华中科技大学

操作系统原理课程实验报告

姓　　名：　　金钊

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　CS2108

学　　号：　　U202115568

指导教师：　　李晓露

分数	
教师签名	

2024 年 1 月 7 日

目　录

实验一　打印用户程序调用栈 1

1.1　实验目的 1
1.2　实验内容 1
1.3　实验调试及心得 4

实验二　复杂缺页异常 5

1.1　实验目的 5
1.2　实验内容 5
1.3　实验调试及心得 5

实验三　进程等待和数据段复制 6

1.1　实验目的 6
1.2　实验内容 6
1.3　实验调试及心得 8

实验四　相对路径 9

1.1　实验目的 9
1.2　实验内容 9
1.3　实验调试及心得 11

实验一 打印用户程序调用栈

1.1 实验目的

本次实验主要考察了对于 pke 的用户层调用到系统调用的路径了解，对于 elf 文件的格式与解析的掌握，对于 riscv 的函数调用栈和寄存器调用约定的了解。

1.2 实验内容

本次实验主要有两项工作需要完成：

- 了解 elf 文件的.strtab，.shstrtab，.symtab 节的格式和解析方式，获取函数名称。
- 了解 riscv 的函数调用栈格式和寄存器调用约定，定位函数名称。

首先来看 elf 文件各节的内容和解析方式。ELF（Executable and Linkable Format）是一种常见的二进制文件格式，广泛用于 Unix 和类 Unix 系统中。它被用于存储可执行文件、共享库、目标文件和核心转储文件。对于本次实验而言，我们需要解析的是用户程序的可执行文件，打印用户程序调用栈。

根据实验指导的内容，我们在本次实验中需要获取.strtab 和.symtab 的内容。首先查询 elf 文件的格式。根绝查询到的资料可以了解到，我们可以根据 elf 文件头首先获取各个 section header 的名称。Elf 文件的各个节的节头的都是固定大小的。他们的名称都存储在.shstrtab 中，使用节头中存储的偏移量访问。通过其他成员变量，如 sh_addr，sh_offset 等我们可以获取该节的具体信息。

Section header 结构 elf.h

```
typedef struct elf_sect_header_t
{
  uint32    sh_name;        /* 4B section 名称相对于字符串表的位置偏移 */
  uint32    sh_type;        /* 4B section 的类型 */
  uint64    sh_flags;       /* 8B section 的标志 */
  uint64    sh_addr;        /* 8B section 的第一个字节应处的位置 */
  uint64    sh_offset;      /* 8B section 的第一个字节与文件头之间的偏移
*/
  uint64    sh_size;        /* 8B section 的长度（字节数） */
  uint32    sh_link;        /* 4B section 的头部表索引链接 */
  uint32    sh_info;        /* 4B section 的附加信息 */
  uint64    sh_addralign;   /* 8B section 的对齐方式 */
  uint64    sh_entsize;     /* 8B section 的每个表项的长度（字节数） */
```

```
} elf_sect_header;          /* total 64B */
```

我们可以通过匹配.shstrtab 中存储的节头的名称来获取目标节的内容。根据查阅的资料可以了解到.shstrtab 位于 e_shoff+ e_shentsize* e_shstrndx 处。使用 elf_fpread 函数读取节头，然后根据节头的内容就可以获取各个节头的名称信息了。

读取. shstrtab 的内容 elf. c

```
  ...
  // 获取.shstrtab 地址
  uint64 shstr_sh_offset = this_header.shoff + this_header.shentsize
* this_header.shstrndx;
  // 读取.shstr 的 section header
  elf_fpread(ctx, (void *)&shstr_sh, SECTION_HEADER_SIZE,
shstr_sh_offset);
  // 开辟缓冲区，读取内容
  char shstr_sect[shstr_sh.sh_size];
  elf_fpread(ctx,shstr_sect,shstr_sh.sh_size,shstr_sh.sh_offset);
  ...
```

获取节头信息后，我们需要根据 elf 文件头的信息遍历所有节，找到.strtab 和.symtab 节的节头并获取这两个节的信息。其中，读取.symtab 的内容时需要首先了解.symtab 的内容格式，因为该节的内容是格式化存储的。此外，还需要了解根据.symtab 节的内容的含义，筛选出我们需要的函数符号。这里可以参考代码中 elf.h 中定义的结构体 sym_t，这里不再赘述。

获取函数信息 elf. c

```
  ...
  // 读取 symtab 和 strtab
  char sym_sect[sym_sh.sh_size];
  char str_sect[str_sh.sh_size];
  elf_fpread(ctx, sym_sect, sym_sh.sh_size, sym_sh.sh_offset);
  elf_fpread(ctx, str_sect, str_sh.sh_size, str_sh.sh_offset);
  int sym_num = sym_sh.sh_size / SYMBOL_SIZE;
  elf_sym* pfind_func;
  int func_idx = 0;
  // 查询 symtab, 获取函数名称和大小
  for(int i = 0; i < sym_num; i++){
    pfind_func = (elf_sym* )(sym_sect + SYMBOL_SIZE * i);
    if(pfind_func->st_name == 0) continue;
    if(pfind_func->st_info == 18){
      this_func_info[func_idx].st_value = pfind_func->st_value;
      this_func_info[func_idx].st_size = pfind_func->st_size;
      strcpy(this_func_info[func_idx].func_name,(str_sect +
pfind_func->st_name));
```

```
      func_idx++;
    }
  }
  func_num = func_idx;
...
```

按照如上代码所示的方法，获取函数的起始地址和大小信息，根据.strtab 获取函数名称信息。这些信息将会在根据调用栈信息定位函数中使用。

接下来让我们来关注 riscv 的函数调用栈结构和寄存器调用约定，以获取调用函数的位置信息。在 riscv 中，约定了使用 S0/fp 来保存栈帧指针。根据实验指导给出的栈帧结构示意图我们可以了解到，该指针指向了栈帧的底部。对于 645 位 riscv 而言，地址减 8 就指向了返回地址 ra，地址减 16 就指向了该栈帧的 fp。当返回到 main 函数时，返回地址位 0。这样我们就完成了函数调用栈的回溯。

回溯函数调用栈 syscall.c

```
...
ssize_t sys_user_print_backtrace(uint64 depth) {
  uint64 trace_depth = 0;
  uint64* fp_addr = (uint64 *)(current->trapframe->regs.s0 - 8);
  uint64* ra_addr = (uint64 *)(*fp_addr - 8);
  while(*ra_addr != 0 && trace_depth != depth){
      if(!query_func_name(*ra_addr)){
        panic("query function name error");
        return -1;
      }
      fp_addr = (uint64 *)(*fp_addr - 16);
      ra_addr = (uint64 *)(*fp_addr - 8);
      trace_depth++;
  }
  return trace_depth;
}
...
```

最后我们需要根据返回地址获取和前面获取的函数起始地址和大小信息获取函数名称。最初我希望希望根据返回地址获取上一条指令的地址，查看反汇编可以发现，函数调用都是通过 jal 指令完成的。但是由于内存对齐的问题，访问代码段中具体的指令会遇到一些麻烦，我选择改用判断返回地址所在区间判断调用者的函数名。最终实现 query_func_name 函数并在 sys_user_print_backtrace 中进行调用即可实现打印用户调用栈的功能。

此外，本次实验中我们首次添加了库函数，通过添加并调用指定的系统调用完成要求的功能。这要求我们自行添加从用户层到系统调用的接口。就本次实验而言，我们添加了从 user_lib 到 syscall 中各自添加了对应的接口并实现了对用功

能，当用户使用该函数时会自陷到 S 模式完成对应调用。之所以要这样设计，是因为完成实验要求的功能需要的部分信息是对用户态隔离的，需要进入监管模式才能使用一些系统调用，获取指定信息。这样的实现方式在之后实验中也会用到。

1.3 实验调试及心得

本次实验中最大的难点在于查阅分析 elf 文件相关信息。本次实验中我们需要自行查询资料获取 elf 文件头的解析方式，获取指定节，获取指定节的内容并完成解析。由于实验指导中没有直接提供完备的资料，学生需要自行查阅甄别有效的资料信息。这部分需要不断地试错验证完成。在查阅正确资料并正确理解的情况下，elf 文件的解析本身没有什么难点。

在完成实验的过程中，我还出现了一些语法上的错误。最初由于忘记指针的加减法的含义，导致在最初回溯调用栈的过程中我访问了未对齐地址。在纠正了这些错误后，我顺利地完成了调用栈的回溯，并结合解析 elf 文件获取的信息完成了本次实验。

实验二 复杂缺页异常

1.1 实验目的

本实验主要考察了 pke 中对于用户态栈的虚拟地址范围限制，以及操作系统对于堆内存越界访问的处理。

1.2 实验内容

在本次实验中，我们需要对内存分配的复杂缺页异常进行处理。在前面的基础实验中，我们已经完成了简单缺页异常的处理。具体来说，就是在发生缺页异常时，为发生缺页异常的虚拟地址分配页面并添加从虚拟地址到物理地址的映射。在基础实验，我们发生缺页异常的地址都位于用户栈，如果发生缺页中断就满足地址的需求。

在挑战实验中，我们使用 navie_malloc 函数从堆上分配了一个 4kb 大小的页面用于存储数组，并且使用错误的下表进行越界访问。由于是位于堆上的内存，其地址范围显然不处于用户栈的范围内，因此我们可以考虑约定用户栈的大小范文，并判断发生缺页异常的地址是否处于用户栈的范围内（这里对于栈空间的限制为 20 个页的大小），就可以确定是不是访问了未经分配的堆内存。如果在则满足缺页异常，否则不为发生缺页异常的地址分配页面。

缺页异常处理 strap.c

```
if(stval<USER_STACK_TOP && stval>=USER_STACK_TOP-20*PGSIZE){
    uint64 new_pa = (uint64)alloc_page();
    if(new_pa == 0) panic("alloc_page failed.\n");
    map_pages(current->pagetable,ROUNDDOWN(stval,PGSIZE),
    PGSIZE,new_pa,prot_to_type(PROT_READ|PROT_WRITE,1));
  } else {
     panic("this addres is not available!");
   }
```

1.3 实验调试及心得

本次实验内容较为简单，仅就通过评测的目的而言的话，并不需要按照实验指导中完成进程的虚拟地址空间进行监视，直接判断缺页异常是否发生在用户栈空间即可。

实验三 进程等待和数据段复制

1.1 实验目的

本次实验主要考察了父子进程查找，进程的状态转换，进程的唤醒以及进程的数据段复制。

1.2 实验内容

本实验的具体要求如下：

- 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。

根据实验要求我们，我们首先添加 wait 的从用户层到系统调用的接口。这部分内容在实验一中已经介绍过，这里不再详细描述。接着在 process.c 中添加 wait 系统调用的实现 do_wait。该功能的实现在实验要求中已经介绍的十分明确了。接下来主要考虑如何实现在子进程退出后父线程继续运行。

观察用户程序可以发现，当进程退出时，将会调用 exit。查看 exit 系统调用的实现，我们发现它会将当前进程状态设置为 ZOMBIE，然后调用 schedule 转进到其他线程中去。因此我们选择在 exit 中添加唤醒父线程的功能实现。这里我们在 process 结构体中添加一个标志位 uint64 wake_up_parent 表示当前进程退出时是否需要唤醒父线程，并添加 wakeup 函数实现父线程的唤醒，在 exit 中进行调用。具体而言，是将父线程的状态设置为 READY，并插入到就绪队列中即可。在当前线程退出时，会自行调用 schedule 转进到其他线程中去。

修改后的 sys_user_exit `syscall.c`

```
ssize_t sys_user_exit(uint64 code) {
  sprint("User exit with code:%d.\n", code);
  // reclaim the current process, and reschedule. added @lab3_1
```

```
  free_process( current );
  wake_up_parent();
  schedule();
  return 0;
}
```

最后让我们来看 do_wait 的实现。do_wait 的功能在实验任务书中介绍的十分详细了，我们只需要按照实验要求寻找子进程，修改标志位，阻塞父线程并按照要求提供返回值即可。详细实现如下：

do_wait 实现 process.c

```
int do_wait(int pid){
    if( pid == -1){
        for(int i = 0; i < NPROC; i++){
            if(procs[i].parent == current && (procs[i].status ==
READY)){
                current->status = BLOCKED;
                procs[i].wake_up_parent = 1;
                int sub_proc_pid = procs[i].pid;
                schedule();
                return sub_proc_pid;
            }
        }
        if(current->status != BLOCKED) return -1;
    } else {
        if(pid >= 0 && pid < NPROC && procs[pid].parent != current)
{
            return -1;
        } else {
            current->status = BLOCKED;
            procs[pid].wake_up_parent = 1;
            schedule();
            return pid;
        }
    }
    return -1;
}
```

值得一提的是，在一般情况下，进程结构体的 queue_next 指针一般指向处于相同状态的下一个进程。但是在本次实验中，我们并没有对处于阻塞状态的进程进行调度，并没有维护一个阻塞进程队列的必要。为了实验实现方便，我省略了这一功能的实现。就本次实验而言，这个省略并不会影响实验功能的完整性。

完成了 do_wait 的实现后让我们来看数据段的复制。在之前的基础实验中，我们已经完成了进程代码段的复制，和代码段复制不同的是，数据段的复制需要

保证父子进程的数据段互相独立。因此我们需要为数据段的复制重新分配页面，复制数据，然后再将页面映射到虚拟地址上。具体实现如下：

数据段复制实现 process.c

```
case  DATA_SEGMENT:
        for( int j = 0; j < parent->mapped_info[i].npages; j++ ){
          uint64 parent_va = parent->mapped_info[i].va + j * PGSIZE;
          uint64 parent_pa = lookup_pa(parent->pagetable, par-
ent_va);
          uint64 child_va = parent_va;
          void * child_pa = alloc_page();
          memcpy(child_pa, (void *)parent_pa, PGSIZE);
          map_pages(child->pagetable, child_va, PGSIZE,
(uint64)child_pa, prot_to_type(PROT_WRITE | PROT_READ, 1));
        }
        child->mapped_info[child->total_mapped_region].va = par-
ent->mapped_info[i].va;
        child->mapped_info[child->total_mapped_region].npages =
          parent->mapped_info[i].npages;
        child->mapped_info[child->total_mapped_region].seg_type =
DATA_SEGMENT;
        child->total_mapped_region++;
        break;
```

完成以上工作，实验三的挑战实验 1 就完成了。

1.3 实验调试及心得

本次实验让我对于实际进程调度中进程状态的转换和进程调度的时机有了更加深刻的理解。实验内容总体较为简单。在完成实验时，我没有注意实验还要求完成数据段的复制，因此在初次完成提交时发现输出中会出现缺页中断。在重新阅读实验要求后，我添加了数据段复制的实现，顺利完成了本次实验。

实验四 相对路径

1.1 实验目的

本实验主要考察了学生对于 pke 的文件系统架构的了解。在实验中，学生需要根据对进程的文件系统，文件的索引节点等结构的理解和掌握，以及对于字符串处理的掌握，完成相对路径的解析的实现。

1.2 实验内容

本次实验具体要求如下：

- 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）

根据实验的具体要求我们可以发现，完成本次实验的基础是相对路径的解析，其余要求可以在现有代码的基础上通过添加路径解析功能和调用已有文件系统接口实现。因此，首先让我们关注相对路径解析功能的实现。

相对路径解析的实现可以分成以下两个步骤：

1. 获取当前工作目录的完整路径。
2. 解析相对路径并和第一步获得的路径拼接获得绝对路径。

当我们想要获取当前工作目录的完整路径时，我们可以从进程的结构中寻找线索。首先，我们观察结构体 process_t 的定义，可以发现它包含一个类型为 proc_file_management* 的成员变量 pfile，用于管理进程的文件信息。

接着，我们观察 proc_file_management 结构体的定义，可以发现它包含一个成员变量 struct dentry *cwd，该指针指向了当前工作目录的 VFS（Virtual File System）入口。

最后，我们观察 dentry 结构体的定义，可以得知当前目录的名称存储在 char name[MAX_DENTRY_NAME_LEN] 中。要获取当前目录的完整路径，可以通过 dentry 结构体中的 parent 指针向上逐级查询目录。

根据上述分析，我们可以得到以下实现思路：递归查询，直到根目录，并逐层添加当前目录名称，以获取完整的路径。

最终，我们可以实现一个函数来获取当前工作目录的完整路径，具体如下：

获取当前工作目录的完整路径的函数实现 proc_file.c

```
void find_pwd_path(struct dentry* now, char record[]){
  if(now -> parent == NULL){
    record[0] = '/';
    record[1] = '\0';
    return ;
  } else {
    find_pwd_path(now->parent, record);
    strcat(record, now->name);
  }
}
```

获取了当前目录的完整路径后，需要开始解析拼接相对路径。相对路径和绝对路径可以根据路径的第一个字符区别。对于相对路径，使用'/'分割后，如果得到的是'.'则不进行操作；如果是'..'，则从已经解析的目录的尾部删除一级目录，根目录除外；如果是一个目录名或文件名则拼接到已经解析的目录尾部。根据以上思路，可以得到如下实现：

解析相对路径的函数实现 proc_file.c

```
void parse_path(const char* pathname, char* resolved_path){
  if(pathname[0] == '/'){
    strcpy(resolved_path, pathname);
    return ;
  }
  find_pwd_path(current->pfiles->cwd, resolved_path);

  size_t path_lengh = strlen(resolved_path);
  if(resolved_path[path_lengh - 1] != '/'){
    strcat(resolved_path,"/");
  }
  // 复制 pathname
  char path_copy[MAX_PATH_LEN];
  strcpy(path_copy, pathname);
  char* token = strtok(path_copy, "/");
  while(token != NULL){
    if (strcmp(token,".") == 0) {
      goto next_part;
    } else if (strcmp(token,"..") == 0) {
      if(strlen(resolved_path) == 1) goto next_part;
      char* las_seq = strrchr(resolved_path,'/');
      if(las_seq != NULL) *las_seq = '\0';
      las_seq = strrchr(resolved_path,'/');
      if(las_seq != NULL) *las_seq = '\0';
       strcat(resolved_path,"/");
```

```
    } else {
      strcat(resolved_path,token);
      strcat(resolved_path,"/");
    }
    next_part:
    token = strtok(NULL, "/");
  }
  char* las_seq = strrchr(resolved_path,'/');
  if(las_seq) *las_seq = '\0';
}
```

为了方便，我自行添加了 strrchr 函数的实现，这里不再详细介绍。这样我们就完成相对路径的解析工作，可以开始进行实验要求的其他功能的实现了。

要想添加 pke 的文件系统对于相对路径的支持，只需要再 proc_file 层中对于所有的使用了路径的函数中添加相对路径的解析即可。添加如下代码，并在之后的代码中将 pathname 替换成 resolved_path。

添加相对路径解析 proc_file.c

```
  ...
  char resolved_path[MAX_PATH_LEN];
  memset(resolved_path,0,MAX_PATH_LEN);
  parse_path(pathname, resolved_path);
  ...
```

具体而言，需要对 do_open，do_opendir 和 do_mkdir 完成上述修改。

用户层的函数实现在完成以上工作后就很简单了。对于从用户层到系统调用添加对应的接口：read_cwd->do_user_call->do_syscall->sys_user_rswd->do_rcwd。接下来再 do_rcwd 中调用 find_pwd_path 函数即可完成实现。

接下来看 change_cwd 的实现。按照上述调用链添加对应接口后，实现对应函数即可。do_ccwd 也在 proc_file.c 中完成实现。调用 do_opendir 函数打开指定目录并修改当前进程的 cwd，最后调用 do_closedir 关闭目录即可。

1.3 实验调试及心得

本次实验中，我犯得最大的错误是没有在解析相对路径的函数中添加给定路径的拷贝。在完成实验时，我选择 proc_file 中实现对应的功能，前面的接口都是比葫芦画瓢完成的。这导致我忽视了在 syscall 中已经完成了到物理地址的映射。由于在用户程序中路径都是直接使用字符串字面量传递参数，前后两次调用中传递的字符串都是是映射在相同物理地址的常量。如果不使用拷贝进行解析，则会导致常量被修改，进而导致第二次调用时无法正常打开文件。初次遇到这个问题时，我认为我在 proc_file 中的实现出现了错误，花费了大量时间去排查错

误—这也导致了我没有按时提交试验，而是在实验截止后完成的。在确认 proc_file 中的实现没有问题，而是参数传递出现了问题后，我更加迷惑了：为什么传递字面量作为参数还能有区别？难不成是我的环境出现了问题？完全没有想到我的错误实现导致常量被修改。

后来，在和殷梓达同学的交流中我意识到前后两次传递的字面量在编译后使用的都是.rodata 段中同一地址的数据，映射在相同的物理地址上。在 syscall 中已经完成了虚拟地址到物理地址的转换。在 s 模式下，我具备修改整个内存的权限。由于没有将路径拷贝后再解析，导致我直接修改了本应当是的只读的数据。导致第二次传入错误的参数。在添加了路径拷贝后，我的程序顺利通过了测试——尽管是在截至时间后。希望老师助教能高抬贵手给实验四一点实验分。

```
obj/app_relativepath:     file format elf64-littleriscv

Contents of section .rodata:
 109b0 6377643a 25730a00 63642066 61696c65  cwd:%s..cd faile
 109c0 640a0000 00000000 0a3d3d3d 3d3d3d3d  d........=======
 109d0 3d205465 73742031 3a206368 616e6765  = Test 1: change
 109e0 20637572 72656e74 20646972 6563746f   current directo
 109f0 72792020 3d3d3d3d 3d3d3d3d 0a000000  ry  ========....
 10a00 2e2f5241 4d444953 4b300000 00000000  ./RAMDISK0......
 10a10 6368616e 67652063 75727265 6e742064  change current d
 10a20 69726563 746f7279 20746f20 2e2f5241  irectory to ./RA
 10a30 4d444953 4b300a00 0a3d3d3d 3d3d3d3d  MDISK0...=======
 10a40 3d205465 73742032 3a207772 6974652f  = Test 2: write/
 10a50 72656164 2066696c 65206279 2072656c  read file by rel
 10a60 61746976 65207061 74682020 3d3d3d3d  ative path  ====
 10a70 3d3d3d3d 0a000000 77726974 653a202e  ====....write: .
 10a80 2f72616d 66696c65 0a000000 00000000  /ramfile........
 10a90 2e2f7261 6d66696c 65000000 00000000  ./ramfile.......
```

图 4-1 .rodata 段内容(红框标记为作为参数的相对路径)

这次错误也让我深刻认识到我是在完成一个操作系统的编写，我拥有至高的权限，我编写的代码不仅仅会影响实验要求功能的实现，还会决定程序本身是否能被正确运行。因此更应该在程序编写时小心谨慎，避免出现错误。