操作系统原理课程设计报告

姓　　名：　　王彬

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　计卓 2101 班

学　　号：　　U202112071

指导教师：　　阳富民

分数	
教师签名	

2024 年 1 月 6 日

目　录

实验一 打印用户程序调用栈

1.1 实验目的

通过修改 PKE 内核，来实现从给定应用（user/app_print_backtrace.c）到预期输出的转换。

1.2 实验内容

对于 print_backtrace()函数的实现要求，应用程序调用 print_backtrace()时，应能够通过控制输入的参数（如例子 user/app_print_backtrace.c 中的 7）控制回溯的层数。例如，如果调用 print_backtrace(5)则只输出 5 层回溯；如果调用 print_backtrace(100)，则应只回溯到 main 函数就停止回溯（因为调用的深度小于 100）。

为完成该挑战，PKE 内核的完善应包含以下内容：

- 系统调用路径上的完善，可参见 3.2 中的知识；
- 在操作系统内核中获取用户程序的栈。这里需要注意的是，PKE 系统中当用户程序通过系统调用陷入到内核时，会切换到 S 模式的“用户内核”栈，而不是在用户栈上继续操作。我们的 print_backtrace()函数的设计目标是回溯并打印用户进程的函数调用情况，所以，进入操作系统内核后，需要找到用户进程的用户态栈来开始回溯；
- 找到用户态栈后，我们需要了解用户态栈的结构。实际上，这一点在我们的第一章就有举例来说明，读者可以回顾一下第一章的例子；
- 通过用户栈找到函数的返回地址后，需要将虚拟地址转换为源程序中的符号。这一点，读者需要了解 ELF 文件中的符号节（.symtab section），以及字符串节（.strtab section）的相关知识，了解这两个节（section）里存储的内容以及存储的格式等内容。对 ELF 的这两个节，网上有大量的介绍，例如这里，或阅读 Linux Man Page。

首先在应用层添加相应的库函数，我们定义 SYS_print_user_backtrace，并将打印参数深度作为第二个参数传入。

```
// Added @lab1_challenge1.
int print_backtrace(int depth){
  return do_user_call(SYS_print_user_backtrace,depth,0,0,0,0,0,0);
}
```

为了保证顺利调用，还需要在 syscall.c 中的 do_syscall 函数内加入对于传入的第一个参数的选择。

```
long do_syscall(long a0, long a1, long a2, long a3, long a4, lon
  switch (a0) {
    case SYS_user_print:
      return sys_user_print((const char*)a1, a2);
    case SYS_user_exit:
      return sys_user_exit(a1);
    case SYS_print_user_backtrace:
      return print_user_backtrace(a1);
    default:
      panic("Unknown syscall %ld \n", a0);
  }
}
```

我们通过编写具体的 print_user_backtrace 函数对于打印调用栈进行支持。具体地，我们需要做的是：

- 在 kernel 内获取函数的调用信息；
- 读取.elf 节头信息，解析出相应函数名称字符串。

对于用户调用栈的跟踪，我们先取出 current->trapframe->regs.sp+24，即用户栈栈底，而后每次以 16 字节累加以此追踪每个函数对应的符号。而后在引入外部变量 elf_loader，即解析后的符号集，并输出每个函数对应的符号。

```
ssize_t print_user_backtrace(int64 depth){
  // uint64 user_sp_top = current->trapframe->regs.sp + 24;
  // // 目前的调用栈深度
  uint64 reg_user_sp = current->trapframe->regs.sp + 16 + 8;
  int64 current_depth = 0;
  for (uint64 cur_p = reg_user_sp; current_depth <= depth; cur_p += 16)
    uint64 ra = *(uint64 *) cur_p;
    if (ra == 0) break;// * 到达用户栈底
    // * 追踪符号
    uint64 tmp = 0;
    int symbol_idx = -1;
    // sprint("ra: %x\n", ra);
    for (int idx = 0; idx < elf_loader.symbol_cnt; ++idx) {
      if (elf_loader.symbols[idx].st_info == STT_FUNC && elf_loader.symb
        tmp = elf_loader.symbols[idx].st_value;
        symbol_idx = idx;
      }
    }
    //sprint("Symbol_idx: %d %x depth=%d\n", symbol_idx, elf_loader.symb
    if (symbol_idx != -1) {
      if (elf_loader.symbols[symbol_idx].st_value >= 0x81000000 )
        sprint("%s\n", &elf_loader.str_table[elf_loader.symbols[symbol_i
        // && elf_loader.symbols[symbol_idx].st_value <= 0x81000000  + 9
      else
        continue;
```

对于 elf 节头的解析，我们在内核每次加载 elf 节头信息时，附带调用我们写的 get_elf_symbol()函数，对于 elf 头进行解析，并将相应信息存放至 elf_loader 中。相应的解析函数如下所示。

```
// Added @lab1_challenge1
elf_status get_elf_symbol(elf_ctx *t){
  // elf头 -> h
  elf_section_header h;
  int strtab_length = 0;
  // load symbol_table & string_table
  for (int i=0, offset_ctx=t->ehdr.shoff; i<t->ehdr.shnum;++i, offset_ct
    // 如果读错直接返回
    if (elf_fpread(t, &h,sizeof(h),offset_ctx)!=sizeof(h)) return EL_EIO

    if (h.sh_type == SHT_SYMTAB){ // 加载.symtab节
      if (elf_fpread(t, &t->symbols, h.sh_size, h.sh_offset) != h.sh_siz
        return EL_EIO;
      // 得到symbol个数
      t->symbol_cnt = h.sh_size / ELF_SYMBOL_SIZE;
    }
    else if (h.sh_type == SHT_STRTAB){
      if (elf_fpread(t, &t->str_table + strtab_length, h.sh_size, h.sh_o
        return EL_EIO;
      strtab_length += h.sh_size;
    }
  }
  return EL_OK;
}
```

为了支撑其所有结构，我们上网搜索了 elf 节头的相关信息，并在 elf.h 中编写了存放节头信息的结构体：

```
// elf节头信息
// section header structure
typedef struct elf_section_header_t {
  uint32 sh_name;      /* Section name */
  uint32 sh_type;      /* Section type */
  uint64 sh_flags;     /* Section flags */
  uint64 sh_addr;      /* Section virtual address at execution
  uint64 sh_offset;    /* Section offset of file */
  uint64 sh_size;      /* Section's size in bytes */
  uint32 sh_link;      /* Section header table link which link
  uint32 sh_info;
  uint64 sh_addralign; /* Section address alignment constraine
  uint64 sh_entsize;   /* Entry size in bytes if section holds
} elf_section_header;
```

1.3 实验调试及心得

本实验调试较为困难，我先是调了很多遍，发现打印出来的结果较为混乱，百思不得其解。之后我打印了每个 elf 的对应信息，最后调整了 elf_loader[symbol_idx].st_value 的取值范围后顺利运行。

实验二 复杂缺页异常

1.1 实验目的

通过修改 PKE 内核（包括 machine 文件夹下的代码），使得对于不同情况的缺页异常进行不同的处理。

1.2 实验内容

程序思路基本同 lab2_3 一致，对给定 n 计算 0 到 n 的和，但要求将每一步递归的结果保存在数组 ans 中。

创建数组时，我们使用了当前的 malloc 函数申请了一个页面（4KB）的大小，对应可以存储的个数上限为 1024。在函数调用时，我们试图计算 1025 求和，首先由于 n 足够大，所以在函数递归执行时会触发用户栈的缺页，你需要对其进行正确处理，确保程序正确运行；其次，1025 在最后一次计算时会访问数组越界地址，由于该处虚拟地址尚未有对应的物理地址映射，因此属于非法地址的访问，这是不被允许的，对于这种缺页异常，应该提示用户并退出程序执行。

这里我们只需要判断缺页异常的原因，从而使用不同的方式进行解决。

```
uint64 stackp = current->trapframe->regs.sp;
if (stval - stackp < 32){
  user_vm_map(current->pagetable, stval / PGSIZE * PGSIZE, PGSIZE
}
else panic("this address is not available!");
```

1.3 实验调试及心得

挑战实验二的难度没有挑战一那么困难，只需要在 strap.c 中进行少量修改即可。

实验三 进程等待和数据段复制

1.1 实验目的

通过修改 PKE 内核和系统调用，为用户程序提供 wait 函数的功能，wait 函数接受一个参数 pid：

- 当 pid 为-1 时，父进程等待任意一个子进程退出即返回子进程的 pid；
- 当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出即返回子进程的 pid；
- 如果 pid 不合法或 pid 大于 0 且 pid 对应的进程不是当前进程的子进程，返回-1。

补充 do_fork 函数，实验 3_1 实现了代码段的复制，你需要继续实现数据段的复制并保证 fork 后父子进程的数据段相互独立。

1.2 实验内容

当一个进程被释放时，如果父进程存在且被阻塞，则需要将其加入就绪队列。

本实验需要特别注意的是编写 sys_user_wait()函数，即按照题设对于 wait 函数进行支持。我们的代码如下：

```
bool get_pid_ready_queue(int pid){
  process *p;
  for ( p = ready_queue_head;p->queue_next != NULL; p=p->queue_next){
    if (p->pid == pid) return TRUE;
  }
  if (p->pid == pid) return TRUE;
  return FALSE;
}

// added @lab3_challenge1
ssize_t sys_user_wait(int pid){
  if (pid == -1){
    // 等待任一子进程退出，进入调度
    current->status = BLOCKED;
    schedule();
  }
  if (pid < NPROC){
    if (get_pid_ready_queue(pid)){
      current->status = BLOCKED;
      schedule();
      return 0;
    }
  }
  return 0;
}
```

其中，我们通过编写 get_pid_ready_queue()函数来查找就绪队列中是否存在进程号为 pid 的目标进程。若是且合法，则将其置为阻塞并进入调度。

对于函数 do_fork()，我们需要增加数据段（DATA_SEGMENT）的支持，代码如下：

```
case DATA_SEGMENT:
  // added @lab3_challenge1
  for (int j = 0; j < parent->mapped_info[i].npages; j++) {
    uint64 pa_of_mapped_va = lookup_pa(parent->pagetable, parent->
    void *new_addr = alloc_page();
    memcpy(new_addr, (void *) pa_of_mapped_va, PGSIZE);
    map_pages(child->pagetable, parent->mapped_info[i].va + j * PG
  }

  // after mapping, register the vm region (do not delete codes be
  child->mapped_info[child->total_mapped_region].va = parent->mapp
  child->mapped_info[child->total_mapped_region].npages =
    parent->mapped_info[i].npages;
  child->mapped_info[child->total_mapped_region].seg_type = DATA_S
  child->total_mapped_region++;
  break;
```

1.3 实验调试及心得

挑战实验三总体上难度不是非常高，只需要小心并注意一些细节即可。刚开始由于我数据段的代码直接复制了代码段的 do_fork()代码，导致里面有几处 CODE_SEGMENT 未改成 DATA_SEGMENT 导致了错误。这些细节需要更加细致。

实验四 相对路径

1.1 实验目的

通过修改 PKE 文件系统代码，提供解析相对路径的支持。具体来说，用户应用程序在使用任何需要传递路径字符串的函数时，都可以传递相对路径而不会影响程序的功能。

完成用户层 pwd 函数（显示进程当前工作目录）、cd 函数（切换进程当前工作目录）。

1.2 实验内容

对内核代码的修改可能包含以下内容：

- 添加系统调用，完成进程 cwd 的读取和修改。
- 修改文件系统对路径字符串的解析过程，使其能够正确处理相对路径。
- 对 RFS 进行适当的修改。

由于我们需要实现的是对于相对路径的支持，而在文件系统中已经有了相关的绝对路径文件的打开和其它操作，因此我们只需要将相对路径转换为绝对路径，再运行之前的代码即可。

具体地，我们需要支持的主要是两个函数，即读当前工作目录，以及写当前工作目录，这里我们分别编写 sys_user_rcwd()函数和 sys_user_ccwd()函数进行实现。

```
// added @lab4_challenge1
ssize_t sys_user_rcwd(uint64 path){
  // 先将用户态va转换为用户态pa
  uint64 pa = (uint64)user_va_to_pa((pagetable_t)(current->pagetable), (void*)path);
  // 将用户态文件名导入物理地址
  memcpy((char*)pa, current->pfiles->cwd->name, sizeof(current->pfiles->cwd->name));
  return 0;
}
```

读工作目录的代码如上。我们通过读当前工作目录，实现写相对路径。相对路径有下列三种形式，即“./”“../”和“/”，分别指代在当前目录下、在上一级目录下和在当前子目录下。这里我们需要对目录字符串进行一些处理，例如找到倒数第二个“/”，实现对于目录的切换。

```
ssize_t sys_user_ccwd(uint64 path){
  // 转化用户态pa
  uint64 pa = (uint64)user_va_to_pa((pagetable_t)(current->pagetable), (void*)path);
  char* dir_data = (char*)pa;
  char dir[MAX_PATH_LEN];
  memset(dir,0x0,MAX_PATH_LEN);
  memcpy(dir, current->pfiles->cwd->name, strlen(current->pfiles->cwd->name));
  // 支持相对路径
  if (dir_data[0]=='.'){
    if (dir_data[1] == '.'){
      // 上级目录 ..
      int path_len = strlen(dir);
      // 回溯至上一级
      for (int i = path_len-1;i>=0;--i){
        if (dir[i] == '/'){
          dir[i] = 0;
          break;
        }
        dir[i] = 0;
      }
      path_len = strlen(dir);
      // 对根目录的情形特殊讨论
      if (path_len == 0){
        // 令 dir = "/"
        dir[1] = 0;
        dir[0] = '/';
```

相似地，我们在用户库函数中更改 open()函数的内容，实现将相对路径转换为绝对路径，使得调用内核时均采用绝对路径的形式。

```
int open(const char *pathname, int flags) {
  // added @lab4_challenge1

  char current_path[MAX_PATH_LEN];
  read_cwd(current_path);
  // 对相对路径实现支持
  char* dir_data = (char*)pathname;
  char dir[MAX_PATH_LEN];
  memset(dir,0x0,MAX_PATH_LEN);
  memcpy(dir, current_path, strlen(current_path));
  // 支持相对路径
  if (dir_data[0]=='.'){
    if (dir_data[1] == '.'){
      // 上级目录 ..
      int path_len = strlen(dir);
      // 回溯至上一级
      for (int i = path_len-1;i>=0;--i){
        if (dir[i] == '/'){
          dir[i] = 0;
          break;
        }
        dir[i] = 0;
      }
```

其中，函数需要调用的 read_cwd()函数，亦即读取当前工作目录，这只需在 syscall 中调用即可。

1.3 实验调试及心得

挑战实验四的核心在于实现对相对路径的支持，只需要将相对路径转化为绝对路径后，支持对当前工作目录的读取即可。