华中科技大学

操作系统原理课程实验报告

姓　　名：　　王嘉成

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　计算机 2207 班

学　　号：　　U202215576

指导教师：　　谢美意

分数	
教师签名	

2025 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，实现用户程序函数调用栈的回溯功能，具体为是在用户程序调用 print_backtrace()时，能够根据输入参数控制回溯深度，并通过解析用户态栈帧和 ELF 符号表，将函数地址转换为符号名称，从而打印出清晰的调用栈信息。实验旨在深入理解系统调用机制、用户态与内核态的切换、栈帧结构以及符号解析技术，增强对操作系统内核的设计与实现能力。

1.2 实验内容

需要有文字解释设计思路，只能贴关键代码或数据结构！

首先我们查看 user 文件下的实验文件，发现调用了 print_backtrace 函数，而 print_backtrace 函数需要我们来构造，根据实验提示我们可以知道，是要在 syscall.c 文件的 do_syscall 中增加一个 case，所以我们觉得在 use_lib.c 文件中来完成函数调用，然后再在 syscall.c 中补充函数实现细节。

```
int print_backtrace(int depth){
    return do_user_call(SYS_print_backtrace,depth,0,0,0,0,0,0);
}
```

图 1 use_lib.h 文件中的 print_backtrace 函数实现

再根据实验指导，完成实验我们需要使用 elf 文件中的符号表等内容并结合系统调用栈打印被调用函数的名称，所以我们需要了解 ELF 文件中的符号节、以及字符串节的相关知识。

符号节是 ELF 文件中的一个节，存储了程序中定义的所有符号的信息，在本实验中，print_backtrace()需要根据符号地址解析出对应的函数名称和其他符号信息。为了实现这一点，需要访问符号节，找到函数的符号条目，并使用该条目中的信息来打印调用栈。

而字符串节是一个特殊的节，存储了所有字符串数据，包括符号名称、节名称等，符号节中的 st_name 字段只是一个索引，它指向字符串节中的字符串位置。当 print_backtrace()找到一个符号时，它需要通过符号表中的 st_name 字段查找该

符号的名称。而这个名称存储在字符串节中，因此在解析符号表时需要访问字符串节。

因此，我们额外构造了 elf_section_header_t 和 elf_sym 结构体，为了供实验使用。

```
typedef struct elf_section_header_t{
    uint32 sh_name;   /* Section name, index in string tbl */
    uint32 sh_type;   /* Type of section */
    uint64 sh_flags;
    uint64 sh_addr;
    uint64 sh_offset; /* Section file offset */
    uint64 sh_size;   /* Size of section in bytes */
    uint32 sh_link;
    uint32 sh_info;
    uint64 sh_addralign;
    uint64 sh_entsize;  /* Entry size if section holds table */
}elf_section_header;

typedef struct elf_sym{
    uint32 st_name;   /* Symbol name, index in string tbl */
    unsigned char st_info;  /* Type and binding attributes */
    unsigned char st_other; /* No defined meaning, 0 */
    uint16 st_shndx;    /* Associated section index */
    uint64 st_value;    /* Value of the symbol */
    uint64 st_size;   /* Associated symbol size */
} elf_symbol;
```

图 2 elf_section_header_t 和 elf_sym 结构体

可以发现 elf_section_header_t 结构体中的 sh_offset 和 sh_size 字段，这两个字段帮助定位节在文件中的偏移位置和大小，通过它们可以找到符号节和字符串节的位置。elf_symbol 结构体中的 st_name 字段，它保存符号名称在字符串节中的偏移，通过查找字符串节，可以得到符号的实际名称。

了解 ELF 文件格式，我们开始补充 syscall.c 文件中的函数细节，先是在 do_syscall 函数中补充一个 case，调用具体实现函数 sys_print_backtrace，首先获取当前进程的栈帧指针，然后调用 find_func_name 来查找当前栈帧对应的函数名称。若未回溯到 main，它会更新栈帧指针，回溯到上一层栈帧并继续查找，直到达到指定的回溯深度或回溯到 main 函数，最终，回溯过程结束时，返回 0。

而 find_func_name 函数根据给定的栈帧指针 fp 查找当前栈帧所在的函数名称。它通过遍历符号表，检查返回地址是否在某个符号的地址范围内，如果找到了匹配的符号，则打印该符号的名称，并判断是否回溯到 main 函数。若回溯到

main，则停止回溯，返回 1；否则，继续回溯。

```
ssize_t find_func_name(uint64 fp){
    fp = *(uint64*)(fp - 8);
    for(int i = 0;i < sym_count;i++){
        if(fp >= symbols[i].st_value && fp < symbols[i].st_value + symbols[i].st_size)
{
            sprint("%s\n",sym_names[i]);
            if(strcmp(sym_names[i],"main") == 0) return 1;
            else return 0;
        }
    }
    return 0;
}

ssize_t sys_print_backtrace(uint64 n){
    uint64 fp = current->trapframe->regs.s0;
    fp = *((uint64*)(fp - 8));
    for(int i = 0;i < n;i++){
        if(find_func_name(fp)) return 0;
        fp = *((uint64*)(fp - 16));
    }
    return 0;
}
```

图 3 sys_print_backtrace 函数实现

1.3 实验调试及心得

在完成这个实验的过程中，我对操作系统内核的工作原理和符号解析有了更深入的理解，尤其是在栈帧回溯和符号表解析方面。这不仅加深了我对操作系统如何处理函数调用、栈管理的认识，还让我学会了如何通过操作系统内核代码来访问和处理符号表，提取程序中的函数名称及其相关信息。通过设计 print_backtrace 函数，我对如何从用户态栈回溯到内核态栈，并提取相应的符号信息有了更全面的理解。

实验二 复杂缺页异常

2.1 实验目的

本实验的目的是通过修改 PKE 内核代码，实现针对不同情况的缺页异常处理。具体为修改进程的数据结构，以便能够监控和管理虚拟地址空间，同时修改 kernel/strap.c 中的异常处理函数，对缺页异常做出不同的处理。对于合理的缺页异常，需要扩大内核栈的大小并为其映射新的物理块；而对于非法地址的缺页异常，则需要报错并终止程序。此外，实验还要求根据不同的错误情况生成适当的路径信息：若是用户源程序发生错误，路径应为相对路径；若是标准库内的错误，路径应为绝对路径。

2.2 实验内容

根据 lab2_3，正常的缺页异常发生在 USER_STACK_TOP 后续的用户栈区域，此时需要分配新的页面供程序使用，而程序中通过 malloc()分配的内存是从 g_ufree_page 开始的。这一分配过程通过系统调用最终调用了 syscall.c 中的 sys_user_allocate_page 函数，该函数从 g_ufree_page 部分开始分配内存。

在处理逻辑中，当缺页异常的原因是 CAUSE_STORE_PAGE_FAULT 时，代码动态增加用户栈的大小，首先判断触发缺页的虚拟地址 stval 是否小于当前栈指针向下扩展一个页面的范围，如果 stval 超出了合理范围，则认为访问了非法地址，直接调用 panic 函数终止程序。否则，分配一个新的物理页面，并通过 map_pages 函数将该页面映射到引发缺页异常的虚拟地址对应的页表中。这样，程序可以在用户态正常访问这个新分配的栈空间。

```
void handle_user_page_fault(uint64 mcause, uint64 sepc, uint64 stval) {
  sprint("handle_page_fault: %lx\n", stval);
  switch (mcause) {
    case CAUSE_STORE_PAGE_FAULT:
      // TODO (lab2_3): implement the operations that solve the page fault to
      // dynamically increase application stack.
      // hint: first allocate a new physical page, and then, maps the new page to the
      // virtual address that causes the page fault.
      if(stval < current->trapframe->regs.sp-PGSIZE) panic("this address is not
available!");
      map_pages(
        current->pagetable,ROUNDDOWN(stval,PGSIZE),PGSIZE,(uint64)alloc_page(),
prot_to_type(PROT_READ|PROT_WRITE,1));

      break;
    default:
      sprint("unknown page fault.\n");
      break;
  }
}
```

图 4 challenge2_1 代码内容

2.3 实验调试及心得

通过完成这个实验，我对操作系统中缺页异常的处理机制有了更深入的理解。实验涉及的动态内存管理和栈空间扩展，强化了我对虚拟内存管理的概念，尤其是页表映射、物理页面分配以及异常地址判断的具体实现，在解决缺页异常时，我学会了如何结合地址范围和异常类型判断合法性，并采取相应的处理策略，这对于保障程序的稳定性和安全性至关重要。

实验三 进程等待和数据段复制

3.1 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 函数使父进程能够等待子进程的退出，并根据传入的 pid 参数进行不同的处理：当 pid 为-1 时，等待任意子进程退出；当 pid 大于 0 时，等待指定 pid 的子进程退出；若 pid 不合法或指定的子进程不属于当前父进程，则返回-1。此外，还需要补充 do_fork 函数，完成数据段的复制，确保父子进程的数据段独立。

3.2 实验内容

打开 user/app.wait.c 文件，顺着追踪 fork 函数可以发现最终到 do_fork 函数，对该函数进行补充，这段代码在 do_fork 函数中用于实现父子进程数据段的复制与映射，其核心是确保子进程的数据段与父进程相互独立，同时保持父进程的数据正确性。

代码通过遍历父进程数据段的每一页，利用 lookup_pa 函数获取父进程页表中虚拟地址对应的物理地址，为子进程分配新的物理页面并使用 memcpy 将父进程页面的数据复制到新页面。接着，调用 map_pages 函数在子进程页表中建立虚拟地址与新物理页面的映射，设置相应的读写权限。复制完成后，将父进程的数据段信息注册到子进程的 mapped_info 中，更新子进程的虚拟内存区域记录，确保内存分配和映射的同步。

```
case DATA_SEGMENT:
  for( int j=0; j<parent->mapped_info[i].npages; j++ ){
    uint64 addr = lookup_pa(parent->pagetable,
      parent->mapped_info[i].va+j*PGSIZE);
    char *newaddr = alloc_page(); memcpy(newaddr, (void *)addr, PGSIZE);
    map_pages(child->pagetable, parent->mapped_info[i].va+j*PGSIZE, PGSIZE,
      (uint64)newaddr, prot_to_type(PROT_WRITE | PROT_READ, 1));
  }

  // after mapping, register the vm region (do not delete codes below!)
  child->mapped_info[child->total_mapped_region].va = parent->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;
```

图 5 do_fork 函数补充

存分配和映射的同步。完成 do_fork 函数后我们要在 syscall.c 完成对 wait() 函数的系统调用，但最终 do_wait 还是写在 process.c 函数里。do_wai 函数为用户程序提供了 wait 功能，允许父进程根据指定的 pid 等待子进程退出，并返回子进程的 pid。

当 pid 为-1 时，函数会检查当前进程的所有子进程。若找到至少一个子进程，将继续判断这些子进程的状态：如果有子进程处于僵尸状态，释放其资源，并返回其 pid；若没有子进程处于僵尸状态，说明尚无子进程退出，当前进程需要挂起，然后调用 schedule 函数切换到其他可运行的进程，返回-2 表示进入阻塞状态。若当前进程没有任何子进程，直接返回-1 表示错误。

当 pid 为合法的具体值时，函数首先检查指定的 pid 是否属于当前进程的子进程。若不是，直接返回-1 表示错误；若是，进一步检查该子进程的状态：若状态为僵尸，释放其资源并返回该 pid；若状态不是僵尸，则将当前进程加入阻塞队列并切换调度，同样返回-2 表示进入阻塞状态。

若 pid 大于进程数上限 NPROC，直接返回-1 表示参数错误。

```
int do_wait(int pid)
{
  int found = 0;
  if (pid == -1) {
    for (int i = 0; i < NPROC; i++)
      if (procs[i].parent == current) {
        found = 1;
          if (procs[i].status == ZOMBIE) {
          procs[i].status = FREE;
          return i;
          }
      }
    if (found == 0) return -1;
    else {
      insert_to_blocked_queue(current);
      schedule();
      return -2;
    }
  }
  else if (pid < NPROC) {
    if (procs[pid].parent != current) return -1;
    else {
      if (procs[pid].status == ZOMBIE) {
        procs[pid].status = FREE;
        return pid;
      }
      else {
        insert_to_blocked_queue(current);
        schedule();
        return -2;
    }
    }
  }
  else return -1;
}
```

图 6 do_wait 函数实现

完成等待的系统调度指令后，最后要完成退出的系统调度指令，在 syscall.c 文件中用 sys_user_exit 函数实现，释放空间并更新状态，将目前进程移出等待序列。

```
ssize_t sys_user_exit(uint64 code) {
  sprint("User exit with code:%d.\n", code);
  free_process( current );
  remove_and_insert(current);
  schedule();
  return 0;
}
```

图 7 sys_user_exit 函数实现

3.3 实验调试及心得

通过本实验，我深入理解了操作系统中进程退出与资源回收的机制。实现 sys_user_exit 让我掌握了进程资源释放和状态更新的关键步骤，wait 函数的开发让我理解了父子进程协作与阻塞同步机制的重要性，同时，通过调度器的调用，我认识到多任务系统中进程切换的核心作用。实验让我强化了对内核代码的理解，提高了对资源管理和状态维护的实践能力，也进一步体会到操作系统设计的严谨性与复杂性。

实验四 相对路径

4.1 实验目的

本实验的目的是通过修改 PKE 内核中的文件系统代码，实现对相对路径的支持。具体要求包括：在用户程序中使用相对路径时，系统能够正确解析路径并执行相关操作；完善 pwd 和 cd 功能，使用户可以查看和修改当前工作目录。同时需要在内核中增加系统调用，用于读取和修改进程的当前工作目录，并优化文件系统路径解析逻辑，确保能够正确处理相对路径。此外，还需对 RFS 进行适当调整以支持这些功能，从而实现文件系统的更灵活使用。

4.2 实验内容

从 user/app_relativepath.c 文件中发现要构造 read_cwd 和 change_cwd 函数，实现以上函数都要在 syscall.c 中文件构造更详细的系统调用函数，在 do_syscall 函数中新增两个 case，分布对应 SYS_user_rcwd 和 SYS_user_ccwd。

在 sys_user_rcwd 中，首先初始化一个缓冲区 path 并清零，然后检查当前工作目录名称是否是绝对路径，如果不是，就在路径开头添加斜杠/，接着，将当前工作目录名称拼接到路径缓冲区中，并将其通过用户虚拟地址到物理地址的映射写入用户提供的路径地址空间。

sys_user_ccwd 用于切换当前工作目录。实验 lab2_1 中我们完成了 user_va_to_pa 函数，而 sys_user_ccwd 函数通过 user_va_to_pa 函数将用户虚拟地址转换为物理地址，并处理多种路径格式。如果路径以"./"开头，表示进入当前目录的子目录，程序会截取路径的后续部分并通过 hash_get_dentry 找到目标目录。如果路径以".."开头，表示切换到父目录，并根据后续路径处理具体子目录，如果路径以"/"开头，表示绝对路径，否则为相对路径。无论哪种情况，程序都会最终调用 hash_get_dentry 解析路径并更新当前目录为解析结果。

```c
ssize_t sys_user_rcwd(char* pathva){
    char path[30];
    memset(path,0,sizeof(path));
    if(current->pfiles->cwd->name[0] != '/') path[0] = '/';
    strcat(path,current->pfiles->cwd->name);
    strcpy(user_va_to_pa(current->pagetable,(void*)pathva),path);
    return 0;
}

ssize_t sys_user_ccwd(char* pathva){
    char * pathpa = (char*)user_va_to_pa((pagetable_t)(current->pagetable), pathva);
    char path[30];
    if(pathpa[0] == '.' && pathpa[1] == '/'){
        strcpy(path,pathpa + 2);
        current->pfiles->cwd = hash_get_dentry(current->pfiles->cwd,path);
    }else if(pathpa[0] == '.' && pathpa[1] == '.'){
        current->pfiles->cwd = current->pfiles->cwd->parent;
        if(pathpa[2] == '/'){
            strcpy(path,pathpa + 3);
            current->pfiles->cwd = hash_get_dentry(current->pfiles->cwd,path);
        }
    }else{
        if(pathpa[0] == '/')
            strcpy(path,pathpa + 1);
        else
            strcpy(path,pathpa);
        current->pfiles->cwd = hash_get_dentry(current->pfiles->cwd,path);
    }
    return 0;
}
```

图 8 sys_user_rcwd 和 sys_user_ccwd 函数实现

4.3 实验调试及心得

本实验让我深入理解了文件系统中路径解析和工作目录管理的机制，掌握了相对路径与绝对路径的解析方法。同时，通过实现 pwd 和 cd 功能，熟悉了系统调用与内核数据结构的交互。