华中科技大学

操作系统原理课程设计报告

姓　　名：　　王怡彬

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　计卓 2001 班

学　　号：　　U201914858

指导教师：　　邵志远

分数	
教师签名	

2023 年 3 月 25 日

目　录

实验一 打印异常代码行

1.1 实验目的

通过修改 PKE 内核，使得用户程序在发生异常时，内核能够输出触发异常的代码所在的文件路径，行号与代码本身。

1.2 实验内容

1. 修改 elf_load 函数

对 elf.c 文件中的 elf_load 函数进行修改，添加存储程序段的最大虚拟地址的变量 max_vaddr，查找 debug_line 段，将其数据保存并调用构造表函数。函数读取存储段名称的段头，然后遍历所有的段，这里可以直接通过比较段名称字符串来查找 debug_line 段。如果找到 debug_line 段，则读取其数据并将其放置在最大虚拟地址处，并调用 make_addr_line 函数，分析 debug_line 段中的数据，存储代码文件的目录路径、代码文件名和关联指令地址与代码行号及代码文件的映射关系。

```
char name[16];
((elf_info *)ctx->info)->p->debugline = NULL;
elf_sect_header name_seg, tmp_seg;

if (elf_fpread(ctx, (void *)&name_seg, sizeof(name_seg),
               ctx->ehdr.shoff + ctx->ehdr.shstrndx * sizeof(name_seg)) != sizeof(name_seg))
    return EL_EIO;

for (i = 0; i < ctx->ehdr.shnum; i++)
{
    off = ctx->ehdr.shoff;
    if (elf_fpread(ctx, (void *)&tmp_seg, sizeof(tmp_seg), off) == sizeof(tmp_seg))
        elf_fpread(ctx, (void *)name, 20, name_seg.offset + tmp_seg.name);
    else
        return EL_EIO;
    if (strcmp(name, ".debug_line") != 0)
        off += sizeof(tmp_seg);
    else
    {
        if (elf_fpread(ctx, (void *)max_vaddr, tmp_seg.size, tmp_seg.offset) == tmp_seg.size)
        {
            make_addr_line(ctx, (char *)max_vaddr, tmp_seg.size);
            break;
        }
        else
            return EL_EIO;
    }
}
return EL_OK;
```

图 1.1 elf_load 函数关键代码

2. 打印错误信息

实现用于打印运行错误的函数并在中断中进行调用以打印异常代码行。

定义 print_error() 函数，用于在发生异常时打印出错误信息。从程序的代码行信息中查找到当前异常发生的位置，然后打印出对应位置的源代码行以及相应的错误信息。为实现该目的，先通过给定地址行中的文件信息和目录信息构建出源文件的路径，并使用库函数打开源文件，读取其中的内容，然后遍历每一行的内容，直到找到指定行的内容，最后通过 sprint() 函数打印出包含错误信息的字符串。

```
void print_error()
{
    uint64 mepc = read_csr(mepc);
    for (int i = 0; i < current->line_ind; i++)
    {
        if (mepc < current->line[i].addr)
        {
            addr_line *errorline = current->line + i - 1;
            int dir_len = strlen(current->dir[current->file[errorline->file].dir]);
            strcpy(path, current->dir[current->file[errorline->file].dir]);
            path[dir_len] = '/';
            strcpy(path + dir_len + 1, current->file[errorline->file].file);
            spike_file_t *file = spike_file_open(path, O_RDONLY, 0);
            spike_file_stat(file, &mystat);
            spike_file_read(file, code, mystat.st_size);
            spike_file_close(file);

            int offset = 0, count = 0;
            while (offset < mystat.st_size)
            {
                int x = offset;
                while (x < mystat.st_size && code[x] != '\n')
                    x++;
                if (count == errorline->line - 1)
                {
                    code[x] = '\0';
                    sprint("Runtime error at %s:%d\n%s\n", path, errorline->line, code + offset);
                    break;
                }
                else
                {
                    offset = x + 1;
                    count++;
                }
            }
            break;
        }
```

图 1.2 print_error 函数代码

1.3 实验调试及心得

在本次实验的初期，我在最大虚拟地址的处理与计算方面遇到了许多困难，无法正确读取 debug_line 的数据，在经历了大量的调试修改后最终使其正确运行。

本次实验加深了我对操作系统运行方式的理解，同时让我学习到了处理异常情况在系统内核中的流程，也进一步提升了我对各种调试手段在内核态中的使用的理解。

实验二 实现信号量

2.1 实验目的

设计系统调用以实现信号量的初始化，分配与 PV 操作，以在内核态使进程并行化；通过修改 PKE 内核，定义信号量的数据结构，设计对应的方法函数。

2.2 实验内容

1. 信号量数据结构

在 process.h 文件中定义信号量的数据结构，整形变量 value，state 分别表示为信号量的值以及该信号量是否被使用，process 指针 p_queue 指向等待进程的队列。同时定义最大进程数上限的常量以限制进程数。

```
typedef struct {
  int value;
  int state;
  process *wait_queue;
} semaphore;
```

图 2.1 信号量数据结构定义

2. 信号量方法函数

分别定义并实现信号量的分配与 PV 操作的各个函数。P_sem 函数将信号量 n 减一。如果信号量的值小于 0（没有可用的资源），则将当前进程阻塞，加入等待队列中，然后调用调度函数 schedule() 进行进程调度。若当前进程已在等待队列中，则直接返回。V_sem 函数将信号量 n 加一。若等待队列不为空，则移出等待队列中的一个进程并将其加入到就绪队列。

```c
int new_sem(int n){
  for (int i=NSEMAPHONE-1; i>=0; i--1){
    if(sems[i].state == 0){
      sems[i].value = n;
      sems[i].state = 1;
      return i;
    }
  }
  return -1;
}


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

图 2.2 信号量关键函数代码

3. 系统调用

与之前的其他实验类似，先把 process.c 中的函数在 syscall.c 中添加对应系统调用函数，最后在 user_lib.c 中将其调用，以实现用户在 app_semaphore.c 中对 sem_P，sem_V 和 sem_new 函数的调用。

2.3 实验调试及心得

这项实验较为简单，信号量的关键操作函数也是操作系统课程的重点内容，在进程等待队列的设计与实现上遇到了一些 bug，但通过不断调试和修改最终得到了解决。本次实验进一步加深了我对系统内核中数据结构使用的理解。

实验三 相对路径

3.1 实验目的

通过修改 PKE 文件系统代码，提供相对路径解析的支持，从而使用户应用程序能够在不影响其功能的情况下传递相对路径。实现用户层的 rcwd 函数，以显示进程的当前工作目录，并实现 ccwd 函数以更改进程的当前工作目录。

3.2 实验内容

修改文件系统中的路径解析函数，以处理相对路径。在 sys_user_open 函数中，调用解析函数 get_absolute_path，以将传递的相对路径转换为绝对路径。在该函数中，遍历当前进程的工作目录，并将相对路径附加到其末尾，以获得绝对路径。然后，将这个新的绝对路径传递给 do_open 函数进行处理。

```
ssize_t sys_user_open(char *pathva, int flags) {
  char* pathpa = (char*)user_va_to_pa((pagetable_t)(current->pagetable), pathva);

  if (pathpa[0] == '.') {
    char absolute_path[MAX_DENTRY_NAME_LEN];
    memset(absolute_path, '\0', MAX_DENTRY_NAME_LEN);
    get_absolute_path(pathpa, absolute_path);
    return do_open(absolute_path, flags);
  }
  return do_open(pathpa, flags);
}
```

图 3.1 sys_user_open 函数代码

实现 rcwd 函数，使用类似的方法遍历当前进程的工作目录，并将每个目录的名称添加到一个字符串中。最后，将该字符串输出，以显示当前进程的工作目录。

实现 ccwd 函数，使用类似方法，先调用 get_absolute_path 函数获取新的绝对路径，然后通过 do_opendir 函数打开该目录并更新当前进程的工作目录，最后再调用 do_closedir 函数关闭目录，即可将当前进程的工作目录更改为新的目录，并在之后的操作中进行使用。

```c
void sys_user_rcwd(char* pathva) {
  char *pathpa = (char*)user_va_to_pa((pagetable_t)(current->pagetable), (void*)
  pathva);
  struct dentry* p = current->pfiles->cwd;
  if (p->parent == NULL) {
    strcpy(pathpa, "/");
  } else {
    while (p) {
      char t[MAX_DENTRY_NAME_LEN];
      memset(t, '\0', MAX_DENTRY_NAME_LEN);
      memcpy(t, pathpa, strlen(pathpa));
      memset(pathpa, '\0', MAX_DENTRY_NAME_LEN);
      memcpy(pathpa, p->name, strlen(p->name));
      if (p->parent != NULL) {
        pathpa[strlen(p->name)] = '/';
        pathpa[strlen(p->name) + 1] = '\0';
      }
      strcat(pathpa, t);
      p = p->parent;
    }
    pathpa[strlen(pathpa) - 1] = '\0';
  }
}

void sys_user_ccwd(char* pathva) {
  char* pathpa = (char*)user_va_to_pa((pagetable_t)(current->pagetable), pathva);
  char absolute_path[MAX_DEVICE_NAME_LEN];
  memset(absolute_path, '\0', MAX_DENTRY_NAME_LEN);
  get_absolute_path(pathpa, absolute_path);

  int fd = do_opendir(absolute_path);
  current->pfiles->cwd = current->pfiles->opened_files[fd].f_dentry;
  do_closedir(fd);
}
```

图 3.2　rcwd 与 ccwd 函数代码

随后在 syscall 函数中添加对应的系统调用即可，此处不再赘述。

3.3 实验调试及心得

本次实验的内容相较于前两个实验，涉及的细节和代码量都多了很多，尤其在获取文件绝对路径函数的实现上一度困扰了相当多的事件，最后通过请教同学和自己的反复调试修改正确实现了功能，通过了测试。

在本次实验中，我加深了对操作系统的基础知识和概念的认识，通过自己亲手编写代码，我深刻理解了操作系统内核中文件路径信息的传递，提高了自己的编程能力和代码调试能力，为今后从事系统开发和底层编程奠定了坚实的基础。

实验四 重载执行

4.1 实验目的

通过对 PKE 内核和系统调用的修改，实现 exec 函数。使用该函数时，提供可执行程序的路径名作为参数，运行该路径下所存储的程序。

4.2 实验内容

exec 函数需要加载一个 ELF 格式的应用程序到当前进程的地址空间中，并且在加载之前，还需清除当前进程的代码、数据和栈段，并将新的代码、数据和栈段映射到当前进程的地址空间。具体来说，首先需要将用户态地址转换为内核态地址，再清除当前进程中的代码、数据和栈段，并将这些段从虚拟地址空间中解除映射；然后分配一个用户栈，并将其映射到用户态地址空间；读取 ELF 格式文件的头信息和程序头信息，并将程序段加载到进程的地址空间中；最后将新加载的代码、数据和栈段映射到当前进程的地址空间中并将当前进程的返回地址设置为应用程序入口地址。加载段信息的对应代码可以直接在 elf.c 文件中找到，此处直接挪用了其部分代码。

```
ssize_t sys_user_exec(char* vfn){
  // fetch filename
  char * pfn = (char*)user_va_to_pa((pagetable_t)(current->pagetable), (void*)vfn);
  sprint("Application: %s\n",pfn);
  char exec_name[100]="hostfs_root";
  strcpy(exec_name+strlen(exec_name),pfn);


  sprint("va\t\t\tnpages\tseg_type\tpa\n");
  for(int i=0;i<current->total_mapped_region;i++)
  {
    sprint("%llx\t%d\t%d\t%llx\n",current->mapped_info[i].va,
                                  current->mapped_info[i].npages,
                                  current->mapped_info[i].seg_type,
                                  (uint64)user_va_to_pa(current->pagetable,(void*)
                                  current->mapped_info[i].va));
  }
  // free CODE_SEGMENT,DATA_SEGMENT,STACK_SEGMENT of old process
  int cnt=0;
  for(int i=0;i<current->total_mapped_region;i++)
  {
    if(current->mapped_info[i].seg_type==CODE_SEGMENT ||
       current->mapped_info[i].seg_type==DATA_SEGMENT ||
       current->mapped_info[i].seg_type==STACK_SEGMENT)
    {
      void* pa=user_va_to_pa(current->pagetable,
                             (void*)current->mapped_info[i].va);
      user_vm_unmap(current->pagetable,current->mapped_info[i].va,PGSIZE,1);
      current->mapped_info[i].npages=0;
      current->mapped_info[i].va=0;
      cnt++;
    }
  }
  current->total_mapped_region-=cnt;
```

图 4.1　exec 函数代码

随后在 syscall 函数中添加对应的系统调用即可，此处不再赘述。

4.3 实验调试及心得

开始时我对如何达成实验目的一头雾水，感到十分困难。不过，在和其他同学交流和观察系统初次运行加载可执行程序的过程后，我学会了仿照系统的加载过程，利用系统内核自带加载 elf 文件的代码来实现目标函数，在经过反复调试后最终通过了实验。

通过这次实验，我深入理解了系统如何加载可执行程序的过程，同时也提高了自己的编程能力和代码调试能力。