要保存其上下文（其实就是 `状态寄存器` 的值），这样之后才能够将其恢复，继续运行。

## 3.3 进程调度与切换的过程

- 在每次时钟中断处理时，操作系统首先会将当前进程的运行剩余时间减少一个单位。之后根据调度算法来确定是继续运行还是调度其他进程来执行。
- 在进程调度时，操作系统会对所有可运行的进程进行判断，按照一定规则选出下一个执行的进程。如果没有符合条件的进程，则会对所有进程的优先级和运行剩余时间相关属性进行更新，再重新选择。最终将选择得到的进程与当前进程切换。
- 在切换的过程中，首先我们需要保存当前进程的执行上下文，再将将要执行进程的上下文载入到相关寄存器中，至此我们完成了进程的调度与切换。

## 4. 实验步骤

### 4.1 环境搭建

#### 4.1.1 建立映射

同lab2的文件夹映射方法，目录名为lab3。

#### 4.1.2 组织文件结构

```
lab3
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   └── sched.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
```

```
│   │   └── vmlinux.lds
│   └── Makefile
├── include
│   ├── put.h
│   ├── rand.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
└── Makefile
```

在之前的实验代码的基础上，提供`rand.h`，`rand.c`，`sched.h`给同学们。将lab2中实现的Makefile放置到对应目录下，并在修改相关Makefile，将新增文件纳入到整个编译的工程管理中。

## 4.2 rand.h rand.c介绍

- `rand.h`中定义`SEED`。
- `rand.c`中实现了一个随机数迭代器，（产生一系列随机数，用来初始化和更新进程的运行时间块与优先级。保证SEED相同时随机数序列是一样的）。

## 4.3 sched.h数据结构定义

```
#ifndef _SCHED_H
#define _SCHED_H

#define TASK_SIZE  (4096)
#define THREAD_OFFSET  (5 * 0x08)

#ifndef __ASSEMBLER__

/* task的最大数量 */
#define NR_TASKS    64

#define FIRST_TASK  (task[0])
#define LAST_TASK   (task[NR_TASKS-1])

/* 定义task的状态，Lab3中task只需要一种状态。*/
#define TASK_RUNNING              0
// #define TASK_INTERRUPTIBLE      1
// #define TASK_UNINTERRUPTIBLE    2
// #define TASK_ZOMBIE             3
// #define TASK_STOPPED            4

#define PREEMPT_ENABLE  0
#define PREEMPT_DISABLE 1

/* Lab3中进程的数量以及每个进程初始的时间片 */
#define LAB_TEST_NUM        4
#define LAB_TEST_COUNTER    5

/* 当前进程 */
extern struct task_struct *current;

/* 进程指针数组 */
extern struct task_struct * task[NR_TASKS];

/* 进程状态段数据结构 */
struct thread_struct {
    unsigned long long ra;
    unsigned long long sp;
    unsigned long long s0;
    unsigned long long s1;
    unsigned long long s2;
```

```
    unsigned long long s3;
    unsigned long long s4;
    unsigned long long s5;
    unsigned long long s6;
    unsigned long long s7;
    unsigned long long s8;
    unsigned long long s9;
    unsigned long long s10;
    unsigned long long s11;
};

/* 进程数据结构 */
struct task_struct {
     long state;    // 进程状态 Lab3中进程初始化时置为TASK_RUNNING
     long counter;  // 运行剩余时间
     long priority; // 运行优先级 1最高 5最低
     long blocked;
     long pid;      // 进程标识符
    // Above Size Cost: 40 bytes

    struct thread_struct thread; // 该进程状态段
};

/* 进程初始化 创建四个dead_loop进程 */
void task_init(void);

/* 在时钟中断处理中被调用 */
void do_timer(void);

/* 调度程序 */
void schedule(void);

/* 切换当前任务current到下一个任务next */
void switch_to(struct task_struct* next);

/* 死循环 */
void dead_loop(void);

#endif

#endif
```

## 4.4 sched.c进程调度功能实现

### 4.4.1 在中断处理中添加保存epc的指令

当**异常**发生时，**epc寄存器**会保存当前的异常返回地址，该地址是与**当前task相关**的。由于task会在异常处理中发生调度，所以我们需要在中断处理时保存当前的epc，当中断处理完成之后再载入**epc**。

### 4.4.2 实现 task_init()

- 本实验中，我们使用了多个进程，需要对物理内存区域进行划分。此次实验中我们手动做内存分配，把物理地址空间划分成多个帧(frame)。即，从0x80010000地址开始，连续地给此次实验的 4 个 Task [1-4]以及内核栈 Task[0]做内存分配，我们以 4KB 为粒度，按照每个 task 一帧的形式进行分配。 **（请同学按照下图的内存空间分配地址，不要随意修改，否则有可能影响到最终的实验结果）**
- 为方便起见，我们要求Task [1-4] 进程均为dead_loop。
- 可将Task [1-4] 看作是从 0 号进程(Task [0] ) fork 出的子进程，后续实验的实现将主要考虑如何对这四个子进程进行调度。

```
-------------------------------------------------------------------------
|        |  Task0  |  Task1  |  Task2  |  Task3  |  Task4  |
|  Kernel  |       |       |       |       |       |
|        |  Space  |  Space  |  Space  |  Space  |  Space  |
-------------------------------------------------------------------------
^        ^       ^       ^       ^       ^
0x80000000  0x80010000 0x80011000 0x80012000 0x80013000 0x80014000
```

- 初始化 `current` 与 `task[0]`

- 设置 current 指向 Task0 Space 的基地址。
- 设置 task[0] 为 current 。
- 初始化 task[0] 中的成员变量

```
state    = TASK_RUNNING
counter  = 0
priority = 5
blocked  = 0
pid      = 0
```

- 设置 task[0] 的 thread 中的 sp 指针为 Task0 Space基地址 + 4KB的偏移。

- 参照 task[0] 的设置，对 task[1-4] 完成初始化设置
  - 在初始化时，我们可以将 task[1-4] 的 thread_struct 中的 ra 指向一个初始化函数，在函数中我们将 dead_loop 赋值给 epc 寄存器。
  - 对于短作业优先非抢占式算法

```
counter = rand()（task[1-4]的初始剩余运行时间均采用rand()得到）
priority = 5（task[1-4]初始优先级均为5）
```

  - 对于优先级抢占式算法

```
counter = 7, 6, 5, 4（分别对应 task[1-4]的运行时长）
priority = 5（task[1-4]初始优先级均为5）
```

#### 4.4.3 实现 do_timer()

- 将当前所运行进程的运行剩余时间减少一个单位（ counter-- ）
- 短作业优先非抢占式
  - 如果当前进程运行剩余时间已经用完，则进行调度，选择新的进程来运行，否则继续执行当前进程。
- 优先级抢占式算法
  - 每次do_timer都进行一次抢占试优先级调度。当在do_timer中发现当前运行进程剩余运行时间为0（即当前进程已运行结束）时，需重新为该进程分配其对应的运行时长。相当于重启当前进程，即重新设置每个进程的运行时间长度和初始化的值一致。

#### 4.4.4 实现 schedule()

本次实验我们需要实现两种调度算法：1.短作业优先非抢占式算法，2.优先级抢占式算法。

##### 4.4.4.1 短作业优先非抢占式算法

- 当需要进行调度时按照一下规则进行调度：
  - 遍历进程指针数组 task ，从 LAST_TASK 至 FIRST_TASK (不包括 FIRST_TASK ，即Task[0])，在所有运行状态(TASK_RUNNING)下的进程运行剩余时间 最小 的进程作为下一个执行的进程。
  - 如果所有运行状态下的进程运行剩余时间都为0，则对这些进程的运行剩余时间重新随机赋值（ 以模拟有新的不同运行时间长度的任务生成 ），之后再重新进行调度。

##### 4.4.4.2 优先级抢占式算法

- 遍历进程指针数组 task ，从 LAST_TASK 至 FIRST_TASK (不包括 FIRST_TASK )，调度规则如下：
  - 高优先级 的进程，优先被运行。
  - 优先级相同 ，则选择 运行剩余时间少 的进程（若运行剩余时间也相同，则遍历的顺序优先选择）。
- 每次schedule，实现随机更新Task[1-4]进程的priority = rand()（ 模拟动态优先级变化 ）

#### 4.4.5 实现 switch_to(struct task_struct* next)

- 判断下一个执行的进程 next 与当前的进程 current 是否为同一个进程，如果是同一个进程，则无需做任何处理。
- 实现切换进程的过程
  - 保存当前进程的 ra， sp， s0~s11 到当前进程的 进程状态段（thread) 中。
  - 将下一个进程的 进程状态段(thread) 的相关数据载入到 ra， sp， s0~s11 中。

### 4.5 编译及测试

仿照lab2进行调试，预期的实验结果如下：(请对test.c做修改，确保输出自己的组号，例第4组修改XX为04)

- 短作业优先非抢占式

```
ZJU OS LAB 3            GROUP-XX
task init...
[PID = 1] Process Create Successfully! counter = 1
[PID = 2] Process Create Successfully! counter = 4
[PID = 3] Process Create Successfully! counter = 5
[PID = 4] Process Create Successfully! counter = 4
[PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 to task 1, prio: 5, counter: 1 // 由于task[1]的剩余运行时间最少，故选择task[1]
[PID = 1] Context Calculation: counter = 1
[!] Switch from task 1 to task 4, prio: 5, counter: 4 // task[1]运行完成之后，当前task[2, 4]的剩余运行时间最少，按照遍历的顺序（由后向前）选择task[4]
[PID = 4] Context Calculation: counter = 4
[PID = 4] Context Calculation: counter = 3
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 to task 2, prio: 5, counter: 4 // 选择task[2]
[PID = 2] Context Calculation: counter = 4
[PID = 2] Context Calculation: counter = 3
[PID = 2] Context Calculation: counter = 2
[PID = 2] Context Calculation: counter = 1
[!] Switch from task 2 to task 3, prio: 5, counter: 5 // 选择task[3]
[PID = 3] Context Calculation: counter = 5
[PID = 3] Context Calculation: counter = 4
[PID = 3] Context Calculation: counter = 3
[PID = 3] Context Calculation: counter = 2
[PID = 3] Context Calculation: counter = 1
//当task[1-4]都运行完成，重置task[1-4]的counter
[PID = 1] Reset counter = 5
[PID = 2] Reset counter = 5
[PID = 3] Reset counter = 5
[PID = 4] Reset counter = 2
[!] Switch from task 3 to task 4, prio: 5, counter: 2
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[!] Switch from task 4 to task 3, prio: 5, counter: 5
[PID = 3] Context Calculation: counter = 5
[PID = 3] Context Calculation: counter = 4
...
```

- 优先级抢占式算法

```
ZJU OS LAB 3            GROUP-XX
[PID = 1] Process Create Successfully! counter = 7 priority = 5
[PID = 2] Process Create Successfully! counter = 6 priority = 5
[PID = 3] Process Create Successfully! counter = 5 priority = 5
[PID = 4] Process Create Successfully! counter = 4 priority = 5
[!] Switch from task 0 to task 4, prio: 5, counter: 4 // 此时所有进程的优先级相同，选择剩余运行时间最少的 task[4]
tasks' priority changed
[PID = 1] counter = 7 priority = 1 // 此时task[1]的优先级最高
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 4 priority = 4
[!] Switch from task 4 to task 1, prio: 1, counter: 7
tasks' priority changed
[PID = 1] counter = 7 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 3 priority = 2 // 此时task[4]的优先级最高
[!] Switch from task 1 to task 4, prio: 2, counter: 3
tasks' priority changed
[PID = 1] counter = 6 priority = 4
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 4 // 此时task[1 2 3]的优先级最高且相同，由于task[3]的剩余运行时间最少，故选择task[3]
[PID = 4] counter = 3 priority = 5
[!] Switch from task 4 to task 3, prio: 4, counter: 5
tasks' priority changed
[PID = 1] counter = 6 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 4
[PID = 4] counter = 2 priority = 2
```

```
[!] Switch from task 3 to task 4, prio: 2, counter: 2
tasks' priority changed
[PID = 1] counter = 6 priority = 5
[PID = 2] counter = 6 priority = 3
[PID = 3] counter = 4 priority = 3 // 此时task[2 3]的优先级最高且相同，由于task[3]的剩余运行时间最少，故选择task[3]
[PID = 4] counter = 2 priority = 4
[!] Switch from task 4 to task 3, prio: 3, counter: 4
...
```

## 5. 实验任务与要求

请仔细阅读背景知识，确保理解进程调度与进程切换过程，并按照实验步骤完成实验，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 由于本次实验需要完成两个调度算法，因此需要两种调度算法可以使用`gcc -D`选项进行控制。
  - -DSJF （短作业优先式）。
  - -DPRIORITY （优先级抢占式）。
    - 在sched.c中使用`#ifdef`，`#endif`语句来控制进程调度的代码实现。
  - 修改`lab3/Makefile`中的`CFLAG = ${CF} ${INCLUDE} -DSJF / -DPRIORITY`(作业提交的时候这个随机确定一个即可。)
- 实验报告：
  - 各实验步骤截图以及结果分析
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（确保make clean）
- 最终目录
  - 将Lab3_319010XXXX目录压缩并打包（若分组，则一组只需要一位同学提交）

```
Lab3_319010XXXX
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   └── sched.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   ├── rand.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
├── Makefile
└── report
    └── 319010XXXX.pdf
```

**本文贡献者**

> 王星宇（背景知识，实验步骤 wangxingyu@zju.edu.cn）
> 张文龙（背景知识，实验步骤 2968829696@qq.com）

# lab4

# Lab 4: RISC-V 虚拟内存管理

## 1. 实验简介

- 结合课堂学习的页式内存管理以及虚拟内存的相关知识，尝试在已有的程序上开启 MMU 并实现页映射，保证之前的进程调度能在虚拟内存下正常运行

## 2. 实验环境

- 在实验 3 的基础上进行（Docker Image）

## 3. 背景知识

### 3.1 虚拟内存

MMU（Memory Management Unit），负责`虚拟地址`到`物理地址`的转换。程序在cpu上运行时，他使用的虚拟地址会由MMU进行翻译。为了加速地址翻译的过程，现代cpu都引入了TLB（Translation Lookaside Buffer）。 分页机制的基本思想是将程序的虚拟地址空间划分为连续的，等长的虚拟页。`虚拟页和物理页的页长固定且相等`（一般情况下为4KB），从而操作系统可以方便的为每个程序构造页表，即虚拟页到物理页的映射关系。

逻辑上，该机制下的虚拟地址有两个部分组成：`1.虚拟页号`；`2.页内偏移`；在具体的翻译过程中，MMU首先解析得到虚拟地址中的虚拟页号，并通过虚拟页号查找到对应的物理页，最终用该物理页的起始地址加上页内偏移得到最终的物理地址。

### 3.2 RISC-V Virtual-Memory System (Sv39)

#### 3.2.1 RISC-V `satp` Register（Supervisor Address Translation and Protection Register）

```
 63    60 59            44 43                    0
 ------------------------------------------------------------------------
|  MODE  |       ASID       |          PPN          |
 ------------------------------------------------------------------------
```

- MODE 字段的取值如下图：

```
                  RV 64
  ------------------------------------------------------------
  |  Value  |  Name  |  Description                    |
  |-----------------------------------------------------------|
  |    0    | Bare   | No translation or protection       |
  |  1 - 7  | ---    | Reserved for standard use          |
  |    8    | Sv39   | Page-based 39 bit virtual addressing  | <-- 我们使用的mode
  |    9    | Sv48   | Page-based 48 bit virtual addressing  |
  |    10   | Sv57   | Page-based 57 bit virtual addressing  |
  |    11   | Sv64   | Page-based 64 bit virtual addressing  |
  | 12 - 13 | ---    | Reserved for standard use          |
  | 14 - 15 | ---    | Reserved for standard use          |
  ------------------------------------------------------------
```

- ASID (Address Space Identifier) ： 用来区分不同的地址空间，此次实验中直接置0即可。
- PPN (Physical Page Number) ：顶级页表的物理页号，通常 `PPN = physical address >> 12`。

#### 3.2.2 RISC-V Sv39 Virtual Address and Physical Address

```
  38      30 29      21 20      12 11                0
  ------------------------------------------------------------------------
 |  VPN[2]  |  VPN[1]  |  VPN[0]  |       page offset       |
  ------------------------------------------------------------------------
                 Sv39 virtual address
```

```
 55          30 29      21 20      12 11                 0
 ------------------------------------------------------------
|     PPN[2]     |  PPN[1]  |  PPN[0]  |       page offset       |
 ------------------------------------------------------------
                 Sv39 physical address
```

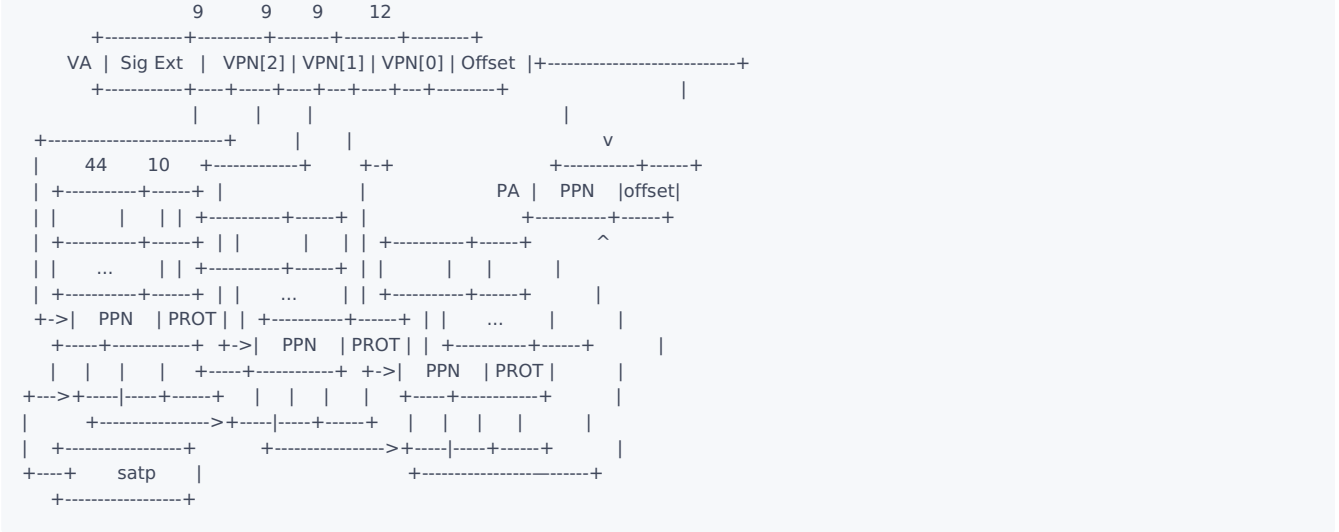
Sv39模式定义物理地址有56位，虚拟地址有64位。但是，虚拟地址的64位只有39位有效，63-39位在本次实验中（高地址映射）需要为1保证地址有效。Sv39支持三级页表结构，VPN[2-0](Virtual Page Number)分别代表每级页表的`虚拟页号`，PPN[2-0](Physical Page Number)分别代表每级页表的`物理页号`。物理地址和虚拟地址的低12位表示页内偏移（page offset）。

### 3.2.3 RISC-V Sv39 Page Table Entry

```
 63    54 53     28 27     19 18     10 9   8 7 6 5 4 3 2 1 0
 ---------------------------------------------------------------
| Reserved |   PPN[2]  |  PPN[1]  |  PPN[0]  | RSW |D|A|G|U|X|W|R|V|
 ---------------------------------------------------------------
                                  |  |||||||| 
                                  |  |||||||`---- V - Valid
                                  |  ||||||`----- R - Readable
                                  |  |||||`------ W - Writable
                                  |  ||||`------- X - Executable
                                  |  |||`-------- U - User
                                  |  ||`--------- G - Global
                                  |  |`---------- A - Accessed
                                  |  `----------- D - Dirty (0 in page directory)
                                  `-------------- Reserved for supervisor software
```

- 0 ~ 9 bit: protection bits
  - V: 有效位，当 V = 0, 访问该PTE会产生Pagefault。
  - R: R = 1 该页可读。
  - W: W = 1 该页可写。
  - X: X = 1 该页可执行。
  - U,G,A,D,RSW本次实验中设置为0即可。

### 3.2.4 RISC-V Address Translation Details




- 1.从satp的`PPN`中获取根页表的物理地址。
- 2.通过pagetable中的VPN段,获取PTE。(可以把pagetable看成一个数组，VPN看成下标。PAGE_SIZE为4KB，PTE为64bit(8B), 所以一页中有4KB/8B=512个PTE，而每级VPN刚好有9位，与512个PTE一一对应)。
- 3.检查PTE的`V bit`，如果不合法，应该产生page fault异常。
- 4.检查PTE的`RWX` bits,如果全部为0，则从PTE中的PPN[2-0]得到的是下一级页表的物理地址，则回到第二步。否则当前为最后一级页

表，PPN[2-0]得到的是最终物理页的地址。
- 5.将得到最终的物理页地址，与偏移地址相加，得到最终的物理地址。 (以上为简要的地址翻译过程，完整过程参考这里)

# 4. 实验步骤

## 4.1 环境搭建

### 4.1.1 建立映射

同lab3的文件夹映射方法，目录名为lab4。

### 4.1.2 组织文件结构

```
lab4
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   ├── sched.h
│       │   └── vm.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   ├── vm.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   ├── rand.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
└── Makefile
```

其中 `vmlinux.lds` 会提供给同学们。

## 4.2 创建映射

- 本次实验使用Sv39分配方案，支持3级页表映射
- 在 `vm.c` 中编写函数 `create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)`，用作页表映射的统一接口，其中参数作用如下：
  - `pgtbl` 为根页表的基地址
  - `va`， `pa` 分别为需要映射的虚拟、物理地址的基地址
  - `sz` 为映射的大小
  - `perm` 为映射的读写权限
  - 函数中，若需要为页表空间分配物理页，可以自由管理分配 `end` 之后的物理内存
- 在 `vm.c` 中编写 `paging_init` 函数，调用 `create_mapping` 函数将内核起始（0x80000000）的16MB空间映射到高地址（以 `0xffffffe000000000` 为起始地址），同时也进行等值映射。将必要的硬件地址（如UART）进行等值映射，无偏移
- 映射图如下：

```
Physical Address
------------------------------------------------------------
|     |UART|            |   Kernel 16MB   |
```


## 4.3 修改head.S

### 4.3.1 修改系统启动部分代码

- 在 `_start` 开头先设置 `satp` 寄存器为0，暂时关闭MMU
- 进入S模式后，在适当位置调用 `paging_init` 函数进行映射
- 设置 `satp` 的值以打开MMU
  - 注意： `satp` 中的PPN字段以4KB为单位
  - 注意：修改 `satp` 后需要执行 `sfence.vma` 指令同步虚拟内存相关映射
- 设置 `stvec` 为异常处理函数 `trap_s` 在**虚拟地址空间下**的地址
- 设置 `sp` 的值为虚拟地址空间下的 `init_stack_top`
- 跳转到虚拟地址下的 `start_kernel` ，并在虚拟地址空间中执行后续语句与进程调度
  - 可以先将 `start_kernel` 的虚拟地址装载在寄存器中，并使用 `jr` 指令进行跳转

### 4.3.2 修改M模式下异常处理代码

- 由于M模式下依然使用物理地址，使用虚拟地址将导致内存访问错误。因此，需要保留一片物理地址区域用于异常处理前保存所有寄存器的值
- `mscratch` 寄存器是M mode下专用的临时寄存器。通常，它就用于保存M mode下上下文物理空间的地址。lds文件中分配出了1个page的空间用于储存进程上下文，其顶部标记为 `stack_top` ，请在head.S进入S mode之前的适当位置，将 `mscratch` 寄存器设置为 `stack_top` 的物理地址。
- 在M mode异常处理函数 `trap_m` 的开头，将 `mscratch` 与 `sp` 寄存器的值交换（hint: 使用 `csrrw` 指令），使用上下文空间作为 `trap_m` 的栈并保存 `x1-x31` 寄存器
- 在 `trap_m` 返回前将 `mscratch` 与 `sp` 寄存器的值重新交换回来

## 4.4 修改进程调度相关代码sched.c

### 4.4.1 修改task_init()调整为虚拟地址

- 由于开启了MMU，因此我们需要修改进程相关代码，确保将 `task_struct` 以及各进程的地址划分到虚拟地址空间。

### 4.4.2 在进程调度时打印task_struct地址

- 修改 `schedule()` 函数在调度时的打印输出，要求打印出 `current` 和 `task[next]` 的地址以及进程栈顶 `sp` 的值

## 4.5 完成对不同section的保护

- 通过修改调用 `create_mapping` 时的 `perm` 参数，修改对不同section所在页属性的设置，完成对不同section的保护
  - 包括：text r-x, rodata r--, other rw-
  - **思考题：如何验证这些属性是否成功被保护**
- 在 `head.S` 中，通过修改 `medeleg` 寄存器，将instruction/load/store page fault托管到S模式下
- 修改 `strap.c` 中的handler，添加对page fault的打印

## 4.6 编译及测试

```
ZJU OS LAB 4           GROUP-XX
[PID = 1] Process Create Successfully! counter = 7 priority = 5
[PID = 2] Process Create Successfully! counter = 6 priority = 5
[PID = 3] Process Create Successfully! counter = 5 priority = 5
[PID = 4] Process Create Successfully! counter = 4 priority = 5
[!] Switch from task 0 [task struct: 0xffffffe000ff2000, sp: 0xffffffe000ff3000] to task 4 [task struct: 0xffffffe000fee000, sp: 0xffffffe000fef000], pri
tasks' priority changed
[PID = 1] counter = 7 priority = 1
```

```
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 4 priority = 4
[!] Switch from task 4 [task struct: 0xffffffe000fee000, sp: 0xffffffe000fef000] to task 1 [task struct: 0xffffffe000ff1000, sp: 0xffffffe000ff2000], pri
tasks' priority changed
[PID = 1] counter = 7 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 3 priority = 2
...
```

## 5. 实验任务与要求

请仔细阅读背景知识，理解如何建立页表映射，并按照实验步骤完成实验，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 实验报告：
  - 各实验步骤截图以及结果分析
  - 回答思考题
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（确保make clean）
- 最终目录
  - 将Lab4_319010XXXX目录压缩并打包（若分组，则一组只需要一位同学提交）

```
lab4_319010XXXX
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   ├── sched.h
│       │   └── vm.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   ├── vm.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── put.h
│   ├── rand.h
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── put.c
│   └── rand.c
├── Makefile
└── report
    └── 319010XXXX.pdf
```

**本文贡献者**

> 王星宇（背景知识）
> 朱璟森，沈韬立（实验步骤）

# lab5

# Lab 5: RISC-V 64 用户模式

## 1 实验简介

进一步巩固课堂学习的页式内存管理以及虚拟内存的相关知识，在低地址空间映射用户态程序，并尝试编写简单的系统调用处理函数。

## 2 实验环境

- Docker Image

## 3 背景知识

### 3.1 User 模式基础介绍

处理器具有两种不同的模式：用户模式和内核模式。在内核模式下，执行代码对底层硬件具有完整且不受限制的访问权限，它可以执行任何CPU指令并引用任何内存地址。在用户模式下，执行代码无法直接访问硬件，必须委托给系统提供的接口才能访问硬件或内存。处理器根据处理器上运行的代码类型在两种模式之间切换。应用程序以用户模式运行，而核心操作系统组件以内核模式运行。

当启动用户模式应用程序时，内核将为该应用程序创建一个进程，为应用程序提供了专用虚拟地址空间等资源。因为应用程序的虚拟地址空间是私有的，所以一个应用程序无法更改属于另一个应用程序的数据。每个应用程序都是独立运行的，如果一个应用程序崩溃，其他应用程序和操作系统不会受到影响。同时，用户模式应用程序可访问的虚拟地址空间也受到限制，在用户模式下无法访问操作系统的虚拟地址，可防止应用程序修改关键操作系统数据。

### 3.2 系统调用约定

系统调用是用户态应用程序请求内核服务的一种方式。在RISC-V中，我们使用 `ecall` 指令进行系统调用。当执行这条指令时处理器会提升特权模式，跳转到异常处理函数处理这条系统调用。

Linux中RISC-V相关的系统调用可以在 `.../include/asm-generic/unistd.h` 中找到，syscall(2)手册页上对RISC-V架构上的调用说明进行了总结，系统调用参数使用a0 - a5，系统调用号使用a7，系统调用的返回值会被保存到a0, a1中。

## 4 实验步骤

### 4.1 环境搭建

#### 4.1.1 建立映射

同lab4的文件夹映射方法，目录名为lab5。

#### 4.1.2 组织文件结构

```
lab5
├── arch
│   └── riscv
│       ├── include
│       │   ├── put.h
│       │   ├── sched.h
│       │   ├── syscall.h
│       │   └── vm.h
│       ├── kernel
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sched.c
│       │   ├── strap.c
│       │   ├── vm.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   └── ...
├── init
```

```
|   └── ...
├── lib
|   └── ...
└── Makefile
```

## 4.2 添加系统调用处理函数

本实验我们将增加对系统调用的处理。在RISC-V中系统调用通过`ecall`（environment call）来实现。在U-mode、S-mode、M-mode下执行`ecall`分别会触发environment-call-from-U-mode异常、environment-call-from-S-mode异常、environment-call-from-M-mode异常。在系统调用的实现中，我们通过在U-mode下执行`ecall`触发environment-call-from-U-mode异常，并由S-mode中运行的内核处理这个异常。

在head.S中内核boot阶段时，设置 medeleg 寄存器为用户模式系统调用添加异常委托。 在没有设置异常委托的情况下，`ecall`指令产生的异常由M-mode来处理，而不是交由内核所在的S-mode进行处理。通过 medeleg 中设置相应的位，可以将environment-call-from-U-mode异常直接交由S-mode处理。具体设置方式参照RISC-V手册。

和实验4一样，为了便于确认实验完成的正确与否，本实验对异常处理函数的命名和类型有要求，要求其函数原型如下：

```
handler_s(size_t scause, size_t sepc, uintptr_t *regs);
```

此处的regs概念对应linux中的pt_regs，将上下文内容所在的地址传入处理函数。

为S模式异常处理函数添加系统调用的处理逻辑，从a7中获得当前系统调用号，从a0 - a5中获得系统调用相关的参数，本次实验中要求实现的系统调用分别为：172号系统调用`SYS_GETPID`，64号系统调用`SYS_WRITE`。 RISC-V中 scause 寄存器储存了中断或异常的类型（原因），通过判断scause的值来进行相应的处理，针对environment-call-from-U-mode异常，从a7中获得系统调用号实现相应的系统调用。系统调用的返回值通过a0，a1寄存器返回。

本次实验要求的系统调用函数原型以及具体功能如下：

- 64 号系统调用 `sys_write(unsigned int fd, const char* buf, size_t count)`
  该调用将用户态传递的字符串打印到屏幕上，此处`fd`为标准输出（1），`buf`为用户需要打印的起始地址，`count`为字符串长度，返回打印的字符数。
- 172 号系统调用 `sys_getpid()`
  该调用从`current`中获取当前的`pid`放入`a0`中返回，无参数。

## 4.3 修改进程初始化以及进程调度相关逻辑

#### task struct的修改

本实验中需要为task struct增添新的变量，根据实现方式的不同，新增的方式会有所不同，同学们根据自己之前的实现**按需设计**。举例如下：

```
size_t sepc;      // 保存的sepc
size_t sscratch;  // 保存的sscratch
mm_struct *mm;    // 虚拟内存映射相关
```

在进程初始化时，除了之前实验所需的内容外，还需要为新进程创建页表。新进程页表首先需要把内核相关的映射建立好，这里可以选择简单地把内核页表复制一份（需要注意如何同步不同进程间内核页表的变化）。还需要建立用户程序以及用户栈的分配和映射。

本实验还需要储存虚拟内存映射相关的`mm_struct`，`mm_struct`需要有进程的页表地址、虚拟地址空间的分配情况等信息。

初始化进程后，需要设置好sepc、sstatus等CSR的值，并且需要修改ra使该进程启动后能进行`sret`，跳转到用户态程序入口。

#### 内存空间分配

由于增加了用户态支持，虚拟地址的映射方式与lab4有所不同，变化如下。虚拟地址0x0-0x80000000供用户态使用，0xffffffdf80000000-0xffffffe000000000作为设备地址（UART等）、以及内核用于进行虚拟地址映射的地址，0xffffffe000000000开始的地址直接与物理内存进行映射。

```
Physical Address
-------------------------------------------------------------
|    |UART|           |   Kernel 16MB   |
-------------------------------------------------------------
     ^              ^
   0x10000000        0x80000000
     |              └──────────────────────────────────────────┐
     └──────────────────────────────────────────┐              |
Virtual Address                                 ↓              ↓
```

```
-----------------------------------------------------------------------------------------------------
| User 2GB |                <-- User Stack |     |UART|              |   Kernel 16MB   |
-----------------------------------------------------------------------------------------------------
^        ^                              ^     ^                   ^
0x0   0x80000000            0xffffffdf80000000     0xffffffdf90000000      0xfffffffe000000000
```

**用户栈与内核栈**

用户态程序与内核并不共用栈，因此内核的实现需要区分用户态栈和内核态栈，这就带来一个问题，即在异常处理的过程中需要对栈进行切换。

从用户态进入内核态时，即发生中断或异常时，需要切换用户栈为内核栈，一种实现方式是，访问进程控制结构中的内核栈数据，我们可以使用sscratch来保存当前进程task struct的地址。为实现这一功能，当异常发生时需要首先取出sscratch的内容，方便在后续的逻辑中直接访问task struct中的内容，同样的，在退出异常时，需要再将task struct的地址存回到sscratch中。

另外一种思路是，在用户态下，使用sscratch来直接保存指向内核栈的指针，使用逻辑与上面类似，不再赘述。同时，在进程的调度过程中，同学们也需要根据自己的实现为进程的sscratch、sepc、scause等必要的特权寄存器内容进行切换。

## 4.4 用户态测试程序

由于内核和用户态程序的功能不同，以及为了兼容将来文件系统等功能，本次实验中所使用的用户态程序需要额外进行编译。同学们在实验的过程中可以选择自行编译用户态程序或者使用仓库中提供的二进制镜像。下面对加载二进制镜像的操作进行讲解，我们使用qemu的initrd选项来加载用户态程序。这里实验仓库中提供的用户态程序是一段经过处理的二进制镜像，同学们在使用下面的命令加载时，实际上是把该段镜像以initfs的形式加载到内存0x84000000处，那么只需要在运行时将此页代码映射至用户空间即可：

```
$ qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -initrd hello.bin
```

`hello.bin` 未经裁切前的内容如下所示，可以看到测试程序在main中执行了一次172号调用来获得进程pid，程序在printf中还调用了64号系统调用来打印字符串。

```
0000000000000004 <main>:
   4:   fd010113            addi    sp,sp,-48
   8:   02113423            sd      ra,40(sp)
   c:   02813023            sd      s0,32(sp)
  10:   00913c23            sd      s1,24(sp)
  14:   01213823            sd      s2,16(sp)
  18:   01313423            sd      s3,8(sp)
  1c:   00010993            mv      s3,sp
  20:   00000917            auipc   s2,0x0
  24:   3b890913            addi    s2,s2,952 # 3d8 <printf+0x358>
  28:   fff00493            li      s1,-1
  2c:   0ac00893            li      a7,172
  30:   00000073            ecall
  34:   00050413            mv      s0,a0
  38:   00098613            mv      a2,s3
  3c:   00040593            mv      a1,s0
  40:   00090513            mv      a0,s2
  44:   03c000ef            jal     ra,80 <printf>
  48:   00048793            mv      a5,s1
  4c:   fff7879b            addiw   a5,a5,-1
  50:   fe079ee3            bnez    a5,4c <main+0x48>
  54:   fd9ff06f            j       2c <main+0x28>

000000000000004c <printf>:
  ...
 380:   04000893            li      a7,64
 384:   00078513            li      a0,1
 388:   00070593            mv      a1,a4 # buf
 38c:   00068613            mv      a2,a3 # count
 390:   00000073            ecall
```

## 4.5 编译及测试

```
ZJU OS LAB 5            STUDEDNT-ID/GROUP-ID
[PID = 1] Process Create Successfully! counter = 5
[PID = 2] Process Create Successfully! counter = 5
```

```
[PID = 3] Process Create Successfully! counter = 5
[PID = 4] Process Create Successfully! counter = 5
[!] Switch from task 0[0xffffffe000006030] to task 4[0xffffffe000006310], prio: 5, counter: 5
[User] pid: 4
[!] Switch from task 4[0xffffffe000006310] to task 3[0xffffffe000006258], prio: 5, counter: 5
[User] pid: 3
[!] Switch from task 3[0xffffffe000006258] to task 2[0xffffffe0000061a0], prio: 5, counter: 5
[User] pid: 2
[!] Switch from task 2[0xffffffe0000061a0] to task 1[0xffffffe0000060e8], prio: 5, counter: 5
[User] pid: 1
```

## 5 实验任务与要求

请仔细阅读背景知识，理解如何建立页表映射，并按照实验步骤完成实验，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 实验报告：
  - 各实验步骤截图以及结果分析
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（执行过make clean的工程，不需要提交中间及结果文件）
  - 将lab5_319010XXXX目录压缩成zip格式（若分组，则一组只需要一位同学提交）
- 将报告和工程压缩包提交至学在浙大

```
lab5_319010XXXX
├── arch
│   └── ...
├── include
│   └── ...
├── init
│   └── ...
├── lib
│   └── ...
└── Makefile
```

**本文贡献者**

> 谢洵、徐金焱

# lab6

# Lab 6：RISC-V 动态内存分配与缺页异常处理

## 1. 实验简介

在充分理解前序实验中RISC-V地址空间映射机制与任务调度管理的前提下，进一步了解与**动态**内存管理相关的重要结构，实现基本的内存管理算法，并最终能够较为**综合**地实现对进程的内存管理与调度管理。

## 2. 实验目标

- 目标一：了解 **Buddy System** 和 **Slub Allocator** 物理内存管理机制的实现原理，并用 **Buddy System** 配合 **Slub Allocator** 管理和分配物理内存，最终实现统一的内存分配/释放接口:**kmalloc**/**kfree**。
- 目标二：在**mm_struct**中补充实现**vm_area_struct**数据结构的支持，并结合后续**mmap**等系统调用实现对进程**多区域**虚拟内存的管理。
- 目标三：实现**222: mmap, 215: munmap, 226: mprotect** 系统调用。
- 目标四：在 **Lab5** 实现用户态程序的基础上，添加缺页异常处理**Page Fault Handler**，并在分配物理页或物理内存区域的同时更新页表。
- 目标五：综合上述实现，为进程加入 **fork** 机制，能够支持创建新的用户态进程，并测试运行。

## 3. 实验环境

- Docker Image

## 4. 背景知识

### 4.1 物理内存动态分配算法

#### 4.1.1 伙伴系统（Buddy System）

操作系统内核可采用伙伴系统（Buddy System）对物理内存进行管理。伙伴系统有多种，最常见的为依次按2的幂次大小组成的二进制伙伴系统（Binary Buddy System），这也是Linux系统中目前所采用的一种物理内存管理算法

1. Binary buddy system <--本次试验所采用的伙伴系统算法
2. Fibonacci buddy system
3. Weighted buddy system
4. Tertiary buddy system

#### 4.1.2 SLAB/SLUB分配器

Buddy System以2^n个page为粒度来进行物理内存的分配管理，Linux中实现的Buddy System最小阶（order）为2^0，即最小4KB，最大阶为2^10，也就是4MB大小的连续物理内存空间。但是Buddy System并不满足更细粒度的内存管理，当由于分配/释放的内存空间与最小存储空间单位(1-Page)不完全相符时，如在先前实验中实现过的**task struct**(该数据结构的大小**小于**4KB)，该类小空间内存的频繁分配/释放容易产生**内部碎片**。为此需要借助专门的内存分配器加以优化解决**内部碎片**的问题。

SLAB是一个通用名称，指的是使用对象缓存的内存分配策略，从而能够有效地分配和释放内核对象。它首先由Sun工程师Jeff Bonwick编写，并在Solaris 2.4内核中实现。Linux目前为它的“Slab”分配器提供了三种选择: SLAB是最初的版本，基于Bonwick的开创性论文，从Linux内核版本2.2开始就可以使用了。它忠实地实现了Bonwick的建议，并在Bonwick的后续论文中描述了多处理器的改变。

SLUB是下一代内存分配器（自Linux 2.6.23以来一直是Linux内核中的默认配置）。它继续采用基本的“SLAB”模型，但修复了SLAB设计中的一些缺陷，特别是在拥有大量处理器的系统上。且SLUB甚至要比SLAB简单。SLUB机制【link】，可以理解为将Page拆分为更小的单位来进行管理。SLUB系统的核心思想是使用**对象**的概念来管理物理内存。

#### 4.1.3 SLUB & Buddy System

本次试验中，SLUB直接给出，学生需要自己实现Buddy System 以来支持 SLUB 功能，下面介绍下SLUB 与 Buddy System的依赖关系：

- Buddy System 是以**页(Page)**为基本单位来分配空间的，其提供两个接口 **alloc_pages(申请若干页)** 和 **free_pages(释放若干页)**
- SLUB在初始化时，需要预先申请一定的空间来做数据结构和Cache的初始化，此时依赖于Buddy System提供的上述接口。（具体可以参考slub.c中的**slub_init**）
- 实验所用的slub提供8, 16, 32, 64, 128, 256, 512, 1024, 2048（单位：Byte）九种object-level（的内存分配/释放功能。(具体可以参考slub.c中的 **kmem_cache_alloc / kmem_cache_free**)
- slub.c 中的(**kmalloc / kfree**) 提供内存动态分配/释放的功能。根据请求分配的空间大小，来判断是通过kmem_cache_alloc来分配object-level的

空间，还是通过alloc_pages来分配page-level的空间。kfree同理。

## 4.2 vm_area_struct 介绍

在先前的实验中，我们已经在 task_struct 中添加了储存虚拟内存映射相关信息的 mm_struct 数据结构，而并未对虚拟地址空间的分配情况进行设置与保存。在linux系统中， vm_area_struct 是虚拟内存管理的基本单元， vm_area_struct 保存了有关连续虚拟内存区域(简称vma)的信息。linux 具体某一进程的虚拟内存区域映射关系可以通过 procfs【Link】 读取 /proc/pid/maps 的内容来获取:

比如，如下一个常规的 bash 进程，假设它的进程号为 7884 ，则通过输入如下命令，就可以查看该进程具体的虚拟地址内存映射情况(部分信息已省略)。

```
#cat /proc/7884/maps
556f22759000-556f22786000 r--p 00000000 08:05 16515165                   /usr/bin/bash
556f22786000-556f22837000 r-xp 0002d000 08:05 16515165                   /usr/bin/bash
556f22837000-556f2286e000 r--p 000de000 08:05 16515165                   /usr/bin/bash
556f2286e000-556f22872000 r--p 00114000 08:05 16515165                   /usr/bin/bash
556f22872000-556f2287b000 rw-p 00118000 08:05 16515165                   /usr/bin/bash
556f22fa5000-556f2312c000 rw-p 00000000 00:00 0                        [heap]
7fb9edb0f000-7fb9edb12000 r--p 00000000 08:05 16517264                   /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fb9edb12000-7fb9edb19000 r-xp 00003000 08:05 16517264                   /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
...
7ffee5cdc000-7ffee5cfd000 rw-p 00000000 00:00 0                      [stack]
7ffee5dce000-7ffee5dd1000 r--p 00000000 00:00 0                      [vvar]
7ffee5dd1000-7ffee5dd2000 r-xp 00000000 00:00 0                       [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0              [vsyscall]
```

从中我们可以读取如下一些有关该进程内虚拟内存映射的关键信息：

- vm_start ：(第1列) 指的是该段虚拟内存区域的开始地址；
- vm_end ：(第2列) 指的是该段虚拟内存区域的结束地址；
- vm_flags ：(第3列) 该 vm_area 的一组权限(rwx)标志， vm_flags 的具体取值定义可参考linux源代码的这里(linux/mm.h)
- vm_pgoff ：(第4列) 虚拟内存映射区域在文件内的偏移量；
- vm_file ：(第5/6/7列)分别表示：映射文件所属设备号/以及指向关联文件结构的指针(如果有的话，一般为文件系统的inode)/以及文件名；

其它保存在 vm_area_struct 中的信息还有：

- vm_ops ：该 vm_area 中的一组工作函数;
- vm_next/vm_prev : 同一进程的所有虚拟内存区域由**链表结构**链接起来，这是分别指向前后两个 vm_area_struct 结构体的指针;

## 4.3 文件/内存映射 系统调用

沿用Lab5中添加syscall的方法，本次实验中要求实现的系统调用函数原型以及具体功能如下：

### 4.3.1 实现 222 号系统调用：mmap. (memory map)

```
void *mmap (void *__addr, size_t __len, int __prot,
            int __flags, int __fd, __off_t __offset)
```

- 说明：在Linux中，使用 mmap 在进程的虚拟内存地址空间中分配新地址空间（从 void *__addr 开始, 长度为 size_t __len 字节），创建和物理内存的映射关系。它可以分配一段匿名的虚拟内存区域，也可以映射一个文件到内存。利用该特性， mmap() 系统调用使得进程之间通过映射同一个普通文件实现共享内存成为可能。普通文件被映射到进程虚拟地址空间后，进程可以向访问普通内存一样对文件进行访问，而不必再调用 read() ， write() 等操作。
- 参数 int __prot 表示了对所映射虚拟内存区域的权限保护要求。利用如下所示不同的 int __prot 取值可设置不同的内存保护权限：

| Defined __prot Symbol | Value | Description |
|---|---|---|
| PROT_NONE | 0x0 | 页内容不可被访问 |
| PROT_READ | 0x1 | 页内容可以被读取 |
| PROT_WRITE | 0x2 | 页可以被写入内容 |
| PROT_EXEC | 0x4 | 页内容可以被执行 |
| PROT_SEM | 0x8 | (可选) 页面可能用于原子操作(atomic operation) |

| PROT_GROWSDOWN | 0x01000000 | (可选) / |
| --- | --- | --- |
| PROT_GROWSUP | 0x02000000 | (可选) / |

- 参数 `int __flags` (可选) 指定了映射对象的类型，具体的参数设置可以参考这里【link】
- 参数 `int __fd` (可选) 为即将映射到进程空间的文件描述字，一般由 `open()` 返回，同时，`__fd` 可以指定为 `-1`，此时须指定 `__flags` 参数中的 MAP_ANON，表明进行的是匿名映射。`mmap` 的作用是映射文件描述符 `__fd` 指定文件的 `[off,off + len]` 区域至调用进程的 `[addr, addr + len]` 的虚拟内存区域。在Lab7 ELF文件的载入运行实验中，可以实现进程之间通过映射同一个普通文件实现共享内存。
- 参数 `__off_t __offset` (可选) 该参数一般设为 `0`，表示从文件头开始映射。
- 调用 mmap `返回：被映射内存区域的指针`

#### 4.3.2 实现 226 号系统调用：mprotect. (memory protect)

```
int mprotect (void *__addr, size_t __len, int __prot)
```

- 说明：在Linux中，使用 `mprotect` 函数可以用来修改一段指定内存区域（从 `void *__addr` 开始, 长度为 `size_t __len` 字节）的内存保护权限属性：`int __prot`。
- 调用 mprotect `返回：0` 表示 mprotect 成功; `返回：-1` 表示 mprotect 出错;

#### 4.3.3 实现 215 号系统调用：munmap. (memory unmap)

```
int munmap (void *__addr, size_t __len)
```

- 说明：在Linux中，使用 `munmap` 解除从 `void *__addr` 开始, 长度为 `size_t __len` 字节区域的任何内存映射。
- 调用 munmap `返回：0` 表示munmap成功; `返回：-1` 表示munmap出错;

### 4.4 缺页异常 Page Fault

缺页异常是一种正在运行的程序访问当前未由内存管理单元（MMU）映射到虚拟内存的页面时，由计算机硬件引发的异常类型。访问未被映射的页或访问权限不足，都会导致该类异常的发生。处理缺页异常通常是操作系统内核的一部分。当处理缺页异常时，操作系统将尝试使所需页面在物理内存中的位置变得可访问（建立新的映射关系到虚拟内存）。而如果在非法访问内存的情况下，即发现触发Page Fault的虚拟内存地址(Bad Address)不在当前进程 `vm_area_struct` 链表中所定义的允许访问的虚拟内存地址范围内，或访问位置的权限条件不满足时，缺页异常处理将终止该程序的继续运行。

#### 4.4.1 RISC-V Page Faults

RISC-V 异常处理：当系统运行发生异常时，可即时地通过解析csr scause寄存器的值，识别如下三种不同的Page Fault。

SCAUSE 寄存器指示发生异常的种类：

| Interrupt/Exception<br>scause[XLEN-1] | Exception Code<br>scause[XLEN-2:0] | Description |
| --- | --- | --- |
| 0 | 12 | Instruction Page Fault |
| 0 | 13 | Load Page Fault |
| 0 | 15 | Store/AMO Page Fault |

#### 4.4.2 常规处理Page Fault的方式介绍

处理缺页异常时所需的信息如下：

- 
  - 触发**Page Fault**时访问的虚拟内存地址VA (Virtual Address)，也称为Bad Address
    - 当触发page fault时，csr STVAL寄存器被被硬件自动设置为该出错的VA地址。
- 
  - 导致**Page Fault**的类型
    - Exception Code = 12: page fault caused by an instruction fetch
    - Exception Code = 13: page fault caused by a read
    - Exception Code = 15: page fault caused by a write

- 
  - 发生Page Fault时的指令执行位置以及当前程序所运行的权限模式(User/Kernel) (可选)
- 
  - 保存在当前进程task_struct->mm_struct中合法的VMA映射关系
    - 保存在 `vm_area_struct` 链表中

### 4.5 `fork` 系统调用

在之前的lab3中，我们创建了四个进程。现在我们需要在lab5用户态支持和lab6之前实现的基础上实现 `fork` 系统调用。

#### 4.5.1 实现 220 号系统调用：fork (clone) .

```
pid_t fork（void)
```

- `fork()` 通过复制当前进程创建一个新的进程，新进程称为子进程，而原进程称为父进程。
- 子进程和父进程在不同的内存空间上运行，在其中一个进程中调用 `mmap` 或 `munmap` 系统调用不会影响另一进程的内存空间。
- 父进程 `fork` 成功时 `返回：子进程的pid` ，子进程 `返回：0` 。 `fork` 失败则父进程 `返回：-1` 。
- 本系统调用创建的子进程需要拷贝父进程 `task_struct` 、 `根页表` 、 `mm_struct` 以及父进程的 `用户栈` 等信息。
- 本系统调用实现时拷贝了父进程的全部内存空间，这是非常耗时的行为。Linux中使用了写时复制（copy-on-write）机制，fork创建的子进程首先与父进程共享物理内存空间，直到父子进程有修改内存的操作发生时再为子进程分配物理内存。

思考题： `fork` 系统调用为什么可以有两个返回值？

## 5. 实验步骤

### 5.1 环境搭建

#### 5.1.1 建立映射

同lab5的文件夹映射方法，目录名为lab6。

#### 5.1.2 组织文件结构

请同学们参考如下结构为lab6组织文件目录：其中 `slub.h` 和 `slub.c` 由实验预先给出

```
lab6
├── arch
│   └── riscv
│       ├── include
│       │   └── ...
│       ├── kernel
│       │   │
│       │  ...
│       │   ├── slub.c
│       │   ├── buddy.c
│       │   └── fault.c
│       └── Makefile
├── include
│   └── ...
├── init
│   └── ...
├── lib
│   └── ...
└── Makefile
```

### 5.2 管理空闲内存空间 ，在 buddy.c 中实现 buddy system

#### 5.2.1 相关数据结构以及接口

```
struct buddy {
```

```
  unsigned long size;
  unsigned *bitmap;
};

void init_buddy_system(void);
void *alloc_pages(int);
void free_pages(void*);
```

- size：buddy system 管理的**页**数
- bitmap：用来记录未分配的内存页信息
- void init_buddy_system(void)：初始化buddy system
- void *alloc_pages(int npages)：申请n个page空间，返回起始虚拟地址
- void free_pages(void* addr)：释放以addr为基地址的空间块

**init_buddy_system：理解下图，并按照后续指导完成buddy system的初始化**

```
假设我们系统一共有8个可分配的页面，可分配的页面数需保证是2^n

memory: （这里的 page 0～7 计做 page X）
+--------+--------+--------+--------+--------+--------+--------+--------+
| page 0 | page 1 | page 2 | page 3 | page 4 | page 5 | page 6 | page 7 |
+--------+--------+--------+--------+--------+--------+--------+--------+

buddy system将页面整理为 2^n 2^(n-1) ... 2^1 2^0 不同大小的块：

8 +---> 4 +---> 2 +---> 1
 |     |     |
 |     |     +---> 1
 |     |
 |     +---> 2 +---> 1
 |           |
 |           +---> 1
 |
 +---> 4 +---> 2 +---> 1
       |     |
       |     +---> 1
       |
       +---> 2 +---> 1
             |
             +---> 1

当申请空间时候，buddy system 会将 满足要求的最小块 分配出去。上图是满二叉树结构，所以我们可以很方便的使用数组来保存相关信息。如下：

buddy.size = 8

buddy.bitmap:
+-----------------------------------------------------------+
| 8 | 4 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-----------------------------------------------------------+

bitmap中的 数字 代表的 可分配的物理地址连续页的最大个数 ， 比如此时：

bitmap[0] 对应的是 page 0 ~ page 7
bitmap[1] 对应的是 page 0 ~ page 3
bitmap[2] 对应的是 page 4 ~ page 7
bitmap[3] 对应的是 page 0 ~ page 1
bitmap[4] 对应的是 page 2 ~ page 3
... 以此类推
```

- 实现的相关细节
  - 计算出总共物理页面。（提示： 查看 PA 的起始 结束地址）。
  - 完成对 buddy 数据结构的初始化。（提示：用数组表示满二叉树，parent 节点的index 与 child节点的index 存在一一对应的关系）
  - 由于kernel的代码占据了一部分空间，在初始化结束后，需要将这一部分的空间标记为已分配。(提示：可以直接使用alloc_pages 将这段空

间标记为已分配)。

- 这里需要注意bitmap 与 page X 的对应关系。（提示 ：查看((index + 1) * buddy.bitmap[index] - buddy.size)) 与 page X 的关系
- 将 init_buddy_system 添加至**建立内核映射**之前， 修改之前实现的物理页分配逻辑

**alloc_pages：理解下图，并完成alloc_pages接口**

```
memory: （这里的 page 0～7 计做 page X）

+--------+--------+--------+--------+--------+--------+--------+--------+
| page 0 | page 1 | page 2 | page 3 | page 4 | page 5 | page 6 | page 7 |
+--------+--------+--------+--------+--------+--------+--------+--------+

buddy.bitmap:
+-----------------------------------------------------------+
| 8 | 4 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-----------------------------------------------------------+

alloc_pages(3) 请求分配3个页面 由于buddy system每次必须分配2^n个页面，所以我们将 3 个页面向上扩展至 4 个页面。

我们从树根出发（buddy.bitmap[0]），查找到恰好满足当前大小需求的节点：
     ↓
8 +---> 4 +---> 2 +---> 1
 |     |     |
 |     |     +---> 1
 |     |
 |     +---> 2 +---> 1
 |           |
 |           +---> 1
 |
 +---> 4 +---> 2 +---> 1
       |     |
       |     +---> 1
       |
       +---> 2 +---> 1
             |
             +---> 1


  index = 1
    ↓
+-----------------------------------------------------------+
| 8 | 4 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-----------------------------------------------------------+

通过 index 找到对应 page X，再通过适当计算最终转化为VA

更新bitmap：

↓     ↓
4 +---> 0 +---> 2 +---> 1
 |     |     |
 |     |     +---> 1
 |     |
 |     +---> 2 +---> 1
 |           |
 |           +---> 1
 |
 +---> 4 +---> 2 +---> 1
       |     |
       |     +---> 1
       |
       +---> 2 +---> 1
             |
             +---> 1

  ↓  ↓
+-----------------------------------------------------------+
| 4 | 0 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
```

```
+-------------------------------------------------------------+
```

由于 index = 1 对应的 4 个 page已经分配出去，所以 bitmap[1] = 0

无需再去更新 index = 1的所有子节点了，因为我们的查找是从父节点开始的， 当一个节点的可分配 页面数 为0，就不会去查找其子节点了。

同理，由于 bitmap[0] 所对应的 8 个物理地址连续页， 其中的 4 个被分配出去了， 所以bitmap[0]对应的可分配物理地址连续的页数 为 4 个。

**free_pages：理解下图，并完成free_pages接口**

我们考虑 alloc_pages(1) 之后的状态：

```
4 +---> 2 +---> 2 +---> 1
 |     |     |
 |     |     +---> 1
 |     |
 |     +---> 1 +---> 0 ←
 |           |
 |           +---> 1
 |
 +---> 4 +---> 2 +---> 1
       |     |
       |     +---> 1
       |
       +---> 2 +---> 1
             |
             +---> 1

buddy.bitmap:
                      ↓
+-------------------------------------------------------------+
| 4 | 2 | 4 | 1 | 2 | 2 | 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------------------------------------------------------------+
```

现在对这个页面进行free_pages的操作：

通过 va 进行适当的转换，我们可以得到该页面对应的 bitmap index，将其更新为正确的值，并更新其祖先的值。

更新其祖先的值的时候，需要去判断是否需要对祖先的左右两个孩子进行合并

若两个孩子都未被分配 则合并，否则不合并，修改其值为 左右两个孩子中最大物理连续页的数量。具体过程如下图：

```
                    index = 7
                      ↓
+-------------------------------------------------------------+
| 4 | 2 | 4 | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  修改 bitmap[7] = 1
+-------------------------------------------------------------+


          index = 3
            ↓
+-------------------------------------------------------------+
| 4 | 2 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  修改 bitmap[4], 由于其左右两个孩子都
+-------------------------------------------------------------+  未被分配，故需要合并 bitmap[4] = 2


    index = 1
      ↓
+-------------------------------------------------------------+
| 4 | 4 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  修改 bitmap[1], 同理 bitmap[1] = 4
+-------------------------------------------------------------+


index = 0
  ↓
+-------------------------------------------------------------+
| 4 | 4 | 4 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  修改 bitmap[0], 同理 bitmap[0] = 8
+-------------------------------------------------------------+
```

## 5.3 在 slub.c 中实现 slub 内存动态分配算法

- 同学们需要在实现Buddy System接口的基础上，进一步实现SLUB分配器
- SLUB分配器的代码**已给出（见代码仓库lab6目录）**，同学们只需将SLUB分配器移植到自己的实验项目中即可

SLUB分配器提供的接口说明如下：

#### 5.3.1 slub接口：kmem_cache_create

```
struct kmem_cache *
kmem_cache_create(const char *name, unsigned int size, unsigned int align,
  slab_flags_t flags, void (*ctor)(void *))
```

#### 5.3.2 slub接口：kmem_cache_destroy

参考 linux-5.9.12/mm/slab_common.c

```
void kmem_cache_destroy(struct kmem_cache *s)
```

#### 5.3.3 slub接口：kmem_cache_alloc

参考 linux-5.9.12/mm/slub.c

Target：Allocate an object from this cache. The flags are only relevant if the cache has no available objects.

Return: pointer to the new object or NULL in case of error

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
```

#### 5.3.4 slub接口：kmem_cache_free

参考 linux-5.9.12/mm/slub.c

Target：Free an object which was previously allocated from this cache.

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp)
```

## 5.4 实现统一内存分配接口

#### 5.4.1 在实现内存分配接口之前，需要介绍下 slub 中页面的属性，相关数据结构如下：

```
// slub.c
enum{
 PAGE_FREE,
 PAGE_BUDDY,
 PAGE_SLUB,
 PAGE_RESEARVE
};

// slub.h
struct page {
 unsigned long flags;
 int count;
 struct page *header;
 struct page *next;
 struct list_head slub_list;
 struct kmem_cache *slub;
 void *freelist;
};
```

其中 page 中 flags 用来表示当前page的类别，因此 如果一个页面是用来做object-level内存分配那么它的flag 应设置为 **PAGE_SLUB**， 如果是通过

Buddy System(alloc_pages)分配的，那么其flags应设置为 **PAGE_BUDDY**。

同学需要弄清楚 **set_page_attr / clear_page_attr / cache_alloc_pages / ADDR_TO_PAGE / PAGE_TO_ADDR** 相关代码实现逻辑

#### 5.4.2 实现内存分配接口 void *kmalloc（size_t size)

该函数的功能是申请大小为 size 字节的连续物理内存，成功返回起始地址，失败返回 NULL。

- 每个 slub allocator 分配的对象大小见 **kmem_cache_objsize**。
- 当请求的物理内存**小于等于**2^11字节时，从 `slub allocator` 中分配内存。
- 当请求的物理内存大小**大于**2^11字节时，从 `buddy system` 中分配内存。

```
void *kmalloc(size_t size)
{
 int objindex;
 void *p;

 if(size == 0)
          return NULL;

      // size 若在 kmem_cache_objsize 所提供的范围之内，则使用 slub allocator 来分配内存
      for(objindex = 0; objindex < NR_PARTIAL; objindex ++){
    // YOUR CODE HERE
 }

      // size 若不在 kmem_cache_objsize 范围之内，则使用 buddy system 来分配内存
 if(objindex >= NR_PARTIAL){
          // YOUR CODE HERE

          set_page_attr(p, (size-1) / PAGE_SIZE, PAGE_BUDDY);
 }

      return p;
}
```

#### 5.4.2 实现内存分配接口 void kfree(void *addr)

```
void kfree(const void *addr)
{
   struct page *page;

   if(addr == NULL)
      return;

   // 获得地址所在页的属性
   // YOUR CODE HERE

   // 判断当前页面属性
   if(page->flags == PAGE_BUDDY){
      // YOUR CODE HERE
      clear_page_attr(ADDR_TO_PAGE(addr)->header);

   } else if(page->flags == PAGE_SLUB){
      // YOUR CODE HERE
   }

   return;
}
```

#### 5.4.3 修改之前代码中的页面分配逻辑，使用 alloc_pages / free_pages / kmalloc / kfree 来进行页面分配和释放

### 5.5 为 mm_struct 添加 vm_area_struct 数据结构

### 5.5.1 定义 `vm_area_struct` 数据结构：

```
typedef struct { unsigned long pgprot; } pgprot_t;
struct vm_area_struct {
   /* Our start address within vm_area. */
   unsigned long vm_start;
   /* The first byte after our end address within vm_area. */
   unsigned long vm_end;
   /* linked list of VM areas per task, sorted by address. */
   struct vm_area_struct *vm_next, *vm_prev;
   /* The address space we belong to. */
   struct mm_struct *vm_mm;
   /* Access permissions of this VMA. */
   pgprot_t vm_page_prot;
   /* Flags*/
   unsigned long vm_flags;
};
```

### 5.5.2 `vm_page_prot` 和 `vm_flags` 的区别

我们可以注意到这两个字段的C语言类型。`vm_page_prot` 的类型是 `pgprot_t`，这是arch级别的数据类型，这意味着它可以直接应用于底层架构的Page Table Entries。在RISC-V 64位上，这个字段直接存储了vma的pte中保护位的内容。而 `vm_flags` 是一个与arch无关的字段，它的位是参照linux/mm.h中定义的。可简单地将 `vm_flags` 看作是 `vm_page_prot` 的翻译结果，方便在操作系统其它地方的代码中判断每个 `vm_area` 的具体权限。

- 其中vm_flags的取值需要采用与这里【link】一致的定义方式。
- 其余结构体内具体的成员变量含义已在 `4.2节` 中描述。

建立上述 `vm_area_struct` 数据结构后，需配合 `mmap` 等系统调用为进程初始化一个合理的 `vm_area_struct` 链表，使得后续进程在运行时可同时管理多个 `vm_area`。

## 5.6 实现mmap/munmap/mprotect的SysCall

#### 5.6.1 实现mmap函数

在Linux中，`mmap` 的功能已在4.2.1中阐述。在本次实验中 `mmap` 的功能有所简化，不涉及到文件映射，而是负责分配对应长度的若干个匿名内存页区域，并将这段内存区域的信息插入到当前进程 `current` 中的 `vm_area_struct` 中，作为链表的一个新节点。具体的函数定义及参数含义如下：

```
void *mmap (void *__addr, size_t __len, int __prot,
            int __flags, int __fd, __off_t __offset)
```

- `size_t` 与 `__off_t` 为 `unsigned long` 的类型别名
- `__addr`：**建议**映射的虚拟首地址，需要按页对齐
- `__len`：映射的长度，需要按页对齐
- `__prot`：映射区的权限，在4.3.1节中已说明
- `__flags`：由于本次实验不涉及 `mmap` 在Linux中的其他功能，该参数无意义，固定为 `MAP_PRIVATE | MAP_ANONYMOUS`
  - `MAP_PRIVATE` 表示该映射是私有的，值为 `0x2`
  - `MAP_ANONYMOUS` 表示映射对象为匿名内存页，值为 `0x20`
- `__fd`：由于本次实验不涉及文件映射，该参数无意义，固定为 `-1`
- `__offset`：由于本次实验不涉及文件映射，该参数无意义，固定为 `0`
- 返回值：**实际**映射的虚拟首地址，若虚拟地址区域 `[__addr, __addr+__len)` 未与其它地址映射冲突，则返回值就是建议首地址 `__addr`，若发生冲突，则需要更换到无冲突的虚拟地址区域，返回值为该区域的首地址

`mmap` 映射的虚拟地址区域采用需求分页（Demand Paging）机制，即：`mmap` 中不为该虚拟地址区域分配实际的物理内存，仅当进程试图访问该区域时，通过触发Page Fault，由Page Fault处理函数进行相应的物理内存分配以及页表的映射。

##### 5.6.1.1 实现get_unmapped_area函数

如上文所提到，若建议的虚拟地址区域 `[__addr, __addr+__len)` 与已有的映射冲突，则需要另外寻找无冲突的虚拟地址区域，该过程由 `get_unmapped_area` 函数完成：

```
unsigned long get_unmapped_area(size_t length);
```

我们采用最简单的暴力搜索方法来寻找未映射的长度为 `length`（按页对齐）的虚拟地址区域：从0地址开始向上以 `PAGE_SIZE` 为单位遍历，直到遍历到连续 `length` 长度内均无已有映射的地址区域，将该区域的首地址返回。

#### 5.6.1.2 实现do_mmap函数

我们可以把 `mmap` 函数看作一个封装，实际工作的函数为

```
void *do_mmap(struct mm_struct *mm, void *start, size_t length, int prot);
```

`do_mmap` 函数主要完成了以下几个步骤：

- 检查建议的虚拟地址区域 `[start, start+length)` 是否与已有映射冲突，若冲突，则调用 `get_unmapped_area` 生成新的地址区域
- 为配合需求分页机制，需要创建一个 `vm_area_struct` 记录该虚拟地址区域的信息，并添加到 `mm` 的链表中
  - 调用 `kmalloc` 分配一个新的 `vm_area_struct`
  - 设置 `vm_area_struct` 的起始和结束地址为 `start`， `start + length`
  - 设置 `vm_area_struct` 的 `vm_flags` 为 `prot`
  - 将该 `vm_area_struct` 插入到 `mm` 的链表中
- 返回地址区域的首地址

#### 5.6.1.3 实现mmap函数并添加到系统调用

- `mmap` 实际为 `do_mmap` 的封装，函数体中只需要将其对应参数传参给 `do_mmap` 函数即可：

  ```
  return do_mmap(current->mm, __addr, __len, __prot);
  ```

- 仿照lab5的做法，将 `mmap` 系统调用添加到异常处理函数 `handler_s` 中，系统调用号为222

### 5.6.2 修改task_init函数代码，更改为需求分页机制

在之前的实验中， `task_init` 函数为各个进程初始化时，已经为它们分配好了物理内存空间并进行了页表映射。本次实验开始改用需求分页机制，因此需要对 `task_init` 函数代码进行改动

- 删除之前实验中为进程分配物理内存空间与页表映射的代码
- 调用 `do_mmap` 函数，建立用户进程的虚拟地址空间信息，包括两个区域
  - 代码区域，即用户进程的 `.text` 字段。该区域从虚拟地址0开始，权限为 `PROT_READ | PROT_WRITE | PROT_EXEC`
  - 用户栈。该区域与lab5一样，范围为 `[USER_END - PAGE_SIZE, USER_END)` ，其中 `USER_END = 0xffffffdf80000000` ，权限为 `PROT_READ | PROT_WRITE`

### 5.6.3 实现munmap函数

`munmap` 函数是 `mmap` 函数的逆操作，用于解除 `mmap` 映射的虚拟地址区域：

```
int munmap(void *start, size_t length)
```

若操作成功，则返回0，否则返回-1

#### 5.6.3.1 实现free_page_tables函数

`free_page_tables` 函数相当于lab4中实现的 `create_mapping` 函数的逆操作，其原型如下

```
void free_page_tables(uint64 pagetable, uint64 va, uint64 n, int free_frame);
```

- `pagetable` 为页表基地址，可从 `task_struct` 中读取或直接从 `satp` 获取
- `va` 为需要解除映射的虚拟地址的起始地址， `n` 为需要解除映射的页的数量
- `free_frame` 为布尔值，决定在解除映射时是否将物理页释放

#### 5.6.3.2 实现munmap函数

- 遍历 `current->mm` 中的 `vm_area_struct` 链表，找到虚拟内存区域 `[start, start+length)` 对应的链表节点，若找不到则返回-1
- 调用 `free_page_tables` 接触相应的页表映射并释放对应的物理页， `free_frame` 参数设置为1
- 从 `current->mm` 中的 `vm_area_struct` 链表删除该区域对应的链表节点，并 `kfree` 释放该节点的内存空间
- 仿照lab5的做法，将 `munmap` 系统调用添加到异常处理函数 `handler_s` 中，系统调用号为215

### 5.6.4 实现mprotect函数

在Linux中，使用 mprotect 函数可以用来修改一段指定内存区域（从 void *__addr 开始, 长度为 size_t __len 字节）的内存保护属性值 int __prot

```
int mprotect (void *__addr, size_t __len, int __prot);
```

- mprotect 中使用的 __prot 参数与 mmap 相同
- 遍历页表至第三级子页表，根据 __prot 参数重新设置页表中的权限位
- 仿照lab5的做法，将 mprotect 系统调用添加到异常处理函数 handler_s 中，系统调用号为226

## 5.7 实现 Page Fault 的检测与处理

- 修改 strap.c 文件, 添加对于Page Fault异常的检测

  RISC-V下的中断异常可以分为两类，一类是interrupt，另一类是exception，具体的类别信息可以通过解析scause[XLEN-1]获得。在前面的实验中，我们已经实现在interrupt中添加了时钟中断处理，在exception中添加了用户态程序的系统调用请求处理。在本次实验中，我们还将继续添加对于Page Fault的处理，即当Page Fault发生时，能够根据scause寄存器的值检测出该异常，并调用针对该类异常的处理函数 do_page_fault 。

  当Page Fault发生时scause寄存器可能的取值声明如下：

  ```
  # define CAUSE_FETCH_PAGE_FAULT    12
  # define CAUSE_LOAD_PAGE_FAULT     13
  # define CAUSE_STORE_PAGE_FAULT    15
  ```

  注：可在 strap.c 中实现使得所有类型的Page Fault处理都最终能够调用到同一个 do_page_fault ，然后再在该处理函数中继续判断不同Page Fault的类型，并针对该类Page Fault做进一步的处理。

- 在 fault.c 中实现 Page Fault 处理函数: do_page_fault

  - 根据前文 5.6节 中所描述的 mmap 映射虚拟地址区域所采用的需求分页（Demand Paging）机制。等进程调度结束，首次切换到某一进程的时候，由于访问了未映射的虚拟内存地址，即会触发产生Page Fault异常。
  - do_page_fault 处理函数中具体实现的功能如下：
  - 
    - 读取 csr STVAL 寄存器，获得访问出错的虚拟内存地址（ Bad Address ），并打印出该地址。
    - 检查访问出错的虚拟内存地址（ Bad Address ）是否落在该进程所定义的某一 vm_area 中，即遍历当前进程的 vm_area_struct 链表，找到匹配的 vm_area 。若找不到合适的 vm_area ，则退出Page Fault处理函数，同时打印输出 Invalid vm area in page fault 。
    - 根据 csr SCAUSE 判断Page Fault类型。
    - 根据Page Fault类型，检查权限是否与当前 vm_area 相匹配。
      - 当发生 Page Fault caused by an instruction fetch 时，Page Fault映射的 vm_area 权限需要为可执行；
      - 当发生 Page Fault caused by a read 时，Page Fault映射的 vm_area 权限需要为可读；
      - 当发生 Page Fault caused by a write 时，Page Fault映射的 vm_area 权限需要为可写；
      - 若发生上述权限不匹配的情况，也需要退出Page Fault处理函数，同时打印输出 Invalid vm area in page fault 。
      - 只有匹配到合法的 vma_area 后，才可进行相应的物理内存分配以及页表的映射。
    - 最后根据访问出错的 Bad Address ，调用Lab4中实现过的 create_mapping 实现新的页表映射关系。注意此时的Bad Address不一定恰好落在内存的4K对齐处，因此需要稍加处理，取得合适的虚拟内存和物理内存的映射地址以及大小。

## 5.8 实现 fork 系统调用

### 5.8.1 修改task_struct

修改task_struct，添加成员：

```
uint64 *stack ;
```

stack 成员用于保存异常发生时的寄存器状态，会在fork函数中使用。可以在每次异常处理 handler_s 触发时都将当前的寄存器状态保存到 stack 成员中。

### 5.8.2 实现fork函数

fork 函数的函数原型如下：

```
uint64 fork(void)
```

- 父进程 fork 成功时 返回：子进程的pid ，子进程 返回：0 。 fork 失败则父进程 返回：-1 。

fork 具体的操作如下：

- 创建子进程，仿照lab5的方式，设置好 task_struct 中的 sp , sepc , sstatus 和 sscratch 成员。根据lab5设置的不同， sscratch 成员可能设置为陷入异常处理的用户态sp或其他值。
- 为子进程 task_struct 中的 stack 成员分配空间，并将父进程的 stack 内容拷贝进去，注意为了让子进程的返回值为0，需要修改寄存器a0状态。
- 为子进程分配并拷贝父进程的 mm_struct 。
- 为子进程新建页表，为了适应page_fault，这里同样只映射内核的映射。
- 拷贝用户栈。我们需要分配一个页的空间并将父进程的用户栈即 [USER_END - PAGE_SIZE, USER_END) 拷贝进去，作为子进程的用户栈。子进程的用户栈起始地址需要保存到 task_struct 中，等待page_fault为其建立映射（5.8.5节）。
- 设置 task_struct 中的 ra 成员为5.8.3中的 forkret 函数。
- 返回子进程的 pid ，结束。
- 仿照lab5的做法，将 fork 系统调用添加到异常处理函数 handler_s 中，系统调用号为220。

#### 5.8.3 实现forkret函数

这是完成fork后，context switch到达子进程时该进程返回到的c语言函数，它的作用主要是调用5.8.4中的汇编函数 ret_form_fork 使fork结束的子进程返回到用户模式。

具体操作：

- 调用5.8.4中的汇编函数 ret_form_fork 返回到用户模式。

#### 5.8.4 实现ret_from_fork函数

这是一个汇编函数，函数原型如下：

```
extern void ret_from_fork(uint64 *stack);
```

- 该函数主要是读取 stack 中的所有寄存器状态并返回到用户模式，调用时 stack 是当前 task_struct 中的成员。
- 读取寄存器时需要注意读取的顺序，以及lab5中加入的 sscratch 机制，保证系统正常运行。

#### 5.8.5 修改 Page Fault 处理

在之前的Page Fault处理中，我们对用户栈Page Fault处理方法是自由分配一页作为用户栈并映射到 [USER_END - PAGE_SIZE, USER_END) 的虚拟地址。但由fork创建的进程，它的用户栈已经拷贝完毕，因此Page Fault处理时直接为该页建立映射即可。

#### 5.8.6 修改task_init函数代码，使fork正常运行

我们新提供的user.bin中添加了两次 fork 调用，为了使其正常运行，我们需要在 task_init 函数中修改为仅初始化一个进程，其余三个进程均通过 fork 创建。

### 5.9 编译及测试

本次实验的输出结果参考如下：

```
ZJU OS LAB 6          GROUP-XX
[S] Create New cache: name:slub-objectsize-8     size: 0x00000008, align: 0x00000008
[S] Create New cache: name:slub-objectsize-16    size: 0x00000010, align: 0x00000008
[S] Create New cache: name:slub-objectsize-32    size: 0x00000020, align: 0x00000008
[S] Create New cache: name:slub-objectsize-64    size: 0x00000040, align: 0x00000008
[S] Create New cache: name:slub-objectsize-128   size: 0x00000080, align: 0x00000008
[S] Create New cache: name:slub-objectsize-256   size: 0x00000100, align: 0x00000008
[S] Create New cache: name:slub-objectsize-512   size: 0x00000200, align: 0x00000008
[S] Create New cache: name:slub-objectsize-1024  size: 0x00000400, align: 0x00000008
[S] Create New cache: name:slub-objectsize-2048  size: 0x00000800, align: 0x00000008
[S] Buddy allocate addr: ffffffff80023000
[S] Buddy allocate addr: ffffffff80098000
[S] Buddy allocate addr: ffffffff80099000
[S] kmem_cache_alloc: name: slub-objectsize-64
addr: ffffffff8008000000000008, partial_obj_count: 1
[S] kmem_cache_alloc: name: slub-objectsize-64
```

```
addr: ffffffff8008004000000008, partial_obj_count: 2
[S] New vm_area_struct: start 0000000000000000, end 00000000000000fd, prot [r:1,w:1,x:1]
[S] kmem_cache_alloc: name: slub-objectsize-64
addr: ffffffff8008008000000008, partial_obj_count: 3
[S] New vm_area_struct: start fffffffeffffff000, end ffffffff00000000, prot [r:1,w:1,x:0]
[PID = 1] Process Create Successfully! counter = 3
[PID = 0] Context Calculation: counter = 1
[!] Switch from task 0[ffffffff80009050] to task 1[ffffffff80009148], prio: 5, counter: 3
[S] PAGE_FAULT: PID = 1
[S] PAGE_FAULT: scause: 12, sepc: 0x00000000, badaddr: 0x00000000
[S] mapped PA :0000000080008000 to VA :0000000000000000 with size :00000000000000fd
[S] PAGE_FAULT: PID = 1
[S] PAGE_FAULT: scause: 15, sepc: 0x000000b4, badaddr: 0xfffffff8
[S] Buddy allocate addr: ffffffff8009c000
[S] mapped PA :000000008009c000 to VA :ffffffffeffff000 with size :0000000000001000
[S] Buddy allocate addr: ffffffff8009f000
[S] kmem_cache_alloc: name: slub-objectsize-256
addr: ffffffff8008800000000008, partial_obj_count: 1
[S] kmem_cache_alloc: name: slub-objectsize-64
addr: ffffffff800800c000000008, partial_obj_count: 4
[S] Buddy allocate addr: ffffffff800a0000
[S] Buddy allocate addr: ffffffff800a1000
[PID = 2] Process fork from [PID = 1] Successfully! counter = 3
[S] Buddy allocate addr: ffffffff800a2000
[S] kmem_cache_alloc: name: slub-objectsize-256
addr: ffffffff8008810000000008, partial_obj_count: 2
[S] kmem_cache_alloc: name: slub-objectsize-64
addr: ffffffff8008010000000008, partial_obj_count: 5
[S] Buddy allocate addr: ffffffff800a3000
[S] Buddy allocate addr: ffffffff800a4000
[PID = 3] Process fork from [PID = 1] Successfully! counter = 3
pid: 1
[PID = 1] Context Calculation: counter = 3
[PID = 1] Context Calculation: counter = 2
[PID = 1] Context Calculation: counter = 1
[!] Switch from task 1[ffffffff80009148] to task 3[ffffffff80009338], prio: 5, counter: 3
[S] PAGE_FAULT: PID = 3
[S] PAGE_FAULT: scause: 12, sepc: 0x00000094, badaddr: 0x00000094
[S] mapped PA :0000000080008000 to VA :0000000000000094 with size :00000000000000fd
[S] PAGE_FAULT: PID = 3
[S] PAGE_FAULT: scause: 15, sepc: 0x00000098, badaddr: 0xffffffc8
[S] mapped PA :00000000800a4000 to VA :ffffffffeffff000 with size :0000000000001000
pid: 3
[PID = 3] Context Calculation: counter = 3
[PID = 3] Context Calculation: counter = 2
[PID = 3] Context Calculation: counter = 1
[!] Switch from task 3[ffffffff80009338] to task 2[ffffffff80009240], prio: 5, counter: 3
[S] PAGE_FAULT: PID = 2
[S] PAGE_FAULT: scause: 12, sepc: 0x00000094, badaddr: 0x00000094
[S] mapped PA :0000000080008000 to VA :0000000000000094 with size :00000000000000fd
[S] PAGE_FAULT: PID = 2
[S] PAGE_FAULT: scause: 15, sepc: 0x00000098, badaddr: 0xffffffc8
[S] mapped PA :00000000800a1000 to VA :ffffffffeffff000 with size :0000000000001000
[S] Buddy allocate addr: ffffffff800ad000
[S] kmem_cache_alloc: name: slub-objectsize-256
addr: ffffffff8008820000000008, partial_obj_count: 3
[S] kmem_cache_alloc: name: slub-objectsize-64
addr: ffffffff8008014000000008, partial_obj_count: 6
[S] Buddy allocate addr: ffffffff800ae000
[S] Buddy allocate addr: ffffffff800af000
[PID = 4] Process fork from [PID = 2] Successfully! counter = 3
pid: 2
[PID = 2] Context Calculation: counter = 3
[PID = 2] Context Calculation: counter = 2
[PID = 2] Context Calculation: counter = 1
[!] Switch from task 2[ffffffff80009240] to task 4[ffffffff80009430], prio: 5, counter: 3
[S] PAGE_FAULT: PID = 4
[S] PAGE_FAULT: scause: 12, sepc: 0x00000094, badaddr: 0x00000094
[S] mapped PA :0000000080008000 to VA :0000000000000094 with size :00000000000000fd
[S] PAGE_FAULT: PID = 4
[S] PAGE_FAULT: scause: 15, sepc: 0x00000098, badaddr: 0xffffffc8
```

```
[S] mapped PA :00000000800af000 to VA :ffffffffefffff000 with size :0000000000001000
pid: 4
[PID = 4] Context Calculation: counter = 3
[PID = 4] Context Calculation: counter = 2
[PID = 4] Context Calculation: counter = 1
[*] Reset task 1 counter 5
[*] Reset task 2 counter 5
[*] Reset task 3 counter 5
[!] Switch from task 4[ffffffff80009430] to task 3[ffffffff80009338], prio: 5, counter: 5
pid: 3
[PID = 3] Context Calculation: counter = 5
[PID = 3] Context Calculation: counter = 4
[PID = 3] Context Calculation: counter = 3
pid: 3
[PID = 3] Context Calculation: counter = 2
[PID = 3] Context Calculation: counter = 1
[!] Switch from task 3[ffffffff80009338] to task 2[ffffffff80009240], prio: 5, counter: 5
pid: 2
[PID = 2] Context Calculation: counter = 5
[PID = 2] Context Calculation: counter = 4
[PID = 2] Context Calculation: counter = 3
pid: 2
[PID = 2] Context Calculation: counter = 2
[PID = 2] Context Calculation: counter = 1
[!] Switch from task 2[ffffffff80009240] to task 1[ffffffff80009148], prio: 5, counter: 5
pid: 1
[PID = 1] Context Calculation: counter = 5
[PID = 1] Context Calculation: counter = 4
[PID = 1] Context Calculation: counter = 3
pid: 1
[PID = 1] Context Calculation: counter = 2
[PID = 1] Context Calculation: counter = 1
[*] Reset task 2 counter 5
[*] Reset task 3 counter 5
[*] Reset task 4 counter 5
[!] Switch from task 1[ffffffff80009148] to task 4[ffffffff80009430], prio: 5, counter: 5
pid: 4
...
```

## 6. 实验任务与要求

请仔细阅读背景知识，完成 5.1 至 5.9的内容，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 实验报告：
  - 各实验步骤截图以及结果分析
  - 回答思考题
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（确保make clean）
  - 将lab6_319010XXXX目录压缩成zip格式（若分组，则一组只需要一位同学提交）
- 最终提交
  - 将报告和工程压缩包提交至学在浙大

# lab7

# Lab 7: A Toy RISC-V OS

## 1 实验简介

利用给定的文件系统和Shell程序，实现从命令行运行给定的用户程序。

## 2 实验目标

- 了解文件系统的实现，学会使用给定的文件系统读取文件内容。
- 理解ELF文件格式，实现简单的ELF Loader，能够解析ELF文件，并将需要的segment装载到内存。
- 实现exec，wait，exit系统调用。
- 使用给定的Shell，能够解析命令行输入，获取需要执行的用户态程序名，运行给定的用户态程序。

## 3 实验环境

- Docker Image

## 4 背景知识

### 4.1 initramfs

文件系统是一套实现了数据的存储、分级组织、存取和获取等操作的抽象数据类型，其底层的操作的文件可能存储在物理的光盘、硬盘上，也可能是来自于网络、内存。在本实验中，我们使用的是initranfsl临时文件系统，进一步的，是使用了newc规范cpio格式的镜像。 在Linux启动的过程中，所有配置信息、工具、库等文件会被存入initramfs中，内核将创建一个tmpfs文件系统访问其中的内容，主要是使用/init来完成对内核的初始化，同学们可以观察自己当前使用的Linux中使用了哪个程序/脚本作为init：

```
$ sudo cat /var/log/kern.log | grep "/init"
Jan  3 17:08:55 phantom-workstation kernel: [    0.954328] Run /init as init process

### 若使用了 /init，且根目录下没有init文件，可以检查/boot/grub/grub.cfg中使用了哪个initrd镜像
$ sudo unmkinitramfs -v /boot/initrd.img-5.4.0-59-generic  .

$ tail main/init
...
exec run-init ${drop_caps} "${rootmnt}" "${init}" "$@" <"${rootmnt}/dev/console" >"${rootmnt}/dev/console" 2>&1
...
```

可以看到，init进程一旦挂载了根文件系统和其他重要文件系统，就会将根切换到真实的根文件系统，并最终在该系统上调用/sbin/init（${init}）二进制文件以继续启动过程，有关cpio newc的相关介绍参考此处。

在本实验中我们提供了一个简单的cpio读取demo，同学们需要在使用时需要自行完成映射，initramfs的物理位置与lab5中一致，位于0x84000000，完成映射后使用namei定位文件，readi读取文件。 namei和readi函数的声明如下：

```
struct inode *namei(char *path);
int readi(struct inode *ip, int user_dst, void *dst, uint off, uint n);
```

### 4.2 ELF文件格式与Loader

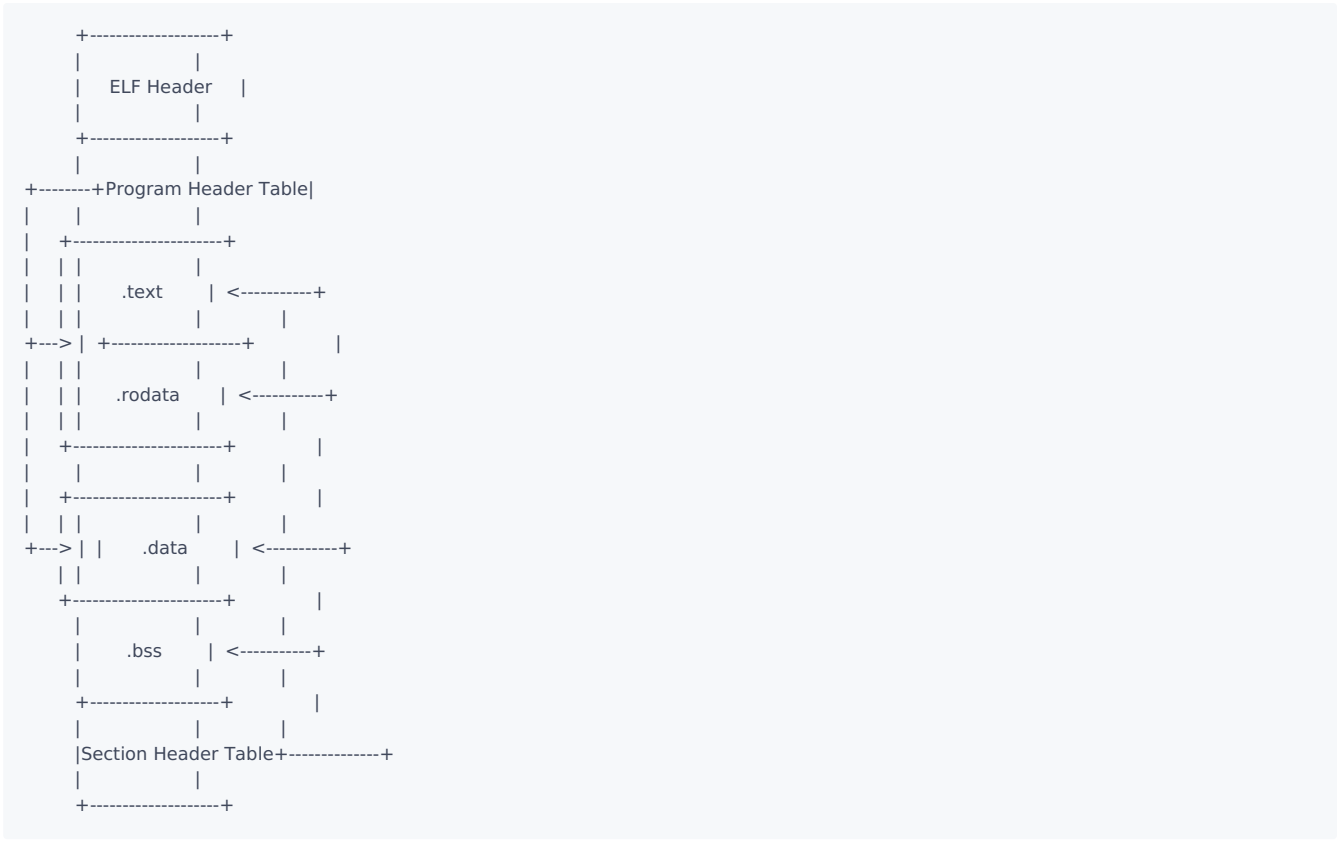
- 本部分有关ELF和Loader的内容参考了《程序员的自我修养--链接、装载与库》，【man elf】以及【Executable and Linkable Format】

#### 4.2.1 什么是ELF文件格式

可执行与可链接格式，即Executable and Linkable Format，缩写为ELF，常被称为ELF格式，在计算机科学中，是一种用于可执行文件、目标文件、共享库和核心转储(core dump)的标准文件格式。1999年，被86open项目选为x86架构上的类Unix操作系统的二进制文件格式标准，用来取代COFF。因其可扩展性与灵活性，也可应用在其它处理器、计算机系统架构的操作系统上。

#### 4.2.2 ELF Header

下图描述了ELF文件的总体结构，我们省去了ELF一些繁琐的结构，把最重要的结构提取出来。




ELF文件格式的最前部是ELF文件头（ELF Header），它描述了整个文件的基本属性，比如ELF文件版本，目标机器型号，程序入口地址等。紧接着是ELF文件各个section。 我们可以用readelf命令来详细查看ELF文件：

```
# 查看ELF文件头
$ riscv64-unknown-elf-readelf -h hello.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:                           RISC-V
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13576 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         1
  Size of section headers:           64 (bytes)
  Number of section headers:         22
  Section header string table index: 21
```

从上面的输出结果可以看到，在ELF的文件头中定义了ELF魔数（0x464C457F），文件机器字节长度（64bit），数据存储方式（小端），文件类型（EXEC），机器类型（RISC-V），ELF程序的入口虚拟地址（0x0），program header table的位置、长度、其中包含的program header的数量，section header table的位置、长度、其中包含的sectio header的数量等信息。有关section header与program header的区别我们将在下一小节介绍。 我们用以下的结构体 `elfhdr` 来解析ELF header：

```
/* elf.h */
// ELF file header struct
struct elfhdr {
    uint magic;          // must equal ELF_MAGIC, i.e., 0x464C457F
```

```
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint64 entry;
    uint64 phoff;       // program header table距离ELF文件开始的偏移(单位：字节)
    uint64 shoff;       // section header table距离ELF文件开始的偏移(单位：字节)
    uint flags;
    ushort ehsize;
    ushort phentsize;   // 单个program header的大小(单位：字节)
    ushort phnum;       // program header数量
    ushort shentsize;   // 单个section header的大小(单位：字节)
    ushort shnum;       // section header数量
    ushort shstrndx;
};
```

#### 4.2.3 ELF文件的链接视图和执行视图

我们可以从链接视图和执行视图来分别理解一个ELF文件。

**链接视图**

ELF文件的section header table包含了ELF文件中的sections。在链接的时候，链接器会合并相同性质的section，比如将所有输入文件的".text"合并到输出文件的".text"段，接着是".rodata"段，".data"段，".bss"段等。我们可以用readelf查看一个ELF文件的section信息，例如：

```
# 查看ELF文件的链接视图
$ riscv64-unknown-elf-readelf -S hello
There are 15 section headers, starting at offset 0x4d38:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000         0     0     0
  [ 1] .text             PROGBITS        00000000000100b0  000000b0
       00000000000025fc  0000000000000000  AX       0     0     2
  [ 2] .rodata           PROGBITS        00000000000126b0  000026b0
       0000000000000012  0000000000000000   A       0     0     8
  [ 3] .eh_frame         PROGBITS        00000000000136c4  000026c4
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .init_array       INIT_ARRAY      00000000000136c8  000026c8
       0000000000000010  0000000000000008  WA       0     0     8
  [ 5] .fini_array       FINI_ARRAY      00000000000136d8  000026d8
       0000000000000008  0000000000000008  WA       0     0     8
  [ 6] .data             PROGBITS        00000000000136e0  000026e0
       0000000000000f58  0000000000000000  WA       0     0     8
  [ 7] .sdata            PROGBITS        0000000000014638  00003638
       0000000000000040  0000000000000000  WA       0     0     8
  [ 8] .sbss             NOBITS          0000000000014678  00003678
       0000000000000028  0000000000000000  WA       0     0     8
  [ 9] .bss              NOBITS          00000000000146a0  00003678
       0000000000000060  0000000000000000  WA       0     0     8
  [10] .comment          PROGBITS        0000000000000000  00003678
       0000000000000012  0000000000000001  MS       0     0     1
  [11] .riscv.attributes RISCV_ATTRIBUTE  0000000000000000  0000368a
       0000000000000035  0000000000000000         0     0     1
  [12] .symtab           SYMTAB          0000000000000000  000036c0
       0000000000000f60  0000000000000018        13    79     8
  [13] .strtab           STRTAB          0000000000000000  00004620
       0000000000000698  0000000000000000         0     0     1
  [14] .shstrtab         STRTAB          0000000000000000  00004cb8
       000000000000007e  0000000000000000         0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

**执行视图**

当可执行文件被装载到内存空间时，如果操作系统以ELF文件的section为单位，为每个section分配单独的内存，那么会造成内存的浪费。【思考题，为什么会造成内存浪费】 其实操作系统装载可执行文件时，并不关心可执行文件各个段所包含的实际内容，操作系统只关心一些跟装载相关的问题，最主要的是section的权限（可读、可写、可执行），而section的权限往往只有为数不多的几种组合，即：

- 可读可执行，以.text段为代表
- 可读可写，以.data, .bss段为代表
- 只读数据段，以.rodata段为代表

因此，为了节省空间，操作系统会把权限相同的sections合并到一起当作一个**segment**进行映射。我们同样可以用readelf查看ELF文件的segment信息：

```
# 查看ELF文件的执行视图
$ riscv64-unknown-elf-readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x100c2
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  LOAD           0x0000000000000000 0x0000000000010000 0x0000000000010000
                 0x00000000000026c2 0x00000000000026c2  R E    0x1000
  LOAD           0x00000000000026c4 0x00000000000136c4 0x00000000000136c4
                 0x0000000000000fb4 0x000000000000103c  RW     0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .eh_frame .init_array .fini_array .data .sdata .sbss .bss
```

可执行文件 `hello` 有2个segment，第1个segment包含了.data .sdata .sbss .bss等这些可读可写的section。【思考题：为什么第0个segment会将只读的.rodata赋给执行权限？】

描述segment的结构体叫做程序头（program header），它描述了ELF文件该如何被操作系统映射到进程的虚拟虚拟空间。我们可以用结构体 `proghdr` 来解析program header。结构体每个成员的含义也已给出：

```
/* elf.h */
// ELF program header struct
struct proghdr {
    uint32 type;   // type的值决定了segment的类型，在本实验中我们只需要关注LOAD类型的segment，即type值为1的segment
    uint32 flags;  // segment的权限属性，包括R,W,X
    uint64 off;    // segment在ELF文件中的偏移(单位：字节)
    uint64 vaddr;  // segment的第一个字节在进程虚拟地址空间的起始位置。整个program header table中，所有“LOAD”类型的segment按照vaddr从小到大排
    uint64 paddr;  // segment的物理装载地址，一般与vaddr一样
    uint64 filesz; // segment在ELF文件中所占空间的长度(单位：字节)
    uint64 memsz;  // segment在进程虚拟地址空间所占用的长度(单位：字节)
    uint64 align;  // segment的对齐属性。实际对齐字节等于2的align次。例如当align等于10，那么实际的对齐属性就是2的10次方，即1024字节
};
```

#### 4.2.4 可执行文件的装载

可执行文件需要装载到内存中才能成为进程，并被CPU执行。可执行文件装载的流程如下： (1)创建一个独立的虚拟地址空间 (2)读取可执行文件头，为进程分配物理内存，读取并复制可执行文件的LOAD类型segment到进程虚拟地址空间中 (3)将PC寄存器设置为可执行文件的入口地址，启动运行

【注意】本实验与Linux的实现有所不同，在读取并复制可执行文件的segment之前分配好了物理内存

### 4.3 系统调用

沿用Lab5中添加syscall的方法，本次实验中要求实现的系统调用函数原型以及具体功能如下：

#### 4.3.1 实现281号系统调用: exec

```
// 同学们可直接复制sys_exec()以及exec()
```

```
uint64 sys_exec(const char *path) {
    int ret = exec(path);
    return ret;
}

int exec(const char *path) {
    int ret = proc_exec(current, path);
    return ret;
}

// 需要同学们实现proc_exec函数
int proc_exec(struct task_struct *proc, const char *path) {
    ...
}
```

- exec的功能是将当前进程正在执行的可执行文件 替换为 一个新的可执行文件。在Shell中，先执行fork，复制父进程的虚拟内存空间到子进程；然后子进程再执行exec，替换为新的可执行文件。
- 函数参数 `proc` 表示当前进程的task_struct, 参数 `path` 表示可执行文件的路径
- 执行成功返回0，执行失败则返回-1

#### 4.3.2 实现260号系统调用:wait, 93号系统调用exit

- 考虑到本次实验没有添加信号机制，wait以及exit的实现由我们自行设计，满足简单的功能。同时需要同学们自行分析所需参数及返回值，只要实现父进程可以等待子进程的功能即可。
- wait系统调用的简单的伪代码描述如下：

```
int sys_wait(uintptr_t *regs) {
    set self as pending state;
     schedule;
}
```

- wait系统调用的功能是将当前进程的state设置为 `TASK_SLEEPING`
- 执行成功返回0，执行失败则返回-1
- exit系统调用的简单的伪代码描述如下：

```
void sys_exit() {
     while(true) {
         if( parent is pending )
             break;
         else
             schedule to give up cpu;
     }
     set parent as running state;
     schedule to give up cpu;
}
```

- 更多内容可以参考【Linux系统调用号】

### 4.4 Shell原理

本次使用的shell为一个简单的user program，实现简单的shell功能，并进行简单的交互。其简单的伪代码描述如下：

```
int main(void)
{
 while(1) {
  printf("$ ");
  if( gets(buf) < 0)
   break;
  if(fork() == 0) {
   exec(buf);
  }
  wait(0);
 }
 exit(0);
```

```
}
```

# 5 实验步骤

## 5.1 环境搭建

### 5.1.1 建立映射

同lab5的文件夹映射方法，目录名为lab7

### 5.1.2 组织文件结构

```
.
├── arch
│   └── riscv
│       ├── include
│       │   ...
│       │   └── elf.h
│       ├── kernel
│       │   ...
│       │   └── exec.c
│       └── Makefile
├── driver
│   ...
├── fs
│   ├── cpio.c
│   └── Makefile
├── include
|   ...
│   └── fs.h
├── init
│   ...
├── lib
│   ...
├── Makefile
└── user
    ├── *.h
    ├── Makefile
    └── shell.c
```

## 5.2 实现简单的ELF Loader

实现简单的ELF Loader实际上就是实现一个loadseg函数，将 `LOAD` 类型的segment的内容复制到内存的某一个位置，其伪代码描述如下：

```
int loadseg(pagetable_t pagetable, uint64 va, struct inode *ip, uint offset, uint filesz) {
    check whether va is aligned to PAGE_SIZE;
    for(i=0; i<filesz; i+=PAGE_SIZE) {
        walk the pagetable, get the corresponding pa of va;
        use readi() to read the content of segment to address pa;
    }
}
```

参数说明：

- pagetable: page table to map the va of the segments
- va: proghdr.vaddr
- ip: point to the inode of the elf file
- offset: proghdr.offset
- filesize: proghdr.filesz

## 5.3 实现exec系统调用

函数proc_exec简单的伪代码描述如下

```
int proc_exec(struct task_struct *proc, const char *path) {
    inode = namei(path);
    create a new page table newpgtbl, copy kernel page table from kpgtbl (same as lab5);
    readi() read the elf header(denoted as elf_header) from elf file;
    check whether elf_header.magic == ELF_MAGIC;
    for(i=0; i<elf_header.phnum; i++) {
        readi() read the program header(denoted as prog_header) of each segment;
        check whether prog_header.type == LOAD;
        parse_ph_flags() parse the prog_header.flags to permissions;
        uvmalloc() allocate user pages for [prog_header.vaddr, prog_header.vaddr+prog_header.memsz] of this segment in newpgtbl, and set prope
        loadseg() copy the content of this segment from prog_header.off to its just allocated memory space;
    }
    allocate a page for user stack and update the newpgtbl for it. The user stack va range: [USER_END-PAGE_SIZE, USER_END];
    set proc's sstatus, sscratch, and sepc like task_init() in Lab5;
    set proc->pgtbl to newpgtbl;
}
```

### 5.4 实现exit, wait系统调用

考虑到本次实验没有添加信号机制，wait以及exit的实现由我们自行设计，满足简单的功能。简单的伪代码描述如下：

```
sys_exit() {
    while(true) {
        if( parent is pending )
            break;
        else
            schedule to give up cpu;
    }
    set parent as running state;
    schedule to give up cpu;
}
sys_wait() {
    set self as pending state;
    schedule;
}
```

同时需要同学们自行分析所需参数及返回值，只要实现父进程可以等待子进程的功能即可。

### 5.5 修改init.c，调用给定的Shell

启动后内核运行在S-mode，需要切换到U-mode来执行用户程序。本实验我们实现了fork和exec系统调用，但是这些系统调用需要在用户态调用，那么如何创建第一个用户态进程呢？

与lab 5相近，我们先分配一个task struct，建立用户态程序的映射，根据用户态程序的入口设置好task struct里保存的sepc、准备好寄存器状态使得切换上下文后能返回到用户态程序入口处继续执行，把进程状态设为ready后由调度器调度到用户态程序执行。

在lab 5中，我们使用objcopy来跳过了ELF loader的部分，直接建立到binary的映射来实现用户态进程的创建。本实验与lab 5不同的是，映射的建立要求通过ELF loader来实现。具体来说需要如下的流程：

```
elf_loader(task, path)
{
    read file (path) in cpio
    load elf file to page table of task
    set sepc to entry
    set sstatus, sscratch ...
}

task_init()
{
    allocate task struct
    elf_loader(task, "shell") // "shell" is the path of shell
    set task status to READY
}
```

### 5.6 参考输出

```
$ hello
[PID = 2] Process fork from [PID = 1] Successfully! counter = 2
[!] Switch from task 1[ffffffff80009180] to task 2[ffffffff800092b0], prio: 3, counter: 2
[User] pid: 2, sp is fffffffeffffffe0
[User] pid: 2, sp is fffffffeffffffe0
[!] Switch from task 2[ffffffff800092b0] to task 0[ffffffff80009050], prio: 5, counter: 1
[!] Switch from task 0[ffffffff80009050] to task 2[ffffffff800092b0], prio: 3, counter: 3
[User] pid: 2, sp is fffffffeffffffe0
[User] pid: 2, sp is fffffffeffffffe0
```

*注：输出的地址信息、打印数量仅供参考 由于没有实现回显，在输入 `hello` 命令的时候Shell并没有显示，但实际上已经输入了命令*

## 6 实验任务与要求

请仔细阅读背景知识，完成 5.1 至 5.6 的内容，撰写实验报告，需提交实验报告以及整个工程的压缩包。

- 实验报告：
  - 各实验步骤截图以及结果分析
  - 回答思考题(2个)
  - 实验结束后心得体会
  - 对实验指导的建议（可选）
- 工程文件
  - 所有source code（确保make clean）
  - 将lab7_319010XXXX目录压缩成zip格式（若分组，则一组只需要一位同学提交）
- 最终提交
  - 将报告和工程压缩包**分别**提交至学在浙大