操作系统原理课程设计报告

姓　　名：　　路昊东

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　CS2011

学　　号：　　U202010755

指导教师：　　阳富民

分数	
教师签名	

2024 年 4 月 1 日

目　录

实验一 打印异常代码行

1.1 实验目的

本实验的目标是通过修改 PKE（Protected Kernel Extension）内核的代码，使得当用户程序发生异常时，内核能够输出触发异常的用户程序的源文件名和对应的代码行。特别地，虽然在示例的 app_errorline.c 中只触发了非法指令异常，但最终测试时，你的内核也应能够对其他会导致 panic 的异常和其他源文件输出正确的结果。

1.2 实验内容

总体来讲，实验步骤主要包括四点：

- ✧ 修改内核代码以读取 ELF 文件的.debug_line 段；
- ✧ 实现 make_addr_line()函数；
- ✧ 在 machine/mtrap.c 中编写 print_errorline()函数；
- ✧ 在每个由用户程序异常导致的中断前调用 print_errorline()函数。

首先，在读取 ELF 文件时，我们需要检索并解析包含的 .debug_line 段，以便内核能够获取用户程序的程序名和代码。为此，我们将 process 结构体的 debugline 字段改为一个足够长的字符串地址，以便存储 .debug_line 段的内容。

为了能够判别读到的段是否为.debug_line 段，我们首先读取 string table 段。string table 段包含了 ELF 文件中所有字符串的列表，包括段名。接着，我们逐个遍历 ELF 的各段，并根据 string table 的首地址和这个段的 name 字段指示的偏移找到此段的名称。如果此段的名称与.debug_line 段相同，那么就将这个段读入当前进程的 debugline 指针所指的位置。

之后，我们调用 make_addr_line()函数来为 process 结构体的 dir、file、line 数组赋值。这个函数会解析.debug_line 段的内容，并填充 dir、file、line 数组，以便内核能够根据指令地址查找对应的源代码文件名和行号。

下面是相关代码以及具体注释：

```
// 1. 加载名称段：读取 ELF 文件的名称段，该段包含了段名称的字符串表
if (elf_fpread(ctx, (void *)&section_header1, sizeof(section_header1),
    ctx->ehdr.shoff + ctx->ehdr.shstrndx * sizeof(section_header1)) !=
    sizeof(section_header1)){
```

```
            return EL_EIO; // 如果读取名称段失败，返回错误
        }
// 2. 加载调试行：遍历 ELF 文件的所有段，查找名称为.debug_line 的段
        for (i = 0, off = ctx->ehdr.shoff; i < ctx->ehdr.shnum; i++, off += sizeof(section_header2)) {
            // 3. 如果读取段头失败，返回错误
            if (elf_fpread(ctx, (void *)&section_header2,
                    sizeof(section_header2), off) != sizeof(section_header2)) {
                    return EL_EIO;
            }
            // 4. 读取段名：根据段头中的 name 字段和名称段的偏移，读取段的名称
            elf_fpread(ctx,
                        (void *)segment_name, 20,
                        section_header1.offset + section_header2.name);
            // 5. 如果段名与.debug_line 匹配，读取并存储.debug_line 段的内容
            if (strcmp(".debug_line", segment_name) == 0) {
                // 6. 如果读取.debug_line 段的内容失败，返回错误
                if (elf_fpread(ctx, ((process*)(((elf_info*)(ctx->info))->p))->debugline,
                   section_header2.size,
                   section_header2.offset) != section_header2.size) {
                      return EL_EIO;
                }
                // 7. 调用 make_addr_line()函数来解析.debug_line 段的内容，并填充相关数组
                make_addr_line(ctx,
                                ((process*)(((elf_nfo*)(ctx->info))->p))->debugline ,
                                section_header2.size);
                // 8. 如果成功找到并读取了.debug_line 段，则跳出循环
                break;
            }
        }
```

简言之，这段代码的目的是从 ELF 文件中读取.debug_line 段，并将其存储在 process 结构体的 debugline 字段中。如果在读取过程中出现错误，函数将返回 EL_EIO 错误代码。一旦找到.debug_line 段，函数将调用 make_addr_line()函数来解析段的内容，并填充相关的数组。

获取信息后，在 machine/mtrap.c 中，我们可以编写一个名为 print_errorline()的函数，该函数将在用户程序发生异常时被调用。

这个函数的主要任务是根据出错的位置和进程中的调试信息找到用户程序出错的代码行并输出。以下是 print_errorline()函数的详细步骤：

- **获取触发异常的指令地址**：通过读取 mepc 寄存器的值，我们可以获取触发异常的指令的地址。mepc 寄存器通常用于存储最后一条执行的指令的地址；
- **生成出错用户程序的地址**：有了指令地址后，我们需要根据当前进程的 line 数组来生成出错用户程序的地址。line 数组包含了用户程序中每行代码对应的指令地址；
- **读取出错代码行的具体内容**：一旦我们确定了出错的代码行，我们可以使用 spike 文件操作函数簇来读取该行代码的具体内容。这些函数簇通常允许我们打开文件、读取文件内容、关闭文件等操作；
- **打印出错代码行**：最后，我们可以使用 spike 的输出函数将出错代码行的具体内容打印出来。

在 machine/mtrap.c 中，我们可以将 print_errorline()函数放在每个由用户程序异常导致的中断处理函数的前面。这样，当用户程序发生异常时，内核就会自动调用 print_errorline()函数，并打印出用户程序出错的代码行。

运行结果如图 2-1 所示：

```
******************HUST PKE********************
spike -p2 obj/riscv-pke /bin/shell /bin/help
In m_start, hartid:0
In m_start, hartid:1
hartid = 0, User application is loading.
hartid = 0, Application: /bin/shell
hartid = 1, User application is loading.
hartid = 1, Application: /bin/help
hartid = 0, Application program entry point (virtual address): 0x00000000000100d4
hartid = 1, Application program entry point (virtual address): 0x00000000000100b0
Welcome to RISCV-PKE Shell.
The following commands are supported in this shell.
```

```
>>> /$ /bin/mkdir /RAMDISK0/sub_dir
hartid = 0, Application: /bin/mkdir
hartid = 0, Application program entry point (virtual address): 0x00000000000100b0
Runtime error at user/app_mkdir.c:7
  asm volatile("csrw sscratch, 0");
Illegal instruction!
System is shutting down with exit code -1.
make: *** [Makefile:290: run] 错误 255
```

图 2-1 运行输出

1.3 实验调试及心得

在进行这个实验的过程中，我们需要对 ELF（Executable and Linkable Format）和 DWARF（Debugging With Arbitrary Record Formats）格式有深入的理解。ELF 格式是 Unix 和类 Unix 系统上用于可执行文件、目标文件、共享库和核心转储的

文件格式。DWARF 是一种用于存储调试信息的格式，它可以与 ELF 文件一起使用，以提供源代码级别的调试信息。

在实验过程中，我们可以使用 readelf 命令来查看 ELF 文件的结构和内容，使用 dwarfdump 命令来查看 DWARF 调试信息。这些工具可以帮助理解 ELF 和 DWARF 格式，以及如何从这些格式中提取有用的信息。

当然，在编写内核代码时，我们会遇到各种错误和问题，例如段错误、内存泄漏或者其他类型的运行时错误。为了调试这些问题，我们需要使用调试工具如 gdb。gdb 是一个强大的命令行调试器，它可以帮助找到代码中的错误，并提供有关变量、寄存器和内存的详细信息。

在实验过程中，我们还需要理解操作系统的内核设计和工作原理。例如需要理解中断处理、进程调度、内存管理等方面的知识。这些知识可以帮助我更好地理解内核的行为，并能够有效地调试内核代码。

总的来说，这个实验是一个很好的机会，可以让我深入了解操作系统的内核设计和调试技术。通过这个实验，不仅可以提高自己的编程技能，还可以更好地理解操作系统的工作原理。

实验二 多核启动及运行

1.1 实验目的

在 RISC-V 处理器中，每个 CPU 被称为一个 hart（硬件线程），从 0 开始编号。本实验的目的是修改 PKE 内核的代码，使其能够通过 spike 启动两个 CPU（hart），并让其中一个（hart 0）执行 app0，另一个（hart 1）执行 app1。当两个进程都执行结束之后，操作系统关闭。

app0 和 app1 的加载地址分别为 0x81000000 和 0x81500000。另外，在 kernel/config.h 文件中，NCPU 参数设置为 2，以便通过 spike -p2 riscv-pke app0 app1 命令启动两核并发的操作系统并执行 app0 和 app1。

本实验在 elf.c 中给出了.debug_line 段的解析函数 make_addr_line()。函数调用结束后，process 结构体的 dir, file, line 三个指针会各指向一个数组，dir 数组存储所有代码文件的文件夹路径字符串指针；file 数组存储所有代码文件的文件名字符串指针以及其文件夹路径在 dir 数组中的索引；line 数组存储所有指令地址，代码行号，文件名在 file 数组中的索引三者的映射关系。

在内核代码中，使用 sprint 输出时，需要在输出中包含 hartid，以便区分每一条输出是哪个核执行的。

1.2 实验内容

在本次实验中，我们将使用 Spike 模拟器来模拟多核处理器，并通过命令行参数来加载两个用户程序。首先需要确保每个核的时钟周期和中断是独立的，并且需要修改代码以处理多个应用程序的内存重叠问题；此外，还需要确保在所有核执行完毕后，才能关闭模拟器。具体来讲分为以下几点：

1. 初始化函数

在 RISC-V 架构的 PKE 操作系统中，为了支持两核并发启动并保证各种初始化操作仅执行一次，我们需要对系统启动的相关函数进行修改。PKE 操作系统的启动分为两个阶段，分别由 machine/minit.c 中的 m_start 函数和 kernel.c 中的 s_start 函数控制。

在 m_start 函数中，spike_file_init()和 init_dtb()两个函数涉及到系统资源的初始化，这些操作只需要在一个核上执行一次。因此，我们规定编号为 0 的核（通常是第一个启动的核）执行这两个初始化函数。

为了防止其他核在未进行初始化的情况下执行后面的操作，我们在初始化函数后添加一个同步语句，以确保后续的操作在初始化完成后才执行。同步之后，我们还需要为每个 CPU 的 tp（Thread Pointer）寄存器设置当前核的编号，以便在函数作用域内可以知道当前是哪个核在调用这个函数。

为了方便地知道进程是在哪个核上执行的，我们在 process 结构体中添加一个成员 hart_id 来表示核编号。以下是具体的代码实现：

```
// 在 m_start 函数中
if   (hartid == 0) {
      spike_file_init();
      init_dtb(dtb);
}
// 同步所有核
sync_barrier(&cnt, NCPU);
// 为每个 CPU 的 tp 寄存器设置当前核的编号
write_tp(hartid);
// 在 process 结构体中添加 hart_id 成员
struct process {
      // ...其他成员...
      int hart_id; // 核编号
};
```

在这个代码中，hartid 是当前核的编号，cnt 是一个全局变量，用于同步所有核，NCPU 是系统中核的总数，write_tp 是一个函数，用于设置 tp 寄存器的值。spike_file_init 和 init_dtb 是初始化函数，它们只在编号为 0 的核上执行。

当然，由于实验做了简化，这段代码是一个示例，实际的实现可能需要处理更多的边缘情况和错误检查。此外，sync_barrier 函数的实现细节并没有在这里展示，因为它取决于具体的同步机制和架构。

在 s_start 函数中，我们需要处理多个核共享的全局变量。为了避免数据竞争和同步问题，我们将 user_app 和 current 变量转换为数组，每个核通过其核编号（hart_id）来访问对应的数组元素。由于每个核的操作不会同时访问同一个数组元素，所以我们不需要使用互斥锁来保护这些共享资源。

2. 多核时钟中断

我们需要处理多核环境下的时钟中断问题，需要确保每个核有独立的时钟计数。同样地，全局变量 g_ticks 也需要转换成多份资源，因此我们将其设置为数

组，每个核通过其核编号来访问对应的数组元素。在 strap.c 中的 handle_mtimer_trap() 函数中，我们首先通过 tp 寄存器得知当前 CPU 的编号，然后通过编号访问 g_ticks 数组，增加其值。最后复位 sip 寄存器。

3. 从命令行加载多个程序

在单核实验中，pke 从命令行第一个参数加载应用程序。为了支持从命令行前两个参数加载两个程序的功能（实验要求），我们需要修改 elf.c 中的 load_bincode_from_host_elf 函数：在经过 parse_args 函数调用之后，arg_bug_msg.argv 数组中会依次存储各核将要运行的程序的路径。我们利用 p->hart_id 作为下标访问此数组，即可得知当前核要加载的 elf 文件路径。

4. 防止内存重叠

由于进程的内存都是事先被分配的，我们需要为这两个核上的进程指定两片一定不会重叠的内存区域。在 kernel.c 中的 load_user_program()函数中，我们分别设置固定的内存。为了实现这一点，我们需要在内存管理子系统中预留出足够的空间，以便为每个核分配独立的内存区域。在我们的实验环境中，这意味着为每个核分配一个固定的内存范围：

hart_id	**用户栈**	**用户系统栈**	**trapframe**
0	0x81100000	0x81200000	0x81300000
1	0x85100000	0x85200000	0x85300000

在 load_user_program()函数中，我们需要根据当前核的编号来确定应该使用哪个内存区域。例如，我们可以通过 p->hart_id 来获取当前核的编号，然后根据这个编号来确定内存区域的起始地址。

此外，由于 current 数组已经被我们设置成了数组，所以所有用到 current 的地方都需要首先使用 write_tp()并以此为下标访问 current 数组得到当前核上的当前进程。这样，我们就确保了每个核都有独立的内存区域来运行进程，从而避免了内存重叠的问题。

5. 同步关闭模拟器

在多核环境下，我们需要确保在所有核的进程都执行完毕后再关闭模拟器。为了实现这一点，我们需要在 syscall.c 中的 sys_user_exit()函数中添加一个同步机制。

具体来说，我们可以在执行 shutdown()之前添加一个 sync_barrier()函数调用。这个函数会阻塞当前核的执行，直到所有其他核也调用了 sync_barrier()函数。这样，我们就可以确保所有核上的进程都已经执行完毕。

以下是具体的代码实现：

```
// 在 syscall.c 中的 sys_user_exit() 函数中
void sys_user_exit(int code) {
    // 同步所有核
    sync_barrier(&exit_cnt, NCPU);
    // 只有编号为0 的核（即第一个核）会执行关闭操作
    if (current[read_tp()]->hart_id == 0) {
        // 打印一条消息，指示哪个核正在执行关闭操作
        sprint("hartid = %d: shutdown with code:%d.\n", current[read_tp()]->hart_id, code);
        // 执行关闭操作
        shutdown(code);
    }
}
```

在这个代码中，sync_barrier()是一个假设的函数，它会阻塞当前核的执行，直到所有其他核也调用了 sync_barrier()函数。exit_cnt 是一个全局变量，用于跟踪已经调用了 sync_barrier()的核的数量。NCPU 是系统中核的总数。

current[read_tp()]->hart_id 用于获取当前核的编号。如果当前核的编号是 0，那么它将执行关闭操作。shutdown(code)是关闭模拟器的函数，其中 code 是关闭状态码。

这样，通过使用 sync_barrier()函数，我们确保了只有当所有核上的进程都执行完毕后，才会执行关闭操作，从而实现了同步关闭模拟器的需求。

至此，我们就完成了对多核的支持。运行结果如图 2-2 所示。

```
******************HUST PKE******************
spike -p2 obj/riscv-pke obj/app0 obj/app1
HTIF is available!
(Emulated) memory size: 2048 MB
In m_start, hartid:0
hartid = 0: Enter supervisor mode...
hartid = 0: Application: obj/app0
hartid = 0: Application program entry point (virtual address): 0x0000000081000000
hartid = 0: Switch to user mode...
In m_start, hartid:1
hartid = 1: Enter supervisor mode...
hartid = 1: Application: obj/app1
hartid = 1: Application program entry point (virtual address): 0x0000000085000000
hartid = 1: Switch to user mode...
hartid = 0: >>> app0 is expected to be executed by hart0
hartid = 1: >>> app1 is expected to be executed by hart1
hartid = 0: User exit with code:0.
hartid = 1: User exit with code:0.
hartid = 0: shutdown with code:0.
System is shutting down with exit code 0.
```

图 2-2 运行输出

1.3 实验调试及心得

在本次实验中，我们需要对 PKE 操作系统进行修改，以支持多核并发启动和运行。通过亲自设计一个支持两个硬件线程并发执行的操作系统，我们可以深入理解进程的同步和互斥。

首先，我们需要理解多核操作系统的基本概念。多核处理器可以同时执行多个线程，这需要操作系统能够有效地管理和调度这些线程，以避免资源争用和死锁。在我们的实验中，我们需要确保每个核的初始化操作仅执行一次，并且在后续操作中，每个核都能独立地访问和修改其资源。

在具体的编程任务中，我们需要修改 Spike 模拟器的启动代码，以支持多核并发启动。这包括在 m_start 函数中对 spike 模拟器的一些虚拟设备和 HTIF 接口进行初始化，以及在 s_start 函数中对操作系统进行初始化。我们还需要修改时钟中断处理函数，以确保每个核都有独立的时钟计数。

在实验中，我们还需要修改 elf.c 中的 load_bincode_from_host_elf 函数，以支持从命令行前两个参数加载两个程序。我们还需要修改 kernel.c 中的 load_user_program 函数，为每个核指定不同的内存区域，以防止进程内存重叠。

在实验的最后，我们需要修改 syscall.c 中的 sys_user_exit 函数，以实现同步关闭模拟器。我们通过在执行 shutdown 之前添加一个 sync_barrier 函数调用来保证所有核上的进程全部执行完毕时再关机。

通过本次实验，我对进程的同步和互斥有了更深的理解。我学习到了如何在多核环境中管理和调度线程，如何避免资源争用和死锁，以及如何实现进程间的同步。同时，我也体会到了操作系统设计的复杂性和挑战性。在接下来的学习和工作中，我将继续深入理解操作系统的相关知识，并尝试设计和实现更复杂的操作系统。