操作系统原理课程设计报告

姓　　名：　董玲晶

学　　院：　计算机科学与技术学院

专　　业：　计算机科学与技术

班　　级：　CS2005

学　　号：　U202090063

指导教师：　石　柯

分数	
教师签名	

2023 年 04 月 01 日

目　录

实验一 打印用户程序调用栈

1.1 实验目的

实现系统调用并完善 print_backtrace( )函数，根据传入的参数能够打印出相应的函数名，同时以倒序输出调用顺序。打印实例如图 1.1 所示。

```
1. In m_start, hartid:0
2. HTIF is available!
3. (Emulated) memory size: 2048 MB
4. Enter supervisor mode...
5. Application: obj/app_print_backtrace
6. Switching to user mode...
7. back trace the user app in the following:
8. f8
9. f7
10. f6
11. f5
12. f4
13. f3
14. f2
15. User exit with code:0.
16. System is shutting down with exit code 0.
```

图 1.1 challenge_lab_1_1 输出示例

1.2 实验内容

首先是添加系统调用，完善 print_backtrace( )函数并在 PKE 用户函数库及内核中实现它的系统调用。所以在 user_lib.c 中实现 print_backtrace( )函数的功能，然后再 user_lib.h 中添加函数原型的声明；同时，再 syscall.h 中为 SYS_user_backtrace 注册一个 ID 号，然后再 syscall.c 中为上述函数添加系统调用。部分对应代码如下代码 1.1 所示。

```
// user_lib.h
int print_backtrace(int depth);
// user_lib.c
int print_backtrace(int depth) {
    return do_user_call(SYS_user_backtrace, depth, 0, 0, 0, 0, 0, 0);
}
```

```
// syscall.h
# define SYS_user_backtrace (SYS_user_base + 2)
// syscall.c 的 do_syscall()函数中添加 case
case SYS_user_backtrace:
    return sys_user_backtrace(a1);
```

代码 1.1-1 用户函数实现和添加系统调用代码

添加完系统调用后要实现上述代码中的 sys_user_backtrace( )函数，主要思路是要找到 print_backtrace( )函数调用的第一个函数的返回地址，然后再依次向栈顶的方向加上 16（每个调用函数的栈帧大小），即为下一个调用函数的返回地址。而要找到所调用的第一个函数的返回地址，首先要通过 trapframe 存储的进程上下文获取用户态栈 SP（因为此时在内核栈），对寄存器 SP 的当前的值（即栈顶地址）加上 32 得到 print_backtrace( )的栈帧，再对其加上 8 即可，此时的值即为调用的第一个函数的返回地址。然后只需要一个循环即可解决问题。

```
ssize_t sys_user_backtrace(uint64 depth) {
  uint64 glamoura = 0;
  uint64 trace_sp = current->trapframe->regs.sp + 32;
  uint64 trace_ra = trace_sp + 8;
  for(; glamoura<depth; glamoura++) {
    if(name_printer(*(uint64*)trace_ra) == 0)
      return glamoura;
    trace_ra += 16;
  }
  return glamoura;
}
```

代码 1.1-2 系统调用顶层函数 sys_user_backtrace( ) 函数实现

在循环中需要调用一个函数来打印对应的函数名，具体实现方法为通过上述提到的函数返回地址反向解析对应的函数名。此时就涉及到了符号表、字符串表和逻辑地址到符号的解析。

通过实验文档中的内容关于 ELF 文件结构的学习，我们可以得到要做到反向解析，我们就需要根据 elf_header 找到 elf_section_header，再通过 name 来读取每个 section 的 sh_name 和 sh_type，从中找到.strtab 和.symboltab。因此首先我们需要编写一个函数实现上述加载.strtab 和.symboltab 的加载，并在 load_bincode_from_host_elf( )函数中实现调用；同时，在 elf.h 文件中添加 sh_type 的合法值以用于 load 函数的判断，并添加 elf_section_header 的结构和 load 函数声明。

其中加载函数的部分代码如代码 1.1-3 和 1.1-4 所示。

```
// FIND TAB OF FIRST
for(int i=0; i<number_section; i++) {
  elf_fpread(ctx, (void*)&tempt_sh, sizeof(tempt_sh), ctx->ehdr.shoff+i*ctx->ehdr.shentsize);
  uint32 judging_type = tempt_sh.sh_type;
  // CONFIRM WHETHER IS THE EXACT TYPE
  if (judging_type == SHT_STRTAB) {
    if (strcmp(temp_str + tempt_sh.sh_name, ".strtab") == 0) {
      string_sh = tempt_sh;
    }
  } else if (judging_type == SHT_SYMTAB) {
    symbok_sh = tempt_sh;
```

代码 1.1-3 加载.strtab 和.symboltab 函数实现（1）

```
// CONTINUE
int count = 0;
uint64 section_string_offset = string_sh.sh_offset;
uint64 symbol_number = symbok_sh.sh_size / sizeof(elf_symbol);
for (int i=0; i<symbol_number; i++) {
  elf_symbol symbol;
  elf_fpread(ctx, (void*)&symbol, sizeof(symbol), symbok_sh.sh_offset+i*sizeof(elf_symbol));
  // WHETHER THE BEGIN
  // NO AND YES REFER TO symbol.st_name=18 and 0 respectively
  if (symbol.st_name != 0) {
    if (symbol.st_info == 18) {
      char symbol_name[32];
      elf_fpread(ctx, (void*)&symbol_name, sizeof(symbol_name), section_string_offset + symbol.st_name);
      symbols[count] = symbol;
      strcpy(sym_names[count], symbol_name);
      count++;
    }
```

代码 1.1-4 加载.strtab 和.symboltab 函数实现（2）

1.3 实验调试及心得

在本次实验中，我学到了关于 elf 文件的知识，了解了 elf 文件在编程中的表达方式。同时，我也学会了如何在内核中添加系统调用的接口，并在用户程序中调用这些接口来完成特定的功能。在实现 print_backtrace 函数时，我需要在内核中添加对应的系统调用，用来获取和打印用户程序的调用栈。通过了解用户程序和内核之间的交互方式，我能够正确地实现系统调用，并在用户程序中使用该函数打印调用栈。我还学会了如何使用调试工具来帮助调试内核和用户程序。在本次实验中，我使用了 spike 模拟器来模拟 risc-v 处理器的运行，通过使用调试工具，我更好地理解到了操作系统的运行原理和实现方式。

总的来说，本次实验让我更深入地了解了操作系统的实现原理和内核开发的基本技术，提高了我的编程技能和操作系统的实际应用能力。同时，也让我更深刻地认识到操作系统的重要性和作用，对我的专业发展和个人成长都有着积极的促进作用。

上述的一些心得也是我整个课设的一些心得，所以后面几个实验的心得可能会写得简短一些~

实验二 打印异常代码行

1.1 实验目的

修改内核的代码包括但不限于完善对应函数和添加系统调用等，使得用户程序在发生异常时，内核能够输出触发异常的用户程序的源文件名和对应代码行，打印实例如图 1.2 所示。

```
$ spike ./obj/riscv-pke ./obj/app_errorline
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
Application: ./obj/app_errorline
Switch to user mode...
Going to hack the system by running privilege instructions.
Runtime error at user/app_errorline.c:13
  asm volatile("csrw sscratch, 0");
Illegal instruction!
System is shutting down with exit code -1.
```

图 1.2 challenge_lab_1_2 输出示例

1.2 实验内容

根据实验文档指示，需要读取 elf 文件中名为.debug_line 的段保存到缓冲区中，然后将缓冲区指针传入这个参数。我们先根据 challenge1_1 的方法找到.strtab，再根据其 name 和 offset 找到 debug_line 段，将指向端数据的指针和我们解析出来的段数据长度以及 elf 文件上下文指针一起传给 make_addr_line 函数。同时我们也像 1-1 那样需要在 load_bincode_from_host_elf( )函数中实现调用。部分代码如代码 1.2-1 所示。

```
uint16 sect_num = ctx->ehdr.shnum;
uint64 shstr_offset = ctx->ehdr.shoff + ctx->ehdr.shstrndx * sizeof(elf_sect_header);
elf_fpread(ctx, (void *)&shstr_sh, sizeof(shstr_sh), shstr_offset);
char temp_str[shstr_sh.size];
uint64 shstr_sect_off = shstr_sh.offset;
elf_fpread(ctx, &temp_str, shstr_sh.size, shstr_sect_off);
for (int i=0; i<sect_num; i++) {
  elf_fpread(ctx, (void*)&temp_sh, sizeof(temp_sh), ctx->ehdr.shoff+i*ctx->ehdr.shentsize);
  if(strcmp(temp_str+temp_sh.name,".debug_line") == 0) {
    debug_line_sh = temp_sh;
  }
}
elf_fpread(ctx, (void*)&dbline_buf, debug_line_sh.size, debug_line_sh.offset);
make_addr_line(ctx, dbline_buf, debug_line_sh.size);
```

代码 1.2-1 加载 debug_line 找到.strtab 代码

以上虽然是通过函数实现解析出了.debug_line 段的内容，我们还要对每次打印异常代码进行中断处理。首先我们需要找到造成异常的代码的所在文件路径，我们根据文件索引找到对应的文件名和文件夹索引，最后找到文件夹名，使用字符串处理将他们拼接成文件路径。据此读入源代码文件后根据换行符计算出具体行号，打印出具体的异常代码。部分代码如 1.2-2 所示。

```
addr_line * excpt_line = current - > line + i - 1;

int dir_len = strlen(current - > dir[current - > file[excpt_line - > file].dir]);

strcpy(full_path, current - > dir[current - > file[excpt_line - > file].dir]);
full_path[dir_len] = '/';
strcpy(full_path + dir_len + 1, current - > file[excpt_line - > file].file);

spike_file_t * _file_ = spike_file_open(full_path, O_RDONLY, 0);
spike_file_stat(_file_, & f_stat);
spike_file_read(_file_, full_file, f_stat.st_size);
spike_file_close(_file_);
```

代码 1.2-2 读取异常代码所在文件并找到行号

最后我们在所有需要打印异常代码行的情况下添加错误代码打印中断，并也在 handle_mtrap( )函数中完善。部分代码如 1.2-3 所示。

```
static void handle_instruction_access_fault() {
  error_printer();
  panic("Instruction access fault!");
}

static void handle_load_access_fault() {
  error_printer();
  panic("Load access fault!");
}

static void handle_store_access_fault() {
  error_printer();
  panic("Store/AMO access fault!");
}
```

代码 1.2-3 添加 error_printer( )中断

1.3 实验调试及心得

本次实验通过实现对异常代码的打印输出，加深了我对操作系统异常处理机制的理解。异常处理是操作系统内核的一个重要组成部分，负责监控和处理程序运行过程中发生的各种异常情况。同时本实验也继续加深了我对在操作系统内核中添加系统调用的接口，并在用户程序中调用这些接口来完成特定的功能这个方法的掌握。

实验三 实现信号量

1.1 实验目的

添加关于信号量功能的数据结构和对应函数，实现信号量的功能，从而实现并发进程之间的同步和互斥操作。

1.2 实验内容

首先是添加信号量的数据结构，设计为一个结构体，具体如代码 1.3-1 所示。包含信号量目前的值、目前的状态（有没有被使用）以及队列头指针，这个队列为等待进程队列。定义一个进程池初始化函数，对所有信号量进行初始化操作。

```
typedef struct semaphore_t {
  int value;
  uint64 stat;
  process *p_queue;
} semaphore;
```

代码 1.3-1 信号量数据结构定义

实现信号量的各种函数操作，包括 P 操作、V 操作、新增信号量操作。P 操作实现包括对信号量的值减一、判断当前进程是否需要被阻塞、将进程加入等待队列、将当前进程的状态设置为阻塞以及调度下一个可运行进程。V 操作实现包括增加信号量的值、判断等待队列中是否由进程等待、从等待队列中取出一个等待进程并将其插入就绪队列中。新增新信号量的操作包括查找未被使用的信号量并初始化信号量。部分代码如下代码 1.3-2 所示。

```
int sys_user_semaphore_P(int n) {
  sems[n].value = sems[n].value - 1;
  if (sems[n].value < 0) {
    if (sems[n].p_queue != NULL) {
      process *temp = sems[n].p_queue;
      while (temp->queue_next > 0) {
        temp = temp->queue_next;
      }
      temp->queue_next = current->queue_next;
      temp = current;
    } else {
      sems[n].p_queue = current;
      current->queue_next = NULL;
    }
    current->status = BLOCKED;
    schedule();
```

代码 1.3-2 信号量 P 操作

在 syscall.c 中添加信号调用，并在 syscall.h 中注册各个 ID 号，这里不再赘述。

1.3 实验调试及心得

本实验相较于前面两个实验较为简单，但让我在上学期学习信号量的课程后，尝试了在操作系统中实现信号量，这是一件挺有意思的事情，让我感到很有趣。同时这个实验也加深了我对信号量这个知识点的理解，也通过代码编写深入理解了 PV 操作对操作系统的影响。