课程名称：操作系统实践	年级：2023级	上机实践成绩：
指导教师：张民	姓名：李彤	
上机实践名称：系统调用	学号：10235101500	上机实践日期：2024.12.30
上机实践编号：	组号：	上机实践时间：

在这次实验中，我们需要完成剩下的系统调用函数，以通过80个测试用例

代码实现

添加结构体定义

在thread.h文件中，我们主要添加了以下内容：

- 在 thread 中增加 child 数据结构，用来保存子线程的相关信息；
- 在 thread 结构体中增加子线程的参数。

```
struct thread {
  // ...

  struct list childs;                 // 子线程列表
  struct child * thread_child;        // 存储当前线程的子线程
  struct semaphore sema;              // 控制子进程的逻辑，父进程等待子进程结束
  bool success;                       // 判断子线程是否执行成功

  // ...
};

// 父进程的子进程，父进程调用 fork 时创建
struct child
{
  tid_t tid;                          // 线程的 tid
  bool isrun;                         // 子线程是否成功运行
  struct list_elem child_elem;        // 子进程列表中的元素
  struct semaphore sema;              // 用于控制等待的信号量
  int store_exit;                     // 子线程的退出状态
};
```

完成这些定义之后，再通过 **thread_creat** 函数将其初始化。

```
tid_t thread_create (const char *name, int priority, thread_func *function, void
*aux)
{
  // 初始化线程的子进程
  t->thread_child = malloc(sizeof(struct child));     // 为子进程分配内存
  t->thread_child->tid = tid;                         // 设置子进程的线程ID
  sema_init(&t->thread_child->sema, 0);               // 初始化信号量，值为0，用于同
步等待
```

```
  list_push_back(&thread_current()->childs, &t->thread_child->child_elem);  // 将子进程添加到当前线程的子进程列表中
  t->thread_child->store_exit = UINT32_MAX;          // 初始化退出状态为最大值，表示尚未退出
  t->thread_child->isrun = false;                    // 子进程未运行，默认为 false
}
```

辅助函数回顾

在上一次的实验中，我定义了一些辅助函数，这里再展示一遍其用途，后面还要使用：

- exit_special —— 异常退出函数，退出当前线程，并设置退出状态为 -1；
- check_ptr —— 检查用户传入的地址是否有效；
- is_valid_pointer —— 检查给定的栈指针及参数是否有效，确保它们指向用户空间且在有效的页面中；
- find_file_id —— 根据文件描述符（file_id）查找当前线程打开的文件；
- get_user —— 从用户空间地址读取一个字节，并返回其值；
- acquire_lock_f —— 给文件操作加锁；
- release_lock_f —— 释放文件操作锁。

系统调用处理函数

syscall_init 函数将所有系统调用的中断处理程序都注册到中断向量表。

而 **syscall_handler** 直接根据系统调用的类型（即系统调用编号）来选择并执行具体的系统调用处理函数。

```
void syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");

  // 关于进程的系统调用
  syscalls[SYS_HALT] = &sys_halt;
  syscalls[SYS_EXIT] = &sys_exit;
  syscalls[SYS_EXEC] = &sys_exec;
  syscalls[SYS_WAIT] = &sys_wait;

  // 关于文件的系统调用
  syscalls[SYS_CREATE] = &sys_create;
  syscalls[SYS_REMOVE] = &sys_remove;
  syscalls[SYS_OPEN] = &sys_open;
  syscalls[SYS_WRITE] = &sys_write;
  syscalls[SYS_SEEK] = &sys_seek;
  syscalls[SYS_TELL] = &sys_tell;
  syscalls[SYS_CLOSE] =&sys_close;
  syscalls[SYS_READ] = &sys_read;
  syscalls[SYS_FILESIZE] = &sys_filesize;
```

```
}

static void syscall_handler (struct intr_frame *f UNUSED)
{
  int * p = f->esp;                // 获取栈指针指向的位置（即系统调用参数的地址）
  check_ptr (p + 1);               // 检查传入的参数指针是否有效，避免访问非法内存
  int type = * (int *)f->esp;      // 从栈中读取系统调用的类型（即系统调用的编号）

  if(type <= 0 || type >= max_syscall)  // max_syscall设置为20
    exit_special ();

  // if(type == SYS_EXIT || type == SYS_WRITE)
  syscalls[type](f);                // 调用对应的处理函数
}
```

系统调用的实现

与进程相关的系统调用

- ~~sys_exit —— 退出当前进程，并将进程的退出状态保存到当前线程中。~~
- sys_exec —— 执行一个新的程序，并将返回值存入 eax。
- sys_halt —— 关机系统。
- sys_wait —— 等待子进程的结束，并将返回值存入 eax。

```
// ...

void sys_halt (struct intr_frame* f)
{
  // 调用关机函数，关闭系统电源
  // 该函数在 devices/shutdown.c 函数中定义
  shutdown_power_off();
}


void sys_exec (struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr (user_ptr + 1);
  check_ptr (*(user_ptr + 1));  // 检查传递的程序名指针是否有效
  *user_ptr++;

  // 调用 exec 函数，执行指定程序，并将返回值存入 eax
  f->eax = process_execute((char*)* user_ptr);
}


void sys_wait (struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
```

```
  check_ptr (user_ptr + 1);
  *user_ptr++;

  // 调用 wait 函数，等待子进程结束，并将返回值存入 eax
  f->eax = process_wait(*user_ptr);
}

// ...
```

需要注意的是，在project1我们提到了进程等待 **process_wait** 是一个空函数，而这里我们的sys_wait又是通过调用process_wait函数实现的，所以我们还需要将process_wait函数给实现：

```
// ...

// 等待指定的子进程（由 child_tid 标识）退出，并返回其退出状态。
// 如果未找到子进程，或子进程已经退出，返回 -1。
int process_wait (tid_t child_tid UNUSED)
{
  struct list *l = &thread_current()->childs;  // 获取当前线程的子进程列表
  struct list_elem *temp = list_begin (l);  // 获取子进程列表的第一个元素
  struct child *temp2 = NULL;

  // 遍历子进程列表，查找匹配的子进程
  while (temp != list_end (l))
  {
    temp2 = list_entry (temp, struct child, child_elem);  // 获取当前子进程
    if (temp2->tid == child_tid)  // 找到目标子进程
    {
      if (!temp2->isrun)
      {
        temp2->isrun = true;         // 标记子进程已开始运行
        sema_down (&temp2->sema);    // 等待子进程退出（通过信号量同步）
        break;
      }
      else
        return -1;  // 如果子进程已运行，返回 -1（表示无法等待）
    }
    temp = list_next (temp);  // 移动到下一个子进程
  }

  // 如果遍历到列表末尾仍未找到子进程，返回 -1
  if (temp == list_end (l))
    return -1;

  list_remove (temp);          // 移除子进程，表示该进程已完成
  return temp2->store_exit;    // 返回子进程的退出状态
}

// ...
```

与文件相关的系统调用

这些系统调用的基本思路都是先上锁，执行对应文件操作（调用自带函数）之后再释放锁。

- ~~sys_write —— 向文件或标准输出写入数据；~~
- sys_create —— 创建指定路径的文件；
- sys_remove —— 删除指定路径的文件；
- sys_open —— 打开指定路径的文件并返回文件描述符；
- sys_tell —— 获取文件指针当前位置；
- sys_seek —— 设置文件指针的位置；
- sys_close —— 关闭文件并释放相关资源。
- sys_filesize —— 获取文件的大小。
- sys_read —— 从文件或标准输入读取数据。

```
// ...

void sys_create(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 5);
  check_ptr(*(user_ptr + 4));
  *user_ptr++;

  acquire_lock_f();  // 获取文件系统锁，确保文件操作的同步
  f->eax = filesys_create((const char *)*user_ptr, *(user_ptr+1));  // 调用文件系统
创建文件函数
  release_lock_f();  // 释放文件系统锁
}


void sys_remove(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 1);
  check_ptr(*(user_ptr + 1));    // 检查文件路径指针是否合法
  *user_ptr++;

  acquire_lock_f();
  f->eax = filesys_remove((const char *)*user_ptr);  // 调用文件系统删除文件
  release_lock_f();
}


void sys_open(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 1);
  check_ptr(*(user_ptr + 1));
```

```
  *user_ptr++;

  acquire_lock_f();
  struct file *file_opened = filesys_open((const char *)*user_ptr);  // 打开文件
  release_lock_f();

  struct thread *t = thread_current();  // 获取当前线程

  if (file_opened) {
    struct thread_file *thread_file_temp = malloc(sizeof(struct thread_file));  //
为文件分配内存
    thread_file_temp->fd = t->file_fd++;                          // 分配新的文件描
述符
    thread_file_temp->file = file_opened;                         // 保存文件指针
    list_push_back(&t->files, &thread_file_temp->file_elem);      // 将文件信息添加
到线程文件列表
    f->eax = thread_file_temp->fd;                                // 返回文件描述符
  }
  else
    f->eax = -1;  // 文件打开失败，返回-1
}


void sys_tell(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 1);
  *user_ptr++;

  struct thread_file *thread_file_temp = find_file_id(*user_ptr);

  if (thread_file_temp) {
    acquire_lock_f();
    f->eax = file_tell(thread_file_temp->file);  // 获取文件指针的位置
    release_lock_f();
  }

  else
    f->eax = -1;  // 文件描述符无效，返回-1
}


void sys_seek(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 5);
  *user_ptr++;

  struct thread_file *file_temp = find_file_id(*user_ptr);
  if (file_temp) {
    acquire_lock_f();
    file_seek(file_temp->file, *(user_ptr + 1));  // 设置文件指针
    release_lock_f();
  }
```

```
}


void sys_close(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 1);
  *user_ptr++;

  struct thread_file *opened_file = find_file_id(*user_ptr);
  if (opened_file) {
    acquire_lock_f();
    file_close(opened_file->file);  // 关闭文件
    release_lock_f();
    list_remove(&opened_file->file_elem);  // 从文件列表中移除
    free(opened_file);                     // 释放文件信息结构体
  }
}


void sys_filesize(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 1);
  *user_ptr++;

  struct thread_file *thread_file_temp = find_file_id(*user_ptr);

  if (thread_file_temp) {
    acquire_lock_f();
    f->eax = file_length(thread_file_temp->file);  // 获取文件大小
    release_lock_f();
  }

  else
    f->eax = -1;
}


void sys_read(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  *user_ptr++;

  int fd = *user_ptr;                                // 获取文件描述符
  uint8_t *buffer = (uint8_t*)*(user_ptr + 1);    // 获取数据缓冲区
  off_t size = *(user_ptr + 2);                      // 获取读取数据的大小

  // 检查缓冲区是否有效，如果无效，终止进程
  if (!is_valid_pointer(buffer, 1) || !is_valid_pointer(buffer + size, 1))
    exit_special();

  if (fd == 0) {  // 如果是标准输入 (fd == 0)，使用 input_getc 读取
    for (int i = 0; i < size; i++)
```

```
      buffer[i] = input_getc();

    f->eax = size;  // 返回读取的字节数
  }

  else {
    struct thread_file *thread_file_temp = find_file_id(*user_ptr);
    if (thread_file_temp) {
      acquire_lock_f();
      f->eax = file_read(thread_file_temp->file, buffer, size);
      release_lock_f();
    }

    else
      f->eax = -1;
  }
}

// ...
```

收尾工作

完成这些系统调用即相关辅助函数的实现后，再将其声明添加到文件头中，并 **#inlcude** 所需文件

```
// for example in thread.c

// ...

#include "devices/shutdown.h"
#include "devices/input.h"
#include "threads/malloc.h"
#include "threads/palloc.h"

// ...

void sys_halt(struct intr_frame* f);
void sys_exit(struct intr_frame* f);
void sys_exec(struct intr_frame* f);
void sys_wait(struct intr_frame* f);

void sys_create(struct intr_frame* f);
void sys_remove(struct intr_frame* f);
void sys_open(struct intr_frame* f);
void sys_filesize(struct intr_frame* f);
void sys_read(struct intr_frame* f);
void sys_write(struct intr_frame* f);
void sys_seek(struct intr_frame* f);
void sys_tell(struct intr_frame* f);
void sys_close(struct intr_frame* f);
```

```
// ...
```

测试

先后执行make、make check 指令，我们可以看到80个测试用例全部通过了

```
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 80 tests passed.
make[1]: Leaving directory '/home/PKUOS/pintos/src/userprog/build'
root@eecbc1a9fe41:~/pintos/src/userprog#
```

心得体会

通过本次实验，我更加熟悉了线程管理、系统调用、文件操作等操作系统的基础知识，并且提高了我编程和调试的能力。