操作系统课程设计

本次实验的要求：

1. 本课程是实验课程，请每位同学确保在规定时间内完成实验。
2. 本课程成绩由两部分构成，验收成绩占 70%，实验报告成绩占 30%。实验结束时，由老师上机验收成果，同学们提交实验报告。
3. 本课程设置了三类实验：操作系统模拟实验、操作系统内核实验和复现操作系统实验，模拟实验难度低于内核实验和复现实验，同学们可以根据自己的情况进行选择。评分会考虑考虑难度因素，但是完成内核实验和复现实验的同学的得分并不一定就高于完成模拟实验的同学。
4. 实验报告提交 word 格式或 PDF 格式文档，实验报告撰写要求参见“操作系统实验报告撰写要求”文档。用 word 编辑实验报告的同学请在文档结构图中保留目录结构。复现实验报告不要抄书，请用自己理解的流程来描述实现方法和过程。所有的内容均写在一个实验报告中，每个实验报告文档需要有摘要、目录和参考文献，还需要记录在本次实验中遇到的问题及解决方案。实验报告不是写的内容越

多越好，请控制在 50 页左右。请保留封面的成绩评分表。

5. 提交方式：请大家加入雨课堂，并根据自己完成的实验类别在正确的页面提交实验报告。实验报告的命名方式为：“学号-姓名-操作系统**实验报告”，其中**指的是模拟、内核、或复现。请在截止时间前提交，不接受截止时间之后通过 QQ 或邮箱单独提交报告。如果发现提交的报告版本有误，请在截止日期前及时联系我退回报告。只提供一次退回机会，所以请准备好了再提交。每个人有责任自己检查是否提交了报告，以及提交的报告是否正确。在截止日期后如果没有发现该同学的报告，我不会再单独提醒。
6. 以下情况会被额外扣分：
   - 没有在报告封面保留成绩评分表
   - 没有按要求命名实验报告文件
   - 实验报告内容不完整，格式不符合要求

操作系统模拟实验

实习一　进程调度实验

一、实习内容

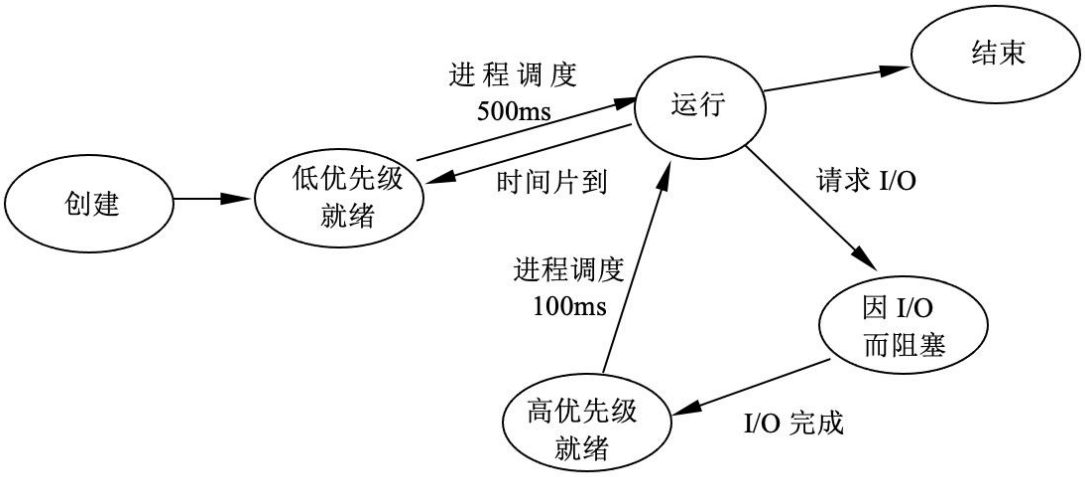
实现进程调度及进程状态转换。

二、实习目的

本实习模拟在单处理器环境下的进程调度及状态转换，加深对进程调度、进程状态及其状态转换的理解。

三、实习题目

请设计一个将优先级调度与时间片调度相结合的调度算法，能够实现如下图所示的进程状态变迁。



系统中有低优先就绪和高优先就绪两种就绪情况。进程被创建之后进

入低优先就绪队列。若一个进程不能在一个时间片内完成，则在时间片用完后进入低优先就绪队列。一个运行的进程如果申请 I/O，则从运行态变为 I/O 阻塞状态，让出 CPU 并进入 I/O 阻塞队列。若一个被阻塞的进程 I/O 完成，则从 I/O 阻塞状态变为就绪状态，且进入高优先就绪队列。当 CPU 空闲时，首先从高优先就绪队列中选择进程运行，给定时间片 100ms；若高优先就绪队列为空，则从低优先就绪队列中选择进程运行，给定时间片 500ms。系统中的进程调度是非抢占式的，当一个进程正在运行时，若有更高优先级的进程进入高优先级就绪队列，此时该进程不会让出 CPU。

请设计合适的 PCB 结构，实现进程的创建、切换、阻塞和终止功能，并通过设计良好的测试用例展示执行以上功能时，进程的 PCB 状态及进程调度过程中各队列的情况。

【要求】：

(1) 根据本实习题目的功能要求，设计合适的进程 PCB 结构；

(2) 系统中需设置三个队列，低优先级队列、高优先级队列和 I/O 阻塞队列。为了简化设计，假设所有因 I/O 而被阻塞的进程都在一个 I/O 阻塞队列中排队；

(3) 进程的整个执行过程是由计算和 I/O 操作交替组成，在创建进程时，需指出该进程是由哪些操作组成的，每个操作持续多长时间。比如一个计算繁忙型的进程由以下操作序列组成：

- **计算 550ms**
- **I/O 50ms**

- 计算 600ms

一个 I/O 繁忙型的进程由以下操作序列组成：

- 计算 100ms
- I/O 200ms
- 计算 50ms
- I/O 100ms
- 计算 50ms
- I/O 100ms
- 计算 10ms

一个进程完成 I/O 需要的时长是不确定的，它取决于系统中 I/O 设备的使用情况。大家可以在创建进程时给一个假设的时长，也可以在出现调度时机时，由用户通过键盘指定哪个进程的 I/O 已完成。

【验收标准】：

(1) 实现的调度算法正确、公平，不存在饥饿或死锁；

(2) 进程状态切换正确，调度过程不丢失上下文，且可以展示进程的上下文；

(3) PCB 设计合理，对进程的创建、切换、终止、阻塞等控制功能齐全；

(4) 有可视性好的用户界面，有良好的测试用例，测试结果符合预期；

(5) 鼓励在实验中加入新的观点或想法，并加以实现。

实习二　虚拟内存管理实验

一、实习内容

模拟请求分页内存管理的功能，包括实现地址映射、页面置换（clock）、空闲页面管理等。

二、实习目的

通过本实习帮助同学们理解请求分页系统的主要管理机制。

三、实习题目

假设物理内存有 256M，虚拟内存有 4GB，请设计基于分页的虚拟内存管理机制，包括地址映射、页面置换（clock）、页面的分配与回收等。

【要求】：

(1) 可以指定物理页和逻辑页的大小，比如 4KB，后续的地址映射、页面置换以及内存的分配与回收都是基于这个设定的页面大小进行的；

(2) 设计虚拟页面到物理页面的映射机制。要求支持动态映射，能够加载和替换物理页面；

(3) 给定一个逻辑地址，如果该逻辑地址所在的逻辑页在内存中，则可以通过页表正确映射到相应的物理地址，若该逻辑地址所在的逻辑页不在内存中，则判断该页面失效，需要将该逻辑页加载到内存，并修改对应的页表项；

(4) 实现 clock 页面置换算法，当分配的物理页面已装满时，若要访问

未加载的逻辑页面，能够根据 clock 算法淘汰合适的页面；

(5) 空闲页面采用位示图进行管理；

(6) 设计测试程序，编写合适的测试样例验证你的虚拟内存系统的各种机制，要求测试不同访问序列下，页面加载、替换的正确性，以及页面失效的检测处理。

(7) 鼓励在实验中加入新的观点或想法，并加以实现。

【验收标准】：

(1) 地址映射算法实现正确；

(2) 页面置换算法实现正确；

(3) 模拟进程从申请内存到释放内存的过程中可能遇到的各种操作，包括内存分配、地址变换、页面访问失效、页面置换、内存回收等功能；

(4) 有可视性好的用户界面，有良好的测试用例，测试结果符合预期。

实习三　文件系统

一、实习内容

设计一个简单的两级目录文件系统。

二、实习目的

通过简单文件系统的设计实现，加深对文件系统的内部数据结构、功能及实现过程的理解。

三、实习题目

设计一个简单的两级目录文件系统。

【要求】：

(1) 假设有 **8** 个用户，每个用户保存文件数不超过 **10** 个，所有用户可同时打开文件数不超过 **16**。

(2) 二级目录结构包含主目录及用户文件目录。主目录应包含用户名、用户文件目录指针等字段，用户文件目录应包含文件名、保护字段（包含三个权限读、写、执行，分别用 **R**、**W**、**X** 表示）、文件长度、文件存放地址等字段，系统打开文件目录结构包含用户文件目录结构的相关字段并增加文件读写指针字段。文件的物理结构采用链接结构（也可以使用其他物理结构）。

(3) 文件可以用内存映射文件方式实现，即通过将磁盘块映射到内存中的一页，允许文件 **I/O** 被处理为常规内存访问。在退出系统时，相关

文件内容及目录内容应以文件形式存放在磁盘上，下次启动文件系统时再装入内存。

(4) 实现下列文件操作命令功能：**dir** 列文件目录、**open** 打开文件、**close** 关闭文件、**create** 创建文件、**delete** 删除文件、**read** 读文件、**write** 写文件。

1）dir 用户名，算法思路如下：

① 判断用户是否存在，若不存在显示错误信息并返回；

② 显示该用户文件目录的内容，包含文件名、文件长度，保护字段。

2）open 用户名/文件名，算法思路如下：

①判断用户是否存在，若不存在显示错误信息并返回；

②系统查找相应用户的用户文件目录，若没找到指定文件则显示文件不存在并返回；

③申请一个空闲的系统打开文件表项，若没有空闲表项则显示打开文件过多并返回；

④将文件目录复制到系统打开文件表项中并初始化文件读写指针，返回打开文件在系统打开文件表的表项下标（后面称为文件描述符）供读写操作使用，系统打开文件表设计为结构数组（也可以设计为其他形式，此时返回的是能找到打开文件属性信息数据结构的指针，也称为文件句柄）。

3）close 用户文件描述符，算法思路如下：

①判断用户文件描述符是否存在，若不存在显示错误信息并返回；

②将文件描述符所指系统打开文件表项释放，使其成为空闲表项。

4）create 用户名/文件名/保护字段，算法思路如下：

①判断用户是否存在，若不存在显示错误信息并返回；

②系统查找相应用户的用户文件目录，检查用户已有文件数是否小于 **10**，若有 **10** 个文件且新建文件名不存在于用户文件目录中则显示创建文件已达最大数目并返回；

③在用户文件目录表中查找指定文件，若找到指定文件则将该文件长度清零，申请一个空闲的系统打开文件表项，将文件目录复制到系统打开文件表项中并初始化文件读写指针，返回该文件的文件描述符。

④若没有找到指定文件，则申请一个空闲文件目录项，填写文件名及保护字段信息，初始化其他字段；再申请一个空闲的系统打开文件表项，将文件目录复制到系统打开文件表项中并初始化文件读写指针，返回文件描述符。

5）delete 用户名/文件名，算法思路如下：

①判断用户是否存在，若不存在显示错误信息并返回；

②系统查找相应用户的用户文件目录，检查文件是否存在，若文件不存在则显示错误信息并返回；

③在用户文件目录表中查找指定文件，释放文件占用的空间，释放目录项。

6）read 文件描述符/存放读数据的缓冲区/本次读字符数，算法思路如下：

①判断用户文件描述符是否存在，若不存在显示错误信息并返回；

②根据文件描述符找到相应的系统打开文件表项，判断该文件是否允许读操作，若不允许则显示错误信息并返回；

③按照本次读字符数从当前文件指针开始读数据到缓冲区中。

7）**write** 文件描述符/存放写数据的缓冲区/本次写字符数，算法思路如下：

①判断用户文件描述符是否存在，若不存在显示错误信息并返回；

②根据文件描述符找到相应的打开文件表项，判断该文件是否允许写操作，若不允许则显示错误信息并返回；

③按照本次写字符数从文件尾开始将缓冲区中的数据写到文件中，并修改目录中的文件大小

(5) 程序启动后，应先将磁盘上的两级目录结构及文件内容读入内存数据结构中并初始化系统打开文件表（可以事先确定文件系统的数据结构初值，包含主目录的 **8** 个用户信息及指向各自用户文件目录的指针，每用户可以有初始文件也可以没有，若有初始文件则还需为这些已存在文件赋值，这些初值以文件形式存放在磁盘上）。然后显示接收用户命令输入的界面，用户输入命令，系统执行命令并显示命令执行结果，还应显示与命令相关的目录及文件内容变化前后情况。

(6) 鼓励在实验中加入磁盘空间管理等功能，以使本文件系统功能更完善。比如磁盘初始化后统计可用空间，文件/目录创建时检查空间，读写时更新文件大小。

【验收标准】：

(1) 目录结构设计合理，支持两级子目录。

(2) 文件/目录创建/删除、打开/关闭、读/写等操作实现正确；

(3) 文件控制块和目录项结构设计合理，存储了所需的文件/目录信息；

(4) 有可视性好的用户界面，有良好的测试用例，测试结果符合预期；

操作系统内核实验

本实验要求同学们通过完善一个运行在 RISC-V 体系结构上的操作系统内核，来加深对操作系统原理的理解。整个实验在类 Linux 环境下进行，同学们首先要构建一个类 Linux 环境，通过直接安装 ubuntu 或麒麟操作系统或 openEuler 操作系统，或者在 windows10 环境下利用 WSL+MobaXterm 搭建一个类 Linux 环境。由于我们并没有 RISC-V 的硬件，所以要安装一个 QEMU 模拟器来模拟 RISC-V 硬件环境。目前大家的电脑基本上都是 x86 的 CPU，要设计能在模拟 RISC-V 硬件上运行的操作系统内核，就需要在 linux 中安装 RISC-V 交叉编译器，使得编译出来的应用程序可以在模拟 RISC-V 硬件上运行，以测试内核能否正常运行。

本次实验将在基于 QEMU 模拟的 RISCV 指令集上进行 Linux 内核实验，涉及的内核部分包括：系统调用、内存管理和进程调度。其中既包括阅读源码的任务，也有修改内核的任务。操作系统内核实验需要阅读大量的资料，这是一个有挑战性的实验任务，欢迎大家接受挑战!

一、实验环境搭建

（一）、什么是 QEMU 和 RISC-V?

QEMU 是一款免费开源的模拟器，可以模拟多种不同的处理器架构，包括 x86、ARM、MIPS 和 RISC-V 等。QEMU 不仅可以模拟处理器，还可以模拟各种外设、网络和存储设备，支持多种操作系统，包括 Linux、Windows 和 FreeBSD 等。QEMU 的主要特点包括：

- 可移植性：QEMU 可以运行在多种操作系统上，包括 Linux、Windows 和 Mac OS 等。
- 灵活性：QEMU 可以模拟多种不同的处理器架构和外设，可以根据需要进行灵活的配置和调整。
- 高效性：QEMU 使用动态二进制翻译技术，可以将模拟的指令快速转换成宿主机器指令，提高模拟器的效率和性能。

RISC-V 是一种新兴的开源指令集架构（ISA），由加州大学伯克利分校开发。RISC-V 的设计理念是“简单、精简、可扩展”，旨在提供一种简单、通用、高效的 ISA，以满足各种应用需求。RISC-V 的指令集是模块化的，只有一小部分必须的指令，其他的模块可以由厂商自由实现，这使得 RISC-V 适用于从小型嵌入式系统到大型超级计算机的各种场景。RISC-V 的主要特点包括：

- 开放性：RISC-V 是一种开放的 ISA，任何人都可以自由使用、修改和扩展。
- 易于实现：RISC-V 的指令集非常简单、规范，易于实现和调试。

- 可扩展性：RISC-V 的指令集可以根据需要进行扩展和定制，可以满足各种应用需求。
- 高效性：RISC-V 的指令集设计非常精简，可以提高处理器的效率和性能。

（二）、安装和配置 QEMU 和 RISC-V 的交叉编译工具链

1、什么是交叉编译?

了解交叉编译之前，先来了解一下本地编译。

（1）本地编译

本地编译可以理解为，在当前编译平台下，编译出来的程序只能放到当前平台下运行。平时我们常见的软件开发，都是属于本地编译。

比如，我们在 x86 平台上，编写程序并编译成可执行程序，就只能在 x86 平台下运行。这种方式下，我们使用 x86 平台上的工具，开发针对 x86 平台本身的可执行程序，这个编译过程称为本地编译。

（2）交叉编译

交叉编译可以理解为，在当前编译平台下，编译出来的程序能运行在体系结构不同的另一种目标平台上，但是编译平台本身却不能运行该程序。

比如，我们在 x86 平台上，编写程序并编译成能运行在 ARM 平台的程序，就算两个平台都运行的是 Linux 系统，但编译得到的程序在 x86 平台上还是不能运行的，必须放到 ARM 平台上才能运行。

（3）交叉编译链

编译过程包括了预处理、编译、汇编、链接等功能。这些不同的子功能都是通过一个单独的工具来实现的，它们合在一起就形成了一个完整的工具集。编译过程是一个有先后顺序的流程，它必然牵涉到工具的使用顺序，每个工具按照先后关系串联在一起，这就形成了一个链式结构。

因此，**交叉编译链**就是为了编译跨平台体系结构的程序代码而形成的由多个子工具构成的一套完整的工具集。同时，它隐藏了预处理、编译、汇编、链接等细节，当我们指定了源文件(.c)时，它会自动按照编译流程调用不同的子工具，自动生成最终的二进制程序映像(.bin)。

注意：严格意义上来说，交叉编译器，只是指交叉编译的 gcc，但是实际上为了方便，我们常说的交叉编译器就是交叉工具链。本文对这两个概念不加以区分，都是指编译链。

（4）交叉编译链的命名规则

我们使用交叉编译链时，常常会看到这样的名字：

arm-none-linux-gnueabi-gcc

arm-cortex_a8-linux-gnueabi-gcc

mips-malta-linux-gnu-gcc

这些交叉编译链的命名规则似乎是通用的，有一定的规则：

arch-core-kernel-system

- **arch**：用于哪个目标平台。

- **core**：使用的是哪个 CPU Core，如 Cortex A8，但是这一组命名好像比较灵活，在其它厂家提供的交叉编译链中，有以厂家名称命名的，也有以开发板命名的，或者直接是 none 或 cross 的。
- **kernel**：所运行的 OS，见过的有 Linux，uclinux，bare (无 OS) 。
- **system**：交叉编译链所选择的库函数和目标映像的规范，如 gnu，gnueabi 等。其中 gnu 等价于 glibc+oabi；gnueabi 等价于 glibc+eabi。

2、安装和配置 QEMU 模拟器

- **从源码安装 QEMU 模拟器**

（1）首先需要确认所使用的操作系统是否支持 QEMU 模拟器，以及是否安装了必要的依赖库。

要在 Ubuntu 中安装 QEMU，您需要安装通常需要安装以下依赖项：

- build-essential: 它是一个包，包括了用于编译软件的基本工具。
- pkg-config: 它是一个包，用于检查系统上是否安装了需要的依赖包。
- zlib1g-dev: 它是一个压缩库，QEMU 需要使用它来压缩和解压缩模拟器镜像。
- libglib2.0-dev: 它是 Glib 库的开发包，QEMU 使用 Glib 库来提供基础设施支持。

```
sudo apt-get install build-essential pkg-config zlib1g-dev libglib2.0-dev
```

不同的操作系统环境可能缺少的依赖项不同。比如在我安装 QEMU 的过程中，就缺少了 Ninja 和 pixman-1，Python 从 3.6 版本更新到了 3.7 以上版本。大家可以根据后序执行 configure 命令时的报错信息，再添加相应的依赖项。

$sudo apt-get install ninja-build

$ninja --version　　##判断 ninja 是否安装成功

$sudo apt-get install libpixman-1-dev

$pkg-config --modversion pixman-1　##判断 pixman-1 是否已安装成功，若成功则显示版本号

$sudo apt-get install python3.7

$sudo rm /usr/bin/python3

$sudo ln -s /usr/bin/python3.7 /usr/bin/python3 ##重新将 Python3.7 和 Python3 链接起来

pkg-config 是一个常用于 Unix 和 Linux 系统的命令行工具，可用于检查特定库的版本、安装路径、包含头文件和链接库等信息。要使用 pkg-config 检查是否安装了所需的库，可以在终端中使用以下命令：

$pkg-config --exists <library_name>

其中 <library_name> 是您要检查的库的名称 。如果库已安装并使用了正确的名称，则该命令将返回退出代码 0，否则将返回退出代码 1。比如：

$pkg-config --exists libcurl

如果库已安装，则会看到这样的输出：

$ pkg-config --exists libcurl

$ echo $?

0

（2）然后可以从 QEMU 官方网站（https://www.qemu.org/download/）下载最新版本的 QEMU 模拟器源代码。

（3）下载完成后，解压缩 QEMU 源代码，并进入源代码目录。

（4）或者通过 git 下载源码。

git clone https://gitlab.com/qemu-project/qemu.git

cd qemu

git submodule init

git submodule update --recursive

（5）执行以下命令进行编译和安装：

$ ./configure --target-list=riscv64-softmmu --prefix=/usr/local/qemu

$ make

$ sudo make install

注意：--target-list 参数指定需要编译的处理器架构，这里选择 riscv64-softmmu 表示编译 64 位 RISC-V 处理器模拟器。

（5）编译完成后，可以在/usr/local/qemu/bin 目录下找到 qemu-system-riscv64 可执行文件, 这是 RISC-V 模拟器的主要执行程序。

$cd /usr/local/qemu/bin

$./qemu-system-riscv64

VNC server running on 127.0.0.1:5900　##QEMU 虚拟机启动

- **安装预编译好的 QEMU 模拟器**

##ubuntu

$ apt-get install qemu

##MacOS

$ brew install qemu

$$64 bit Windows 7 or above (in MINGW64)

pacman -S mingw-w64-x86_64-qemu

注意：当操作系统版本比较低的时候可能无法安装预编译的 QEMU, 此时只能下载源码进行编译安装。

3、安装 RISC-V 的交叉编译工具链

RISC-V 交叉编译工具链中包含的组件和库主要有：

- GCC：GNU Compiler Collection，是一个包含编译器、链接器、汇编器等工具的集合，用于编译各种编程语言的代码。
- Binutils：二进制工具集，包括汇编器、链接器、反汇编器等工具，用于处理二进制文件。
- Glibc：GNU C Library，是一个 C 语言标准库的实现，提供了许多常用的函数和数据结构。
- Linux 内核头文件：包括一些系统调用的头文件和系统相关的定义。
- libgcc：GCC 内置的 C 运行库，提供了一些基本的函数和支持。

RISCV 的交叉编译工具链是一个包含多个组件的工具集，用于编译和调试 RISCV 的程序和系统。每个组件都有其特定的作用，通过这些组件的协同工作，可以构建出一个完整的 RISCV 开发环境。

安装 RISC-V 交叉编译工具链可以按以下步骤进行：

（1）首先需要从 RISC-V 交叉编译工具链官方网站（https://github.com/riscv/riscv-gnu-toolchain）下载最新版本的源代码。

```
$git clone https://github.com/riscv/riscv-gnu-toolchain
```

（2）下载完成后，进入源代码目录。

```
$cd riscv-gnu-toolchain
```

（3）安装 Ubuntu 的标准依赖包。

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build
```

（4）执行以下命令进行编译和安装：

$ ./configure --prefix=/usr/local/riscv --with-arch=rv64imafdc --with-abi=lp64d

$ make linux

$ sudo make install

注意：--with-arch 和--with-abi 参数指定需要编译的处理器架构和 ABI，这里选择 rv64imafdc 表示编译 64 位 RISC-V 处理器，lp64d 表示使用 LP64 数据模型和双精度浮点数。

（4）编译完成后，可以在/usr/local/riscv/bin 目录下找到 RISC-V 交叉编译工具链的各个可执行文件，例如 riscv64-unknown-elf-gcc、riscv64-unknown-elf-gdb 等。

其它操作系统的安装方法参考官网的 README。

也可以直接安装预编译好的交叉编译工具链。只有 Ubuntu 操作系统有预编译版本的交叉编译工具链，其它的操作系统需要从源码开始编译安装。

（1）首先需要从 RISC-V 交叉编译工具链官方网站（https://github.com/riscv-collab/riscv-gnu-toolchain/releases）打开 Assets，找到适合你的 Ubuntu 操作系统版本的预编译压缩文件，比如 riscv64-glibc-ubuntu-18.04-nightly-2022.11.12-nightly.tar.gz。

- ELF (Executable and Linkable Format) 是一种二进制文件格式，用于描述可执行文件、目标文件和共享对象等。ELF 文件格式包含用于描述程序和数据的头部、程序代码和数据段等。

- glibc: glibc (GNU C Library) 是一个 C 语言库, 用于在 Linux 和其他 Unix 型操作系统中提供标准的系统和库函数。glibc 具有丰富的 API，包含众多标准 C 库函数，例如字符串处理、文件 I/O、进程控制、内存管理、网络通信等。
- musl: musl 是一个轻量级的 C 语言库，其设计目标是提供相对较小而快速的标准 POSIX API, 并且尽可能地遵循正式的 POSIX 标准。与 glibc 相比，musl C 库不包含可执行文件、动态链接器和其他操作系统附带的大容量库或共享对象。

如何要使用 RISCV GNU toolchain 编译 Linux 操作系统内核, 则需要选择 glibc。

（2）解压预编译压缩文件到指定目录~/riscv。

```
sudo tar xf
~/Downloads/riscv64-glibc-ubuntu-18.04-nightly-2022.11.12-nightly.tar.gz
~/riscv
```

（3）添加工具链到 PATH 环境变量中，可以在~/.bashrc 文件中添加以下行:

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

（4）验证是否安装成功。

```
$riscv64-unknown-linux-gnu-gcc --version
$riscv64-unknown-linux-gnu-ld --version
$riscv64-unknown-linux-gnu-gdb --version
```

如果出现 gcc、ld、gdb 的版本信息, 则表明成功安装了 RISCV GNU 的工具链。

安装完成后, 就可以使用各种 RISC-V GNU Toolchain 工具来开发和调试 RISC-V 程序了。

（三）、下载和运行 RISC-V 的 Linux 内核和文件系统

1、下载 Linux 内核源代码：从 Linux 官方网站(https://www.kernel.org/)上下载 RISC-V 架构的 Linux 内核源代码, 可以选择最新版本或者稳定版本。将下载的源代码解压到本地文件夹，例如/home/user/linux。也可以通过 git clone 命令下载。不管最终运行的处理器架构是什么，都是从同一源码仓库中下载的。然后可以使用特定的交叉编译工具链来编译这些源代码，以生成适合特定处理器架构的可执行文件、内核映像等。

```
$git clone https://github.com/torvalds/linux.git
```

2、编译 Linux 内核：使用 RISC-V 工具链中的交叉编译器，将 Linux 内核源代码编译成 RISC-V 架构的可执行代码。具体步骤如下：

（1）配置内核：在终端中进入 Linux 内核源代码目录，执行以下命令, 采用 RISCV 的默认设置来配置内核，这会生成.config 默认配置文件。

```
$cd linux
$make ARCH=riscv defconfig
```

如果希望修改配置，则可以通过 menuconfig 进行交互式调整。

```
$make ARCH=riscv menuconfig
```

（2）编译内核：在终端中执行以下命令，选择 RISCV GNU 交叉编译工具链编译内核。

$make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu-

-j$(nproc)

`-j$(nproc)`选项指定编译时使用的线程数。`$(nproc)`是一个变量，它代表系统上的 CPU 核心数。这条命令将最大限度地利用系统上的所有 CPU 核心来加速内核编译。如果不指定`-j$(nproc)`，则默认只使用 1 个线程，不会全部利用 CPU 资源，编译会慢很多。通过在命令行直接输入`nproc`命令可以查看你的系统上的 CPU 核心数。

在编译内核时，可能会出现一些选择，比如：

Enable register zeroing on function exit(ZERO CALL USED

REGS)(N/y/?)

这个选项是启用或禁用函数退出时的寄存器清零。当一个函数退出时，它的寄存器通常包含敏感数据，如果这些寄存器的内容不清零，就可能泄漏内核数据。因此，启用它可以提高内核的安全性。以下是选项的解释：

- 如果选择 "y"，则将启用该选项并清零寄存器。这将增加一些开销，但会提高内核的安全性。
- 如果选择 "n"，则将禁用该选项，并且内核将不会在函数退出时清零寄存器。
- 如果选择 "?"，则将显示选项帮助信息。

建议启用该选项以提高内核的安全性，特别是在处理内核数据时。但是，它会增加一些开销，因此在性能敏感的场景下，可能需要权衡考虑。

如果在编译过程中遇到缺少依赖包的错误，那么需要安装这些依赖包后再重新编译。如果遇到选项，可以根据实际情况选择，或者回车接受其默认选择。

整个编译过程耗时较长，需要耐心等待。

3、编译结束后，将在 arch/riscv/boot 目录下生成 Image、Image.gz 和 dts 文件。

- arch/riscv/boot/Image：这是 RISC-V Linux 内核的 ELF 格式镜像文件，包含内核代码和数据。
- arch/riscv/boot/Image.gz：Image 文件的 gzip 压缩版本，用于引导时解压使用。
- arch/riscv/boot/dts/：内核所需要的 RISC-V 平台设备树文件，描述系统硬件信息。
- vmlinux：未压缩的 Linux 内核 ELF 文件，包含调试符号，用于内核调试。

Image 和 Image.gz 包含了 Linux 内核的机器代码, 当使用引导加载程序(如 U-Boot)加载后，就可以在目标板上启动 Linux 系统。如果要在 QEMU 模拟的 RISCV 机器上运行，通常直接使用 Image 或 Image.gz 文件作为-kernel 参数进行引导即可。这样可以直接运行并进入 Linux 内核，开始执行初始化进程和系统引导。

4、使用 GDB 在 QEMU 中加载 vmlinux 进行源码级调试。

-在 QEMU 中加载 vmlinux，使用 loader 设备来加载 vmlinux 到 0x80200000 地址并执行，达到启动内核的目的，内核引导后进入 gdbstub 调试接口：

```
$qemu-system-riscv64 -machine none -nographic  \
-device loader,file=vmlinux,addr=0x80200000
```

-然后在另外一个终端上启动 GDB，并连接到 gdbstub 调试 QEMU：

```
$riscv64-linux-gnu-gdb
(gdb)
```

5、如果希望直接引导 linux 内核，则可以在步骤 3 的基础上，在 QEMU 中使用 Image 内核镜像进行引导：

```
$qemu-system-riscv64 -machine virt -nographic -kernel Image
```

从而直接进入 Linux 内核启动过程。通过打印 kmsg 信息，可以观察到系统调用、内存管理和调度等行为。

6、用 qemu 启动 riscv 虚拟机的启动参数很多，为了方便管理和复用启动参数，可以设置一个配置文件，将一些固定参数设置在其中。比如，设置 qemu_riscv64.conf 如下：

```
[ riscv64 ]
# 选择 Raspbian 作为 Guest OS
kernel = "/path/to/Image"
initrd = "/path/to/initrd.img"
```

```
# QEMU 的参数选择
machine = "virt"          # 选择 virtIO 模拟的板卡
cpu = "rv64,x=true"     # 64 位 RISC-V CPU 类型
bios = ""                  # 不使用 BIOS
boot-cpus = "2"           # 使用 2 个虚拟 CPU
mem-size = 512M           # 分配 512MB 内存

# 网络配置
netdev = "eth0,nic"     # 使用 virtIO 网卡
macaddr = "52:54:00:fa:9f:02"

# 存储设备
drive0 = "root=/dev/vda rw"   # 通过 vda 来引导系统，用于指定操作系统启动和根文件系统所在的存储设备
drive1 = "/dev/sda,file=/home/user/raspbian.img"   # 镜像文件，指定虚拟硬盘镜像文件的路径
```

在 QEMU 启动时,可以通过 -readconfig 选项来加载配置文件中设置的参数。

```
qemu-system-riscv64 -readconfig qemu_riscv64.conf
```

也可以在载入配置文件里设置的参数后，再手动加入其它参数，比如：

```
qemu-system-riscv64 -readconfig qemu_riscv64.conf -m 512M -smp 2
-nographic
```

可以将所有 QEMU 虚拟机的配置文件放在一个统一的目录下，比如创建一个配置文件目录/usr/local/qemu/configs/，然后每个虚拟机配置一个文件，且以虚拟机名称命名，比如 vm1.conf，启动时则直接使用相对路径加载即可。

```
QEMU_CONFIG_DIR=/usr/local/qemu/configs
qemu-system-riscv64 -readconfig $QEMU_CONFIG_DIR/vm1.conf
```

3、创建一个虚拟磁盘映像，并将其用作 QEMU 的虚拟硬件设备。您可以使用以下命令创建一个大小为 2GB 的磁盘映像：

```
$qemu-img create -f raw mydisk.img 2G
```

4、在 QEMU 中启动 RISC-V 体系结构的 Linux 内核。以下是启动 QEMU 的命令：

```
$qemu-system-riscv64 \
-machine virt \
-kernel path/to/vmlinux \
-append "console=ttyS0" \
-monitor stdio
```

在这个命令中， -machine virt 表示使用 virtio 平台模拟器，-nographic 表示不使用图形化界面，输出通过终端调试， -kernel 表示指定要加载的 Linux 内核文件，将/path/to/vmlinux 替换为 Linux 内核映像的实际路径， -append 后面的参数表示将命令行参数传递给内核， console=ttyS0 表示使用 virtio 串口设备作为控制台输出。

（四）、在 QEMU 上调试 RISC-V 的 Linux 内核

1、启动 QEMU：在终端中执行以下命令，启动 QEMU。

```
qemu-system-riscv64 -machine virt -cpu rv64 -nographic -smp 4 -m 2G
-kernel linux/arch/riscv/boot/Image -append "root=/dev/vda console=ttyS0"
-drive file=output/images/rootfs.ext2,format=raw,id=hd0 -device
virtio-blk-device,drive=hd0 -s -S
```

该命令会启动 QEMU，并将 Linux 内核和文件系统镜像加载到虚拟机中。其中，-s 和-S 参数用于启用 GDB 调试。

2、使用 GDB 调试：在另一个终端中，执行以下命令，启动 GDB。

```
riscv64-unknown-linux-gnu-gdb vmlinux
```

然后，使用以下命令连接到 QEMU。

```
target remote localhost:1234
```

接着，可以使用 GDB 调试命令进行调试，例如设置断点、单步执行等。

参考文献

[1]: https://dingfen.github.io/risc-v/2020/07/23/RISC-V_on_QEMU.html "QEMU 上运行 RISC-V Linux 内核 - 峰子的乐园"

[2]: https://blog.csdn.net/df12138/article/details/120441829 "QEMU 上运行 RISC-V Linux 内核_riscv stubs-lp64.h_df12138 的博客-CSDN 博客"

[3]: https://zhuanlan.zhihu.com/p/383724936 "在嵌入式 Linux 系统中使用 QEMU 运行 RISC-V 的入门指南 - 知乎"

[4]: https://blog.csdn.net/blaction/article/details/109372582 "如何在 Linux 下使用 QEMU 运行一个 RISC-V 架构的 Linux (busybox 篇) _risc-v qemu 怎么运行单个程序 ..."

[5]: https://zhuanlan.zhihu.com/p/258394849 "在 QEMU 上运行 RISC-V 64 位版本的 Linux - 知乎"

[6]: https://wiki.qemu.org/Documentation/Platforms/RISCV "Documentation/Platforms/RISCV - QEMU"

二、系统调用实验

（一）、系统调用的概念和作用

系统调用是操作系统内核提供给应用程序的接口，应用程序通过接口来获得系统资源和操作系统内核提供的服务。它可以被应用程序用来请求操作系统执行某些特权操作，比如文件读写、网络通信、进程管理等。系统调用可以提供给应用程序更高级别的抽象，使得应用程序可以更方便地使用底层系统资源和服务，同时也保证了系统的安全性和稳定性。

（二）、RISC-V 的系统调用约定和机制

RISC-V 的系统调用约定和机制如下：

1、系统调用号：系统调用号通过寄存器 a7 传递给内核，内核根据不同的系统调用号执行不同的操作。

2、参数传递：系统调用的参数通过寄存器 a0-a6 传递给内核，如果参数较多，则可以使用栈来传递。

3、返回值：系统调用的返回值通过寄存器 a0 返回给应用程序，通常返回值为 0 表示成功，非 0 表示失败。

4、系统调用指令：RISC-V 的系统调用指令为 ecall，执行 ecall 指令会触发系统调用。

5、特权级别：执行系统调用需要进入特权级别，即从用户态切换到内核态，RISC-V 采用 M 态（机器态）和 U 态（用户态）两个特权级别。

6、系统调用处理：在 M 态下，内核会根据系统调用号和参数执行相应的操作，并将返回值写入寄存器 a0 中，然后再切换回 U 态，让应用程序继续执行。

RISC-V 的系统调用过程大致如下：

1、用户程序将系统调用号存入 a7 寄存器，将系统调用参数存入 a0-a6 寄存器。

2、用户程序执行 ecall 指令，触发异常，进入机器模式。

3、机器模式的异常处理程序根据 a7 寄存器的值，分发系统调用请求到相应的服务例程。

4、服务例程执行系统调用的功能，并将返回值存入 a0-a1 寄存器。

5、服务例程执行 mret 指令，返回用户模式，继续执行用户程序。

（三）、理解 Linux 中的系统调用

要充分理解一个架构(比如 RISC-V 或 x86)的系统调用，需要参考以下几个文件：

1. include/uapi/asm-generic/unistd.h： 该文件定义了通用的系统调用宏和编号，这些系统调用在所有架构上都是相同的，还定义了系统调用的总数，比如

```
/* fs/read_write.c */          //指出系统调用处理函数所在的文件
#define __NR3264_lseek 62
__SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
#define __NR_read 63     // 定义系统调用号
__SYSCALL(__NR_read, sys_read) //系统调用号与系统调用函数关联起来
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)

#undef __NR_syscalls
#define __NR_syscalls 451       //通用系统调用的总数
```

2. arch/xxx/include/uapi/asm/unistd.h： 该文件定义了架构 xxx 特有的系统调用宏和编号。
3. arch/xxx/include/asm/syscall.h：该文件定义了系统调用辅助函数，以方便内核在处理系统调用时获取相关信息或操作系统调用流程。比如：

syscall_get_nr: 获取当前正在执行的系统调用号。这在系统调用返回后非常有用,可以用于日志或调试。

syscall_rollback: 回滚当前系统调用。在某些情况下,内核需要在系统调用处理的中间阶段就返回到用户空间,此时可以使用 syscall_rollback() 函数。

syscall_get_error: 获取当前系统调用返回的错误号。在系统调用处理函数中,内核通过 retval 来返回错误号或成功返回值,此函数可以提取其中的错误号。

syscall_get_return_value: 获取系统调用的返回值。与 syscall_get_error 配合使用,可以获取 syscall 返回的正确返回值或错误号。

4. arch/xxx/kernel/ syscall_table.c：该文件完成了 sys_call_table 数组的定义和初始化，将每个系统调用号映射到其处理函数。比如将 __NR_read 映射到 sys_read，将__NR_write 映射到 sys_write。

```
#undef __SYSCALL
#define __SYSCALL(nr, call) [nr] = (call),

void * const sys_call_table[__NR_syscalls] = {
    [0 ... __NR_syscalls - 1] = sys_ni_syscall,   // 未使用的系统调用号都被映射到 sys_ni_syscall
#include <asm/unistd.h>
};
```

5. fs/read_write.c(或其他 C 文件): 该文件中定义了与文件读写相关的系统调用处理函数，包括 read 和 write **系统调用的处理函数**。

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t,
count)
{// read 为系统调用的名称，接下来的 6 个参数分别是三组参数的数据类型和值
    return ksys_read(fd, buf, count);   //实现 read 的系统调用
}

SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    return ksys_write(fd, buf, count); //实现 write 的系统调用
}
```

SYSCALL_DEFINE3(name, arg1, arg2, arg3)：这个宏用来定义带 3 个参数的系统调用处理函数。比如 read 和 write 系统调用就有 3 个参数。

所以，要充分理解一个架构的系统调用，需要同时参考：

1) include/uapi/asm-generic/unistd.h 通用架构的系统调用定义；

2) arch/xxx/include/uapi/asm/unistd.h 特定架构的系统调用定义；

3) arch/xxx/kernel/ syscall_table.c 完成系统调用号与系统调用处理函数的映射;

4) **.c 定义系统调用处理函数的具体实现;

这四个文件共同定义了该架构的系统调用框架。

（四）、如何在 Linux 内核中添加和实现自定义的系统调用

假设要在内核源代码中添加一个名叫 sys_riscv_hello 的系统调用, 它打印一条消息到内核日志。在 Linux 内核中添加和实现该系统调用可以按照以下步骤进行:

1、首先定义系统调用号，并将系统调用号与系统调用处理函数关联起来。

```
//   arch/riscv/include/uapi/asm/unistd.h
#include <asm-generic/unistd.h>
#define __NR_hello (__NR_arch_specific_syscall + 1)
__SYSCALL(__NR_hello, sys_riscv_hello)
```

注意：要确保定义的系统调用号是没有用过的。

2、在 arch/riscv/kernel/目录下添加一个.c 文件，比如 sys_riscv_my.c, 里面包含 riscv_hello 系统调用的实现。

```
#include <linux/kernel.h>
#include <asm/unistd.h>
SYSCALL_DEFINE0(riscv_hello)
```

```
{
    printk(KERN_INFO "Hello world!\n");
    return 0;
}
```

3、重新编译内核，并将编译后的内核文件安装到系统中。

4、在应用程序中调用系统调用，例如使用 C 语言的'syscall'函数，其中第一个参数为系统调用号，后面的参数为系统调用的参数。

```
syscall(sys_riscv_hello);
```

（五）、系统调用实验要求

请自定义一个系统调用，获取当前的时间，并将该时间以北京时间输出到内核日志中。

- 首先在内核的系统调用表中注册一个新的系统调用，包括系统调用的接口和号码；
- 然后在内核源码中实现该自定义的系统调用函数，可以获取当前时间并将时间输出到内核日志；
- 接着编写一个用户程序，使用 syscall()函数来调用自定义的系统调用，传入系统调用号及其它参数；
- 最后，使用 RISC-V 工具链编译程序，生成可执行文件，将可执行文件复制到 QEMU 模拟器中运行，观察系统调用的效果。

在内核日志中应该可以看到当前时间。请注意，自定义系统调用的系统调用号需要与内核中注册的号码保持一致。

参考文献

[1]: https://zhuanlan.zhihu.com/p/259305354 "搭建 RISC-V 编译环境与运行环境 - 知乎"

[2]: https://zhuanlan.zhihu.com/p/487648323 "给 Linux 内核添加自己定义的系统调用 - 知乎"

[3]: https://blog.csdn.net/qq_34258344/article/details/103228607 "Linux 系统调用（二）——使用内核模块添加系统调用（无需编译内核）_JinrongLiang 的博客-CSDN 博客"

[4]: https://blog.csdn.net/Xiaobai__Lee/article/details/101979481 "如何给 Linux kernel 添加一个系统调用_Xiaobai__Lee 的博客-CSDN 博客"

[5]: https://blog.csdn.net/weixin_43641850/article/details/104906726 "linux 内核编译及添加系统调用(详细版)_LitStronger 的博客-CSDN 博客"

[6]: https://blog.csdn.net/kwame211/article/details/77572123 "Linux 系统调用函数列表_DemonHunter211 的博客-CSDN 博客"

[7]: https://blog.csdn.net/dillanzhou/article/details/82733562 "linux 系统调用原理及实现_dillanzhou 的博客-CSDN 博客"

[8]: https://blog.csdn.net/Smart_yujin/article/details/10515247 "Linux 下的 exec 系统调用详解_linux 上 exec dde_yujin735 的博客-CSDN 博客"

三、内存管理实验

（一）、内存管理的目标和内容

操作系统内存管理的目的是使得计算机能够高效地利用内存资源，同时确保程序不会越界访问内存、不会相互干扰、不会因为内存不足而崩溃。

内存管理的主要内容包括：

1、内存分配：为程序分配所需的内存空间，确保每个程序都能够获得足够的内存空间。

2、内存保护：对内存空间进行保护，防止程序越界访问内存、修改其他程序的内存空间。

3、内存回收：当程序不再需要内存空间时，及时回收内存资源，避免内存泄漏。

4、内存共享：允许多个程序共享同一块内存空间，提高内存利用率。

5、内存交换：将部分不活跃的程序或数据从内存交换到硬盘中，以释放内存资源，从而为其他程序提供更多的内存空间。

（二）、RISC-V 的虚拟地址空间和页表结构

RISC-V 是一种开源的指令集架构，支持多种虚拟地址空间和页表结构。

RISC-V 的虚拟地址空间是指程序使用的逻辑地址空间，它可以被分为用户空间和内核空间。为了将虚拟地址空间映射到物理地址空间，RISC-V 的页表是关键。页表可以是单级的，也可以有多级，每级页表都有一个基址寄存器。

最新的 RISC-V 的 64 位 Linux 内核支持以下几种页表结构：Sv48、Sv39 和 Sv57。

- Sv48 是支持 48 位虚地址的页表结构，它将 64 位的虚拟地址分为 9 位的页目录索引、9 位的二级页表索引、9 位的三级页表索引、9 位的四级页表索引和 12 位的页内偏移量，它支持 256TB 的虚拟地址空间和 256TB 的物理地址空间。
- Sv39 是支持 39 位虚地址的页表结构，它将 64 位的虚拟地址分为 9 位的页目录索引、9 位的二级页表索引、9 位的三级页表索引和 12 位的页内偏移量，它支持 512GB 的虚拟地址空间和 512GB 的物理地址空间。该页表可以在编译内核时选择。
- Sv57 是支持 57 位虚地址的页表结构，是 Linux 5.18 中新增加的虚拟地址模 v 式。64 位虚地址的最高位为符号位，用于区分用户空间和内核空间。剩下的 57 位虚地址分为 5 个部分，每个部分有 9 位，分别对应 5 级页表的索引。每个页表项有 64 位，其中包含了物理页号、权限位、全局位、访问位和脏位等信息。一个 Sv57 页表结构可以映射2^{57}字节的虚拟地址空间到2^{48}字节的物理地址空间。它可以用来支持未来多达2^{57}字节，即 128PB(Petabyte)的物理内存容量。

最新的 RISC-V 的 64 位 Linux 内核仍然支持 Sv39 页表结构，但是默认的虚拟地址模式是 Sv48。如果你想使用 Sv39 页表结构，则需要在编译内核时修改配置文件，将 CONFIG_RISCV_VA_BITS_64 选项设置为 3。Sv39 页表结构使用 3 级页表，支持 39 位虚拟地址，用户空间和

内核空间的虚拟地址范围分别是 0x0000000000000000 - 0x0000007fffffffff 和 0xffffff8000000000 - 0xffffffffffffffff。

在操作系统中，MMU:内存管理单元(memory managemeny unit, MMU)负责虚拟地址到物理地址的转换。RISC-V 通过修改 Satp 寄存器的值来决定是否开启 MMU 功能。

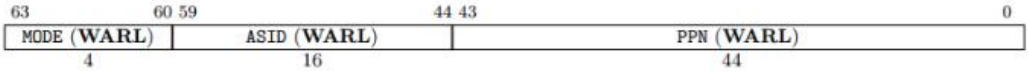


图 3.1 Satp 寄存器的结构

- PPN：保存根页表的物理页号；
- 地址空间标识符 (ASID)，用于唯一标识每个进程的地址空间，当进程执行时，它的 ASID 会与 Satp 寄存器中的 ASID 进行比较，以确保它只能访问其自己的地址空间。这种机制有助于保护进程之间的内存隔离，并提高系统的安全性。
- MODE 字段，表示当前的地址转换方案。当 MODE == 0 时，虚拟地址和物理地址是相等的, 同时 Satp 寄存器其它位域不起作用。当 MODE == 9 时，表示采用的是 Sv48 地址转换方案。

下面详细介绍 Sv48 页表结构及虚拟地址到物理地址的转换过程。其中 VPN 表示虚拟页号，PPN 表示物理页号。每个页号占 9 位，页内偏移占 12 位。Sv48 的虚拟地址结构如图 3.1 所示。虚拟地址的 63-48 位必须全部等于第 47 位，否则将发生页面错误异常。

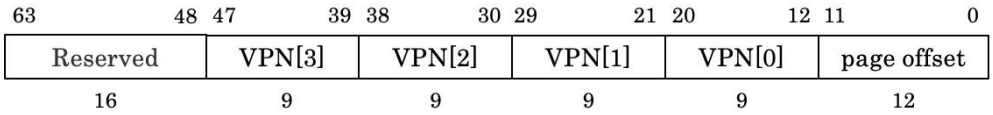


图 3.1 Sv48 的虚拟地址

当采用 Sv48 页表结构时, RISCV 支持的物理地址的长度为 44 位。Sv48 的虚拟地址的 36 位 VPN 通过 4 级页表转换为 44 位的 PPN，而 12 位的页偏移不进行转换。因此，物理地址的长度为 44 位+12 位=56 位，但是 RISC-V 规范要求物理地址的长度不能超过虚拟地址的长度，所以只有低 44 位的物理地址是有效的。

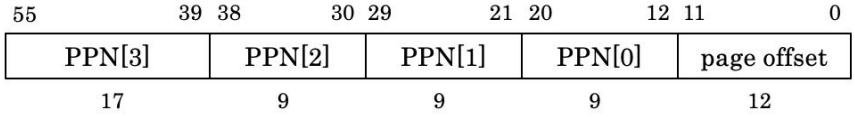


图 3.2 Sv48 的物理地址

Sv48 的页表项 PTE 结构如图 3.3 所示。

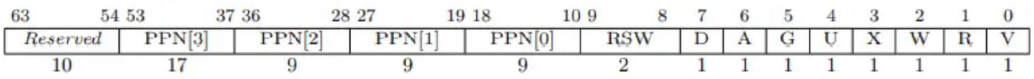


图 3.3 Sv48 的页表结构

- PTE [9:0] 表示控制位。
  - PTE[0]是 Valid 位指示该 PTE 是否有效；
  - PTE[1]:Readable，PTE[2]:Writeable，PTE[3]:eXecutable 分别指示了该页是否可读、可写、可运行，当这三位都是 0 时，表明该 PTE 指向了下一层页表，其为非叶子 PTE ，否则就是叶子 PTE；
  - PTE[4]:User 位指示了该页是否可以被用户模式访问；
  - PTE[5]:Global 指示了全局映射，存在于所有的地址空间中；
  - PTE[6]:Access 位指示了该页最近是否被读、写、取；
  - PTE[7]:Dirty 位指示了虚拟页最近是否被写过。

- 对于非叶 PTE，D，A，U 位被保留，并被清零。RSW 是保留的，用于操作系统软件。

从虚拟地址到物理地址的转换过程如下：

- 首先，检查虚拟地址的 63-48 位是否都等于 47 位，如果不是，则发生页面错误异常。
- 然后，从 satp 寄存器中获取根页表的物理基址（PPN），并将其左移 12 位，得到根页表的物理地址。
- 接着，从虚拟地址中提取 VPN2（47-39 位），并将其左移 3 位（相当于*8），得到第一级页表项的索引。将根页表的物理地址与第一级页表项的索引相加，得到第一级页表项的物理地址。
- 然后，从第一级页表项中读取 PPN（53-10 位），并将其左移 12 位，得到第二级页表的物理地址。
- 接着，从虚拟地址中提取 VPN[0]（29-21 位），并将其左移 3 位（相当于*8），得到第三级页表项的索引。将第三级页表的物理地址与第三级页表项的索引相加，得到第三级页表项的物理地址。
- 然后，从第三级页表项中读取 PPN（53-10 位），并将其左移 12 位，得到第四级页表的物理地址。
- 接着，从虚拟地址中提取 VPN[0]（20-12 位），并将其左移 3 位（相当于*8），得到第四级页表项的索引。将第四级页表的物理地址与第四级页表项的索引相加，得到第四级页表项的物理地址。

- 然后，从第四级页表项中读取 PPN（53-10 位），并将其左移 12 位，得到第五级页表的物理地址。
- 接着，从虚拟地址中提取 VPN[0]（11-0 位），并将其左移 3 位（相当于*8），得到第五级页表项的索引。将第五级页表的物理地址与第五级页表项的索引相加，得到第五级页表项的物理地址。
- 最后，从第五级页表项中读取 PPN（53-10 位），并将其左移 12 位，得到物理页面的基址。将物理页面的基址与虚拟地址中的偏移量（11-0 位）相加，得到物理地址。

（三）、内存管理实验要求

本实验要求大家阅读 riscv 架构下的 linux 源码，理解 linux 的分页内存管理机制，将默认的 LRU 页面置换算法替换为 FIFO 的页面置换算法，编写测试用例来观察页面的置换情况，从而更深入地理解分页内存管理的原理。本实验的要求和主要步骤如下：

1、查找与分页相关的数据结构，包括页表和页帧的定义，分析页表项（PTE）的结构和属性；

2、查看内核是如何初始化分页的；

- 找出分配页目录页表和初始化 PTE 的函数
- 分析其工作原理

3、寻找将虚地址转换为物理地址的函数；

4、查找内存分配与回收的函数，分析缺页中断处理过程；

5、将默认的 LRU 页面置换算法替换为 FIFO 页面置换算法，观察页面的置换情况。

6、编译内核并安装新内核，使修改生效。

参考文献

[1]: https://blog.csdn.net/dongze2/article/details/124548612 "RISCV 四级页表_sv39 sv48_繁华渲染悲凉的博客-CSDN 博客"

[2]: https://joessem.com/archives/riscv-mmu.html "Riscv 虚拟内存 Sv48 - 四十九 C-137"

[3]: https://www.jianshu.com/p/e58d20876615 "Riscv 虚拟内存 Sv48 - 简书"

[4]: https://zhuanlan.zhihu.com/p/78444537 "RISC-V Supervisor mode 地址翻译 (Sv48) - 知乎 - 知乎专栏"

[5]: https://docs.kernel.org/translations/zh_CN/riscv/vm-layout.html "RISC-V Linux 上的虚拟内存布局 — The Linux Kernel documentation"

[6]: https://marz.utk.edu/my-courses/cosc562/mmu/ "(MMU) SV39 Memory Management Unit – Stephen Marz"

[7]: https://unix.stackexchange.com/questions/281974/what-page-replacement-

algorithms-are-used-in-linux-kernel-for-os-file-cache "What page replacement algorithms are used in Linux kernel for OS file cache?"

[8]: https://www.kernel.org/doc/gorman/html/understand/understand013.html "Page Frame Reclamation"

[9] RISC-V Linux 内核 Paging & MMU 技术详解 https://www.bilibili.com/video/BV1Qi4y1r7av/?share_source=copy_web&vd_source=f43413ed69702715f11a3bb708e20bdf

[10] Linux mem 2.2 内核地址空间布局详解 https://blog.csdn.net/pwl999/article/details/112055498

四、处理器调度实验

（一）、处理器调度的概念和设计目标

处理器调度是操作系统的一个核心组件，它实现了对 CPU 资源的分配和调度，以使得多个进程或线程能够共享 CPU 资源，从而提高系统的性能和响应能力。处理器调度的设计目标是使系统的整体效率最大化，同时确保公平性和响应性。其设计目标主要包括以下几个方面：

1. 公平性：处理器调度的公平性是指所有的进程或线程都应该有机会获得 CPU 时间片，避免某些进程或线程长时间占用 CPU 导致其他进程或线程无法运行。为了实现公平性，操作系统采用了多种调度策略，

例如时间片轮转和多级反馈队列。在这些策略中，操作系统会将 CPU 时间分配给处于就绪状态的进程或线程，以便它们能够执行其任务。

2. 响应性：响应性是指系统应该能够快速响应用户的请求，将 CPU 分配给用户正在使用的应用程序，从而提高用户体验。为了实现响应性，操作系统采用了优先级调度和最高响应比优先调度等策略。这些策略优先分配 CPU 资源给高优先级的进程或线程，以确保系统能够快速响应用户的请求。

3. 吞吐量：吞吐量是指系统应该尽可能地处理更多的进程或线程，提高系统的整体效率。为了实现高吞吐量，操作系统采用了短作业优先和最短剩余时间优先调度等策略。这些策略优先分配 CPU 资源给执行时间短的进程或线程，以便系统在一段时间内能够处理更多的任务。

4. 资源利用率：资源利用率是指系统应该尽可能地利用 CPU 资源，避免 CPU 空闲浪费。为了提升资源利用率，操作系统采用了抢占式调度和非抢占式调度等策略。这些策略会在进程或线程需要等待 I/O 或其它操作时，将 CPU 资源分配给其他可执行的进程或线程，以最大化 CPU 的利用率。

在实际的应用场景中，操作系统通常采用多种调度策略的组合来实现处理器调度。例如，一个系统可能会采用时间片轮转策略来保证公平性和响应性，采用优先级调度策略来提高系统的吞吐量，以及采用抢占式调度策略来最大化 CPU 资源的利用率。这些策略的选择取决于系统的实际需求和特点，以及硬件资源的限制等因素。

为了保证操作系统的高效率、高响应和公平性，现代操作系统采用了多种处理器调度算法，例如：

1. 先来先服务（First-Come, First-Served, FCFS）调度算法：这是一种最简单的调度算法，按照进程提交的顺序进行调度，先提交的进程先获得 CPU 资源。这种算法简单易实现，但可能会导致长作业效应，即较长的进程占用 CPU 资源时间过长，导致其他进程等待时间和响应时间过长。

2. 最短作业优先（Shortest Job First, SJF）调度算法：按照进程需要的 CPU 时间长度进行调度，需要 CPU 时间最短的进程先获得 CPU 资源。这种算法可以最大限度地减少进程等待时间，但需要预测进程需要的 CPU 时间长度，实现较为困难。

3. 优先级调度算法（Priority Scheduling）：为每个进程分配一个优先级，按照优先级进行调度，优先级高的进程先获得 CPU 资源。这种算法可以根据进程的重要性和紧急性进行调度，但可能会导致低优先级进程长时间等待 CPU 资源。

4. 时间片轮转（Round-Robin, RR）调度算法：将 CPU 时间分为多个时间片，每个进程按照时间片轮流获得 CPU 资源，如果时间片用完则放弃 CPU 资源，重新排队等待下一轮调度。这种算法可以公平地分配 CPU 资源，同时能够保证进程响应时间和吞吐量。

5.多级反馈队列调度算法（Multilevel Feedback Queue）：该算法将进程按照优先级划分为多个队列，每个队列分配一个不同的时间片大小，每个队列的优先级都比前一个队列低。当进程被分配到一个队列时，

如果它的执行时间超过了该队列分配的时间片，则被移到下一个队列，直到执行完毕为止。这种算法可以兼顾进程的执行时间和优先级，同时保证所有进程都能够获得公平的 CPU 时间。

除了以上算法，现代操作系统中还有许多其他的处理器调度算法，比如公平分享调度算法。根据不同的应用场景和需求，选择不同的调度算法可以最大程度地提高系统的性能和利用率。

（二）进程调度实验要求

本实验要求大家阅读 riscv 架构下的 linux 源码，理解 linux 的进程调度机制，设计一个非抢占式的 FCFS 的调度器，编写测试用例来观察进程的调度顺序，从而更深入地理解进程调度的原理。本实验的要求和主要步骤如下：

1、查找 linux 内核源码中关于进程状态的定义；

2、找到就绪队列和等待队列的实现代码，分析它们在进程调度中的作用；

3、找到进行上下文切换 context switch 的过程，理解上下文切换完成了哪些任务。修改上下文切换的代码，使得可以打印进程名称及一些寄存器的状态信息；

4、找到 linux 调度器的入口，实现 FCFS 调度逻辑：

- 从就绪队列中取出最先到达的进程
- 将该进程设置为运行状态

- 调用上下文切换函数，切换到该进程上执行

5、进程执行完成后，重新运行调度器，不断将排在就绪队列队首的进程调度运行；

6、重新编译内核，运行多个进程来进行测试，观察进程的切换情况是否满足 FCFS 的顺序。

参考文献

1、RISC-V Linux 上下文切换分析

https://tinylab.org/riscv-context-switch/

2、RISC-V Linux 内核调度详解

https://www.bilibili.com/video/BV1Fv4y137LF/?share_source=copy_web&vd_source=f43413ed69702715f11a3bb708e20bdf

操作系统复现实验

本实验要求同学们按下面的步骤实现一个操作系统，这是一个具有挑战性的任务，需要花费大量的时间学习新的知识。以下复现操作系统的步骤主要参考《**自己动手写操作系统**》，同学们也可以参照其他资料来复现一个操作系统。本文最后列出了一些有用的参考资料，同时欢迎同学们在复现操作系统的过程中根据自己的需求去寻找更多的资料。

1 实验环境及环境搭建

本实验在 Ubuntu 操作系统下完成，若同学们的工作环境是 Windows 操作系统，则需要首先在 Windows 操作系统下安装虚拟机软件（如 Virtualbox, VMware 等), 然后在虚拟机上安装 Ubuntu 操作系统。Bochs 是一款 X86 硬件平台的开源模拟器, 可以通过纯软件的方式模拟硬件, 从启动到重启，包括 PC 外设键盘、鼠标、VGA 卡、磁盘、网卡等全部通过软件来模拟，因此 Bochs 模拟器非常适合开发操作系统，其自带 bochsdbg 调试器使得调试操作系统变得更容易。因此本实验将在 Bochs 虚拟出来的计算机上进行我们所设计的操作系统的模拟测试和运行。参考的版本为：VMware 15.5.1；Ubuntu 20.04.2.0；NASM 2.15.05；Bochs 2.6.9。

简要步骤如下：

1) 卸载系统中已有的虚拟机如 Virtualbox 或 VMware, 安装最新版本的虚拟机；
2) 在虚拟机上安装 Ubuntu；
3) 下载 bochs 源码，编译安装；
4) 开始重新参考书上的第一个例子，操作系统引导盘的制作。

2 动手写一个最小的“操作系统”

实验内容：通过编译一段最基本的 asm 代码来初次体验操作系统的设计以及了解 NASM 编译的使用方法、dd 命令写入磁盘的方法以及 Bochs 的使用方法。

3 实现保护模式

实验内容：认识保护模式，实现从实模式到保护模式的转换，GDT 描述符；实现实模式大于 1MB 内存的寻址能力，并接着上一次实验，从保护模式返回到实模式，重新设置各个段寄存器的值；LDT 描述符；学会使用挂载指令和运行程序。

4 切换到保护模式

实验内容：引导扇区突破 512 个字节的限制，将工作分给 loader；加载 loader 进入内存并运行；将控制权交给 loader。

5 操作系统内核的雏形

实验内容：在 Linux 下用汇编写 Hello, World! ；进一步，汇编和

C 同时使用；从 loader 到 kernel 内核，把 kernel 内核加载到内存；将控制权交给 kernel 内核；跳入保护模式,并显示内存的使用情况。

6 进程与进程调度

实验内容：进程切换；丰富中断处理程序，比如让时钟中断处理可以不停地发生而不是只发生一次，进程状态的保存与恢复，进程调度，解决中断重入问题。具体可以见博客，博客的网址为 https://blog.csdn.net/qq_35353673/article/details/119010254

7 操作系统的输入/输出系统

实验内容：实现简单的 I/O，从键盘输入字符的中断开始；获取并打印扫描码；创建对应打印扫描码解析数组，打印对应字符。

8 复现操作系统的过程中可能存在的问题

(1) bochsrc 配置文件在新 bochs 版本下可能报错，因此需进行修改。文件的路径名取决于各位同学的系统。需将标记为红色的代码更改为下一行代码。

filename of ROM images

#vgaromimage: /usr/share/vgabios/vgabios.bin

vgaromimage: file=/usr/share/vgabios/vgabios.bin

enable key mapping, using US layout as default.

#keyboard_mapping:enabled=1,map=/usr/share/bochs/keymaps/x11-pc-us.

map

keyboard: keymap=/usr/share/bochs/keymaps/x11-pc-us.map

(2) 挂载点/mnt/floppy 不存在

该问题出现的原因是/mnt 目录下没有对应的/floppy 文件夹。

解决办法是在/mnt 目录下建立一个 floppy 文件夹，需要输入以下指令：

sudo mkdir /mnt/floppy

(3) ld 指令出现 incompatible with i386:x86-64 output

在实验 5.1 中，输入 ld -s hello.o -o hello 会报错，这是由于安装的 Ubuntu 是 64 位，默认产生 64 位的目标代码，但此处应编译 32 位目标代码，所以 ld 连接指令应改为 ld -m elf_i386 -s -o hello hello.o

(4) 第 6、7 章的代码调试过程中出现乱码或红色错误提示

代码 lib/kliba.asm 的 disp_str 函数中存在部分代码缺失，需要补充大红色的代码部分：

```
; from chapter6\a\lib\kliba.asm
disp_str:
        push    ebp
        mov     ebp, esp
        push    ebx
        push    esi
        push    edi
        mov     esi, [ebp + 8]      ; pszInfo
        mov     edi, [disp_pos]
        mov     ah, 0Fh
.1:
        lodsb
        test    al, al
        jz      .2
        cmp     al, 0Ah       ;  是回车吗?
        jnz     .3
        push    eax
        mov     eax, edi
        mov     bl, 160
        div     bl
        and     eax, 0FFh
        inc     eax
        mov     bl, 160
        mul     bl
        mov     edi, eax
```

```
        pop     eax
        jmp     .1
.3:
        mov     [gs:edi], ax
        add     edi, 2
        jmp     .1

.2:
        mov     [disp_pos], edi
        pop     edi
        pop     esi
        pop     ebx
        pop     ebp
        ret
```

针对上述复现过程中可能存在的版本问题，另一个解决办法是下载 32 位的 Linux 操作系统，使用 32 位的 bochs。

9 在按步骤复现了一个基本的操作系统之后，请实现以下功能：

(1) 自定义一个系统调用，能够统计一个进程在执行的过程中被调度的次数。编写一个简单的用户程序，调用该自定义的系统调用，从而将进程及其调度的次数输出在屏幕上。

(2) 在实现了三个进程的优先级调度的基础上，将三个进程的循环次数从无限循环修改为有限次数，当三个进程执行完成时，计算三个进程在优先级调度算法下的周转时间、等待时间以及该系统的平均周转时间、平均等待时间和吞吐量。也可以添加新的进程，然后计算该系统的平均周转时间、平均等待时间和吞吐量。

参考资料

1.《自己动手写操作系统》，电子工业出版社，于渊；该书第 2 版名为

《Orange'S：一个操作系统的实现》。虚拟机 bochs 可能不太好用，可以使用其他虚拟机软件，如 vmware。

2.《自己动手写操作系统》读后感，网址：

http://blog.csdn.net/zgh1988

3. 维基百科上与操作系统开发相关的网站：

http://wiki.osdev.org/Main_Page

4. 如何从零开始写一个简单操作系统，网址：

https://www.zhihu.com/question/25628124

5.《操作系统真相还原》，人民邮电出版社，郑钢，2016

6. MIT 的操作系统设计课程，网址：

https://pdos.csail.mit.edu/6.828/2017/xv6.html

xv6 起源于 MIT 的实验课程“操作系统工程”，英文名称是“Operating Systems Engineering”。xv6 是对 Dennis Ritchie 和 Ken Thompson 的 Unix 版本 6 (v6) 的重新实现，它松散地遵循 v6 的结构和风格，但使用 ANSI C 并基于 x86 多处理器实现。

7. 《Linux 内核完全注释》，赵炯，机械工业出版社， 2007。基于某 Linux 版本，在此基础上改写系统部分代码。

8. 《30 天自制操作系统》，川合秀实，人民邮电出版社，2012。

9.《一个 64 位操作系统的设计与实现》，田宇，人民邮电出版社，2018 年。

10. 清华大学操作系统设计课程网址：

https://chyyuu.gitbooks.io/ucore_os_docs/content/

11. https://os.educg.net/ 是 2021 年全国大学生计算机系统能力大赛操作系统设计赛的网站，技术报告那一栏有一些材料，感兴趣的同学可以下载来看。也可以将实验与参赛结合起来实现。