Lab3 Interrupt and Exception, Process and Schedule

本次实验的内容包括：

- 中断与异常
- 进程与调度

首先，合并新的代码：

```
git fetch --all
git checkout lab3
git merge lab2 # Or lab2-dev, or else
# Handle Marge Conflicts
```

Part I. Interrupt and Exception

我们会学习操作系统里中断相关的知识，将会涉及如下几个部分

1. 理解中断的流程
2. 了解中断的配置
3. 设计 trap frame，实现保存并恢复 trap frame （习题）
4. 理解中断处理函数

3.1.1 中断的流程

为了更好的了解中断，我们需要知道 CPU 在中断时究竟做了些什么。ARMv8 架构下中断时，CPU会先**关闭所有中断**（查看 DAIF 寄存器的值可以发现 DAIF 都被 mask 掉了，当然我们也可以手动开启，但为便于实现，我们暂不支持内核态的中断），根据中断向量表进行跳转，保存一些中断信息，并进行栈的切换。更具体的，

- 中断后自动关中断，这意味着所有的中断只会在用户态发生，即**只有从 EL0 到 EL1 的中断**
- 根据不同的中断种类，跳到中断向量表中的不同位置
- 保存中断前的 PSTATE 至 SPSR_EL1，保存中断前的 PC 到 ELR_EL1，保存中断原因信息到 ESR_EL1
- 将栈指针 sp 从 SP_EL0 换为 SP_EL1，这也意味着 SPSEL_EL1 的值为 1

3.1.2 中断的配置

3.1.2.1 中断向量表

在 `src/core/main.c, src/core/trap.c:init_trap()` 中我们用 `arch_set_vbar` 加载了ARMv8 的中断向量表 `src/asm/exception_vector.S`，其作用是，当硬件收到中断信号时，会根据其中断类型（Synchronous/IRQ/FIQ/System Error）和当前 PSTATE（EL0/EL1，AArch32/AArch64），保存中断原因到 ESR_EL1，保存中断发生时的 PC 到 ELR_EL1，然后跳转到不同的地址上。描述这些地址的那张表称为中断向量表。

中断向量表的地址可以通过 VBAR_EL1 配置（注意这是一个**虚拟地址**），表中的每一项是一种中断类型的入口，大小为 128 bytes，共有 16 项。上节中提到过我们只支持从 EL0 到 EL1 的中断，且我们的内核只支持 AArch64，于是在这里只需关注如下两项即可，其他的详见 [ARM Cortex-A Series Programmer's Guide for ARMv8-A] Chapter 10.4

Address	Exception type	Description	Usage
VBAR_EL1+0x400	Synchronous	Interrupt from lower EL using AArch64	System call
VBAR_EL1+0x480	IRQ/vIRQ	Interrupt from lower EL using AArch64	UART/timer/clock/sd

于是在 `src/asm/exception_vector.S` 中除了这两项，其他的都会执行 `src/core/trap.c:trap_error_handler` 来报错。

3.1.2.2 中断路由

中断产生后（多核情况下）硬件应该中断哪个 CPU 或哪些 CPU？这是我们需要告诉硬件的。树莓派比较特殊，所有的中断都会先发送到 GPU，再通过 GPU 来分配到具体哪些 CPU 上，详细请参考 BCM2836。我们只会用到如下几种中断，为了便于实现，我们将所有全局的中断都路由到了第一个核（CPU0）：

- UART 输入：在 qemu 中就是键盘输入，路由至 CPU0（后续会添加）
- 局部时钟（clock）：每个 CPU 都有的一种时钟中断，共四个，频率随 CPU 频率的变化而变化，用于记录进程的时间片，四个中断分别路由至四个 CPU
- SD卡设备：之后文件系统部分会加入，路由至 CPU0

上述已实现的中断的初始化和处理函数分别位于 `src/driver/uart.c`、`src/driver/clock.c`。

3.1.3 Trap frame

Trap frame 结构体的作用在于保存中断前的所有寄存器加上一些必要的中断信息。在我们的实验中，需要关注30个通用寄存器的值（x1~x30）以及系统寄存器 ELR_EL1，SPSR_EL1，SP_EL0。

3.1.3.1 习题一

请简要描述一下在你实现的操作系统中，中断时 CPU 进行了哪些操作。

3.1.3.2 习题二

请在 `src/core/trap.h` 中设计你自己的 trap frame，并简要说明为什么这么设计。

3.1.3.3 习题三

请补全 `src/asm/trap.S` 中的代码，完成 trap frame 的构建、恢复。

3.1.4 中断的处理

所有的合法中断会被 `src/asm/exception_vector.S` 发往 `src/exception_vector.S`，进而调用 `src/core/trap.c:trap_global_handler` 函数，这就是我们的中断处理函数。其中根据不同的中断来源而调用相应的处理函数。

3.1.4.1 测试中断

如果想测试一下中断是否正确实现的话，我们需要手动在内核开启中断（**后续请务必记得关闭**，除非你想支持内核态中断），看看我们的 timer/clock/uart 是否能够正常中断，需要做如下修改：

1. 在 `src/asm/exception_vector.S` 中将 `trap_error(4), trap_error(5)` 替换成 `enter_trap`，因为这一项代表在使用 SP_EL1 栈指针的 EL1 的 IRQ 中断。注意到 `src/start.S`

中 `msr spsel, #1` 指令已经将栈指针切换成了 SP_EL1，这就是我们的内核栈，之后用户进程使用的是 SP_EL0。
2. 在 `src/core/main.c:main` 函数最后调用 `inc/arm.h` 中的 `sti` 函数开启中断。

正常的话，qemu 上会有 timer/clock 的输出，输入字母的话也会显示在上面。

3.1.5 参考资料

实验过程中需重点关注 ARM Cortex-A Series Programmer's Guide for ARMv8-A 中的 Chapter 9 & 10，其中

- Chapter 9 规范了 ARMv8 架构下汇编和 C 语言之间的交互标准，如各寄存器的用处（哪个是作为函数返回地址的？哪些是 caller/callee 保存的寄存器？），stack frame 结构。在设计 trap frame 的时候会用到相关知识
- Chapter 10 是关于中断的，其中 10.5 有一段简易的参考代码，类似我们的 trapasm.S

Part II. Process and Schedule

3.2.1 实验内容简介

3.2.1.1 实验目标

在本次实验中，需要实现教学操作系统内核的进程管理部分，分为：

- 内核的进程管理模块，负责进程的创建、调度（执行）
- 内核的（简单）系统调用模块，负责响应用户进程请求的系统调用

最终实现将 `user/initcode.S` 作为第一个用户进程予以调度，并响应其执行过程中的系统调用。建议在实验中先切换为单处理器模式，确保单处理器下所有功能均正常，再切换为多处理器模式，以将并发问题与其他功能性问题做到分离。

3.2.1.3 xv6 手册

本次实验可以参考 xv6 实验手册

- Chap1：Operating system organization 从 Code: creating the first process 开始到 Chap1 截止的部分
- Chap5：Scheduling 部分

3.2.2 进程管理

我们在 `src/core/proc.h` 定义了 xv6 中进程的概念（本实验中仅需关注如下内容），每个进程都拥有自己的内核栈，页表（与对应的地址空间）以及上下文信息。

```
struct proc {
    uint64_t sz;             /* Size of process memory (bytes)          */
    uint64_t *pgdir;         /* Page table                              */
    char *kstack;            /* Bottom of kernel stack for this process */
    enum procstate state;    /* Process state                           */
    int pid;                 /* Process ID                              */
    struct proc *parent;     /* Parent process                          */
    struct trapframe *tf;    /* Trapframe for current syscall           */
    struct context *context; /* swtch() here to run process             */
};
```

3.2.2.1 问题一

在 proc（即 PCB）中仅存储了进程的 trapframe 与 context 指针，请说明 trapframe 与 context 的实例存在何处，为什么要这样设计？（Hint：如果在 proc 中存储 `struct context context` 与 `struct trapframe tf` 能否实现 trap 与 context switch）

3.2.2.2 Context switch

context switch（即上下文切换）是操作系统多道程序设计（multitasking/multiprogramming）的重要组成部分，它实现了从一个进程切换到另一个进程，此时需要切换能准确描述当前进程在 CPU 上运行情况的上下文信息包括通用寄存器堆、运行时栈（即每个进程的内核栈）以及 PC（lr/x30 寄存器）。

当用户进程放弃 CPU 时，该进程的内核线程（kernel thread）会调用 `src/asm/swtch.S` 中的 `swtch` 来存储当前进程的上下文信息，并切换到内核调度器的上下文，每个进程的上下文信息均由 `struct proc` 的 `struct context*` 描述。请完成 `src/core/proc.h` 中 `struct context` 的定义以及 `src/asm/swtch.S` 中 context switch 的实现。

3.2.2.3 问题二

在 `src/core/sched_simple.c` 中将 `swtch` 声明为 `void swtch(struct context **, struct context *)`，请说明为什么要这样设计（Hint：如果声明为 `void swtch(struct context *, struct context *)` 有什么问题）？`context` 中仅需要存储 callee-saved registers，请结合 PCS 说明为什么？与 trapframe 对比，请说明为什么 trapframe 需要存储这么多信息？trapframe **似乎** 已经包含了 context 中的内容，为什么上下文切换时还需要先 trap 再 switch?

> For the Arm architecture, the Procedure Call Standard, or PCS specifies:
>
> - Which registers are used to pass arguments into the function.
> - Which registers are used to return a value to the function doing the calling, known as the caller.
> - Which registers the function being called, which is known as the callee, can corrupt.
> - Which registers the callee cannot corrupt.

PCS 是 ARM 架构在函数调用过程中通用寄存器的使用规范，当我们自己手写汇编，同时会借助其他开源工具如编译器生成汇编指令时，应当尽量遵守这个规范，以保证代码整体正确、高效。[1]

3.2.2.4 创建与调度进程

请根据 `src/core/proc.c` 中相应代码的注释完成内核进程管理模块以支持调度第一个用户进程 `src/user/init.S`。

3.2.3 系统调用

目前内核已经支持基本的异常处理，在本实验中还需要进一步完善内核的系统调用模块。用户进程通过系统调用来向操作系统内核请求服务以完成需要更高权限运行的任务。在 armv8 中，用户进程通过 `svc` 指令来请求系统调用，内核会根据 `ESR_EL1` 中的 `EC[31:26]` 以分派给相应的 handler 进行处理[2]。

用户进程在请求系统调用时，应告知内核相应的 system call number 以及系统调用所需的参数信息，system call number 的宏定义可见 `src/core/syscallno.h`。

3.2.4 参考文献

Exercise

截止时间：`2021-10-22 15:24:59`。

提交方式：将实验报告提交到`elearning`上，文件名：`学号-lab3.pdf`。

---

1. https://developer.arm.com/architectures/learn-the-architecture/aarch64-instruction-set-architecture/procedure-call-standard
2. https://developer.arm.com/docs/ddi0595/h/aarch64-system-registers/esr_el1