实验报告：系统调用 2

课程名称：操作系统	**年级：**2023 级	**上机实践成绩：**
指导教师：张民	**姓名：**张建夫	
上机实践名称：系统调用 2	**学号：** 10235101477	**上机实践日期：** 2024/12/30
上机实践编号：	**组号：**	**上机实践时间：** 14:50~16:30 点

一、目的

实现用户程序和 OS 之间的系统调用，搞明白系统调用在 pintos 里具体是如何实现的。（本次实验使用 cs162，已和张民老师和周孜为助教沟通过）

二、系统调用的分析过程（此处重点分析要求的那四个文件系统调用，其他的系统调用分析放在实现里面了）

Create：

create

```
bool create (const char *file, unsigned initial_size)
```

Creates a new file called `file` initially `initial_size` bytes in size. Returns true if successful, false otherwise. Creating a new file does not open it: opening the new file is a separate operation which would require an `open` system call.

Cs162 文档要求 create 系统调用创建一个大小为 initial_size 的文件，返回值是是否创建成功，并且不允许打开新创建的文件（open 系统调用做这件事）。

而在系统调用的前半部分，文档还有这两段话：

In addition to the process control syscalls, you will also need to implement the following file operation syscalls: `create`, `remove`, `open`, `filesize`, `read`, `write`, `seek`, `tell`, and `close`. Pintos already contains a basic file system which implements a lot of these functionalities, meaning you will not need to implement any file operations yourself. Your implementations will simply call the appropriate functions in the file system library.

Implementation details

Pintos' file system is not thread-safe, so you must make sure that your file operation syscalls do not call multiple file system functions concurrently. In Project File Systems, you will add more sophisticated synchronization to the Pintos file system, but for this project, you are permitted to use a global lock on file operation syscalls (i.e. treat it as a single critical section to ensure thread safety). **We recommend that you avoid modifying the `filesys/` directory in this project.**

第一段话说明 pintos 已经有对应的函数来处理主要的部分（即我们只要处理和内核线程相关的部分就行）

第二段说明我们要处理多个线程访问同一个文件的情况（文档里也说明了实现方法，加一个全局锁即可）

这两段话对接下来的三个系统调用的实现都是需要注意的。

因此，create 的系统调用还是比较简单的，在检查了用户传入的参数并加锁后，直接调用 filesys_create 函数即可

open：

open

```
int open (const char *file)
```

Opens the file named `file`. Returns a nonnegative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened.

File descriptors numbered 0 and 1 are reserved for the console: 0 (`STDIN_FILENO`) is standard input and 1 (`STDOUT_FILENO`) is standard output. `open` should never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

Each process has an independent set of file descriptors. File descriptors in Pintos are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to `close` and they do **not** share a file position.

Open 函数返回一个文件描述符，该文件描述符不能是特殊的文件描述符（0，1，2）

对于多个线程（文档上写的是进程，但实际上 pintos 中的进程使用线程模拟的），每个线程维护自己的文件列表，自己打开文件要自己关闭，互相之间不干扰，也允许一个线程打开多次同一个文件。

根据以上要求，我们需要为每个线程创建一个自己的文件列表，在 open 的时候将文件加入到列表，close 的时候弹出列表。

close：

close

```
void close (int fd)
```

Closes file descriptor `fd`. Exiting or terminating a process must implicitly close all its open file descriptors, as if by calling this function for each one. If the operation is unsuccessful, it can either `exit` with `-1` or it can just fail silently.

关闭文件，还提醒了 thread_exit 函数里面应该关闭所有打开的文件，对于因任何原因打开失败的，可以不做任何处理。

根据要求，只要文件列表维护好了，该函数将文件从列表中删除即可

Read：

read

```
int read (int fd, void *buffer, unsigned size)
```

Reads `size` bytes from the file open as `fd` into `buffer`. Returns the number of bytes actually read (0 at end of file), or -1 if the file could not be read (due to a condition other than end of file, such as `fd` not corresponding to an entry in the file descriptor table). `STDIN_FILENO` reads from the keyboard using the `input_getc` function in `devices/input.c`.

该系统调用要求从一个文件读取内容，包括从标准输入读取（文档中已提示使用 input_getc 函数），要求对任何读不了的情况进行返回 0 或-1 处理（例如文件未打开）。

根据上述要求，只要用户传参合法，文件打开并位于线程的文件列表中就执行 read 操作，实际上只要有了文件列表就很好实现。

三、使用环境

1. 主机 os：Windows-11
2. 虚拟机 os:Linux
3. 虚拟环境：docker 的 linux 镜像
4. 代码编辑器:vscode

四、实验过程

在实现系统调用之前，我们先要实现用户所传参数的合法性检查（合法性检查包括用户传递参数的位置（也就是 f->esp 的值的检查），以及用户传递参数自身的检查（如果用户传的是一个字符串，要逐字节检查字符串指针的合法性）），为了方便所有系统调用进行检查，我们将这部分功能封装成一个函数 check_ptr：

```
bool check_ptr(uint32_t* unchecked)
{
  for(int i=0;i<4;i++)
  {
    if(!is_user_vaddr(unchecked+i)||!pagedir_get_page(thread_current()->pcb->pagedir,unchecked+i))
    return false;
  }
  return true;
}
```

该函数检查传入的指针是否合法，指针本身是否有分配页面，这里 i<4 是因为 32 位机器内指针的长度等于 4 字节

而对于用户给的字符串指针，我们另外写一个函数查看字符串指针指向的位置是否合法：

```
bool check_string(const char*being_checked)
{
  if(being_checked==NULL||being_checked>PHYS_BASE)return false;
  int i=0;
  if(!is_user_vaddr(&being_checked[i])||!pagedir_get_page(thread_current()->pcb->pagedir,&being_checked[i]))
  return false;
  for(;being_checked[i]!='\0';i++)
  {
    if(!is_user_vaddr(&being_checked[i+1])||!pagedir_get_page(thread_current()->pcb->pagedir,&being_checked[i+1]))
    return false;
  }
  return true;
}
```

同样是逐字节检查其合法性，直到结束符'\0'

接着，由于各个系统调用要对不合法的指针进行处理，一般是终止进程，而终止进程需要提示（用例会对打印的提示信息进行检查），我们也将这部分封装成一个函数 sys_exit：

```
void sys_exit(int exit_status)
{
  printf("%s: exit(%d)\n", thread_current()->pcb->process_name, exit_status);
  thread_current()->exit_status=exit_status;
  process_exit();
}
```

需要注意的是，cs162 与学校的退出方式略有不同，cs162 为了更好地模拟进程，编写了 process_exit 函数，该函数内部调用了 thread_exit，本质上是一样的。

接着，我们在系统调用主函数的前面加上检查参数和系统调用号的部分：

```
  if(!check_ptr((uint32_t*)f->esp))
  {
    sys_exit(-1);
  }

  uint32_t*args = ((uint32_t*)f->esp);

  if(args[0]<=0||args[0]>SYS_INUMBER)
  {
    printf('不合法系统调用，硬件问题？');
    sys_exit(-1);
  }
```

只要 f->esp 通过以上测试，就证明该系统调用目前是合法的。

接着，我们要创建一个全局锁，用来处理多线程操作同一文件的同步：

在 thread.h 中建立 static 变量：

```
/*文件系统调用的锁*/
static struct lock f_lock;
```

任何锁都要在使用前 init，这里我们在 pintos 主线程的线程初始化函数里 init 这个锁（在这里 init 的原因是这里定义了所有的线程相关的全局变量，如实验 1 的就绪队列和所有线程的队列 r）：

```c
void thread_init(void) {
  ASSERT(intr_get_level() == INTR_OFF);

  lock_init(&tid_lock);
  list_init(&ready_list);
  list_init(&all_list);
  lock_init(&f_lock);

  /* Set up a thread structure for the running thread. *
  initial_thread = running_thread();
  init_thread(initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid();
}
```

最后，为每一个线程创建属于自己的文件列表，因而我们要在 thread.h 和 thread.c 中进行修改（list 的使用不再赘述，list.h 里写的很清楚，甚至给了示例）：

Thread.h：

在 tcb 块中加上下面这两个

```c
int cur_file_fd;                    /*下一个使用的文件描述符*/
struct list open_files;             /*所有打开的文件*/
```

Thread.c 中的 thread_create 函数：

```c
198 tid_t thread_create(const char* name, int priority, thread_func* function, void* aux) {
199   struct thread* t;
200   struct kernel_thread_frame* kf;
201   struct switch_entry_frame* ef;
202   struct switch_threads_frame* sf;
203   tid_t tid;
204
205   ASSERT(function != NULL);
206
207   /* Allocate thread. */
208   t = palloc_get_page(PAL_ZERO);
209   if (t == NULL)
210     return TID_ERROR;
211
212   /* Initialize thread. */
213   init_thread(t, name, priority);
214   tid = t->tid = allocate_tid();
215
216   /*初始化可能正在等待的锁*/
217   t->lock=NULL;
218
219   /*初始化文件链表*/
220   list_init(&t->open_files);
221   /*初始化文件描述符*/
222   t->cur_file_fd=3;
223
224   /*初始化父进程与子进程通信的信号量*/
225   sema_init(&t->wait_for_child,0);
226
227   /*初始化是否等待过*/
228   t->waited=false;
```

初始化管理文件的相关字段，此处设为 3 是因为 0：标准输入；1：标准输出；2：标准错误，因此从 3 开始

接下来正式处理系统调用：

Create 系统调用：

```
if(args[0]==SYS_CREATE)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2])||!check_string(args[1]))
  {
    sys_exit(-1);
    return;
  }
  char*file=args[1];
  int initial_size=args[2];
  lock_acquire(&f_lock);
  f->eax=filesys_create(file,initial_size);
  lock_release(&f_lock);
}
```

得到用户参数并加锁后直接调用 filesys_create 函数即可，比较简单

Open 系统调用：

```
if(args[0]==SYS_OPEN)
{
  if(!check_ptr(&args[1])||!check_string(args[1]))
  {
    sys_exit(-1);
    return;
  }
  char*file=args[1];
  struct thread*cur=thread_current();

  struct thread_file* tmp=malloc(sizeof(struct thread_file));
  tmp->fd=cur->cur_file_fd++;
  strlcpy(tmp->name,file,sizeof(tmp->name));
  lock_acquire(&f_lock);
  tmp->f=filesys_open(file);
  lock_release(&f_lock);
  if(tmp->f==NULL)
  {
    f->eax=-1;
    free(tmp);//必须释放资源
    return;
  }
  list_push_back(&cur->open_files,&tmp->elem_tf);
  f->eax=tmp->fd;
}
```

这就与用到了先前加入 tcb 块中的 open_files 和 cur_file_fd，我们没有办法得知打开了哪些文件，而有了打开的文件列表，就可以将新打开的文件加入列表（list_push_back 语句）

需要注意的是文件结构体是位于堆上的，如果打开失败要释放内存。

Close 系统调用：

```
if(args[0]==SYS_CLOSE)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    return;
  }
  lock_acquire(&f_lock);
  file_close(tf->f);
  lock_release(&f_lock);
  list_remove(&tf->elem_tf);
  free(tf);
}
```

关闭文件，需要注意的点是要在关闭文件的同时将文件从列表中移除（list_remove 语句），同时释放文件结构体（free 语句）

Find_file 函数实现如下（是为找到句柄对应的文件结构体）：

```
struct thread_file*find_file(int fd)
{
  struct thread*cur=thread_current();
  struct list_elem*e;
  for(e=list_begin(&cur->open_files);e!=list_end(&cur->open_files);e=list_next(e))
  {
    struct thread_file*tmp=list_entry(e,struct thread_file,elem_tf);
    if(tmp->fd==fd)
    return tmp;
  }
  return NULL;
}
```

Remove 系统调用：

```
if(args[0]==SYS_REMOVE)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  char*file=args[1];
  if(!check_string(file))
  {
    f->eax= false;
    return;
  }
  lock_acquire(&f_lock);
  bool success=filesys_remove(file);
  lock_release(&f_lock);
  f->eax=success;
}
```

该系统调用与 create 一样，参数合法并加锁后调用相应函数即可

Filesize 系统调用：

```
if(args[0]==SYS_FILESIZE)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    f->eax=-1;
    return;
  }
  lock_acquire(&f_lock);
  f->eax=file_length(tf->f);
  lock_release(&f_lock);
}
```

返回文件长度，找到文件并调用 file_length 函数即可

Read 系统调用：

```
if(args[0]==SYS_READ)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2])||!check_ptr(&args[3]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  char*buffer=args[2];
  int size=args[3];

  if(!check_string(buffer))
  {
    sys_exit(-1);
    return;
  }

  if(fd==STDIN_FILENO)
  {
    int total=0;
    for(int i=0;i<size;i++)
    {
      buffer[i]=input_getc();
      total++;
    }
    f->eax=total;
    return;
  }
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    f->eax=-1;
    return;
  }
  lock_acquire(&f_lock);
  f->eax=file_read(tf->f,buffer,size);
  lock_release(&f_lock);
}
```

这里要处理输入是标准输入的情况（STDIN_FILENO 部分），对于标准输入，使用 input_getc 函数，对于文件输入，使用 file_read 函数。

Write 系统调用：

```
if(args[0]==SYS_WRITE)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2])||!check_ptr(&args[3]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  char*buffer=args[2];
  int size=args[3];

  if(!check_string(buffer))
  {
    sys_exit(-1);
    return;
  }
  if(fd==STDOUT_FILENO)
  {
    putbuf(buffer,size);
    f->eax=size;
    return;
  }
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    f->eax=-1;
    return;
  }
  if(is_executing(tf->name))
  {
    f->eax=0;
    return;
  }
  lock_acquire(&f_lock);
  f->eax=file_write(tf->f,buffer,size);
  lock_release(&f_lock);
}
```

该系统调用要处理标准输出（STDOUT_FILENO 部分），同时由于如果文件正在被执行，是不能进行写操作的，因此在函数的后部有一个 is_executing 函数判断该文件是否正在被执行，该函数实现如下：

```
/*判断一个线程是否正在被执行*/
bool is_executing(const char*name)
{
  struct list_elem*e;
  for(e=list_begin(&all_list);e!=list_end(&all_list);e=list_next(e))
  {
    struct thread*tmp=list_entry(e,struct thread,allelem);
    if(strcmp(tmp->name,name)==0)return true;
  }
  return false;
}
```

（实际上就是看 all_list（该列表包含所有正在运行的线程，pintos 自带）里面有没有这个文件的名字）

Tell 系统调用：

```
if(args[0]==SYS_TELL)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  struct thread_file*tf=find_file(fd);
  if(tf!=NULL)
  {
    lock_acquire(&f_lock);
    f->eax=file_tell(tf->f);
    lock_release(&f_lock);
  }
}
```

返回文件指针的位置，检查参数无误后调用 file_tell 函数即可

Seek 系统调用：

```
if(args[0]==SYS_SEEK)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  unsigned pos=args[2];
  struct thread_file*tf=find_file(fd);
  if(tf!=NULL)
  {
    lock_acquire(&f_lock);
    file_seek(tf->f,pos);
    lock_release(&f_lock);
  }
}
```

改变文件指针的位置，同样检查参数无误后调用 file_seek 函数即可

Halt 系统调用：

```
if(args[0]==SYS_HALT)
{
  shutdown_power_off();
}
```

最简单的系统调用，根据文档调用该函数即可

Exec 系统调用：

```
if(args[0]==SYS_EXEC)
{
  if(!check_ptr(&args[1])||!check_string(args[1]))
  {
    sys_exit(-1);
  }
  f->eax=process_execute(args[1]);
}
```

检查完用户的参数后调用 process_execute 函数即可（该函数的实现已在上一次传参实验中说明）

Wait 系统调用：

```
if(args[0]==SYS_WAIT)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
  }
  int ch_pid=args[1];
  f->eax=process_wait(ch_pid);
}
```

主要是 process_wait 函数，该函数等待特定的子进程完成，通过信号量告诉父进程自身结束，并返回子进程的退出状态。

下面是 process_wait 函数的具体实现：

```
int process_wait(pid_t child_pid) {
  /*得到父进程*/
  struct thread*cur=thread_current();

  struct thread *cp=get_child_process(child_pid,cur);

  /*找不到该子进程*/
  if(!cp)return TID_ERROR;

  /*子进程已被等待过*/
  if(cp->waited)return TID_ERROR;

  /*标记子进程已被等待*/
  cp->waited=true;

  /*等待子进程结束*/
  sema_down(&cur->wait_for_child);

  /*移除子进程*/
  list_remove(&cp->elem_process);

  /*保存进程退出状态*/
  int es=cp->exit_status;

  /*杀死子进程*/
  if(cp!=initial_thread)
  palloc_free_page(cp);

  return es;
}
```

Get_child_process 函数用于得到该线程需要等待的子进程，函数截图如下：

```
/*根据子进程的tid返回子进程的指针，如果没找到则返回NULL*/
struct thread *get_child_process(pid_t child_tid,struct thread*father)
{
  struct list_elem*e;
  for(e=list_begin(father->child_process);e!=list_end(father->child_process);e=list_next(e))
  {
    struct thread*cp=list_entry(e,struct thread,elem_process);
    if(cp->tid==child_tid)
    {
      return cp;
    }
  }
  return NULL;
}
```

父进程开始等待

返回进程退出状态

通过截图：代码库：https://github.com/saydontgo/cs162-group

```
workspace@69d2794469c5 [07:02:40] userprog $ make check
cd build && make check
make[1]: Entering directory '/home/workspace/code/group/src/userprog/build'
pass tests/userprog/do-nothing
pass tests/userprog/stack-align-0
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
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/iloveos
pass tests/userprog/practice
pass tests/userprog/stack-align-1
pass tests/userprog/stack-align-2
pass tests/userprog/stack-align-3
pass tests/userprog/stack-align-4
pass tests/userprog/floating-point
pass tests/userprog/fp-simul
pass tests/userprog/fp-asm
pass tests/userprog/fp-syscall
pass tests/userprog/fp-kernel-e
pass tests/userprog/fp-init
pass tests/userprog/kernel/fp-kasm
pass tests/userprog/kernel/fp-kinit
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 96 tests passed.
make[1]: Leaving directory '/home/workspace/code/group/src/userprog/build'
```

五、总结

这个 lab 特别有含金量，做的过程是极其痛苦的（有时候一个测试做一天都过不了，但是觉得自己是对的），但是当测试都通过的那一瞬间又有发自内心的畅快，这么说呢，独立做完这个 lab 之后的那一周感觉自己可以手撕操作系统，很多具体的实现等到自己上手才知道理论很简单，实现很困难，要考虑的情况也相当多。总体来讲，做完这个收获还是很多的，最主要的就是加深了对操作系统的理解，而且是非常深刻的理解，能将每个调度细节和系统调用给你讲明白的那种理解。