User Programs

- If you need more infomation about this part.Click **this**

1.System Calls

Implement the system call handler in userprog/syscall.c. The skeleton implementation we provide "handles" system calls by terminating the process. It will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

Implement the following system calls. The prototypes listed are those seen by a user program that includes lib/user/syscall.h. (This header, and all others in lib/user, are for use by user programs only.) System call numbers for each system call are defined in lib/syscall-nr.h:

- System Call: void **halt** (void)

Terminates Pintos by calling shutdown_power_off() (declared in threads/init.h). This should be seldom used, because you lose some information about possible deadlock situations, etc.

- System Call: void **exit** (int status)

Terminates the current user program, returning status to the kernel. If the process's parent waits for it (see below), this is the status that will be returned. Conventionally, a status of 0 indicates success and nonzero values indicate errors.

- System Call: pid_t **exec** (const char *cmd_line)

Runs the executable whose name is given in cmd_line, passing any given arguments, and returns the new process's program id (pid). Must return pid -1, which otherwise should not be a valid pid, if the program cannot load or run for any reason. Thus, the parent process cannot return from the exec until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this.

- System Call: int **wait** (pid_t pid)

Waits for a child process pid and retrieves the child's exit status. If pid is still alive, waits until it terminates. Then, returns the status that pid passed to exit. If pid did not call exit(), but was terminated by the kernel (e.g. killed due to an exception), wait(pid) must return -1. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent calls wait, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

wait must fail and return -1 immediately if any of the following conditions is true:

pid does not refer to a direct child of the calling process. pid is a direct child of the calling process if and only if the calling process received pid as a return value from a successful call to exec.

Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to wait(C) by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.

The process that calls wait has already called wait on pid. That is, a process may wait for any given child at most once.

Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its struct thread, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling process_wait() (in userprog/process.c) from main() (in threads/init.c). We suggest that you implement process_wait() according to the comment at the top of the function and then implement the wait system call in terms of process_wait().

Implementing this system call requires considerably more work than any of the rest.

- System Call: 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 a open system call.

- System Call: bool **remove** (const char *file)

Deletes the file called file. Returns true if successful, false otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it. See Removing an Open File, for details.

- System Call: int **open** (const char *file)

Opens the file called 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: fd 0 (STDIN_FILENO) is standard input, fd 1 (STDOUT_FILENO) is standard output. The open system call will 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 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.

- System Call: int **filesize** (int fd)

Returns the size, in bytes, of the file open as fd.

- System Call: 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). Fd 0 reads from the keyboard using input_getc().

- System Call: int **write** (int fd, const void *buffer, unsigned size)

Writes size bytes from buffer to the open file fd. Returns the number of bytes actually written, which may be less than size if some bytes could not be written. Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or 0 if no bytes could be written at all.

Fd 1 writes to the console. Your code to write to the console should write all of buffer in one call to putbuf(), at least as long as size is not bigger than a few hundred bytes. (It is reasonable to break up larger buffers.) Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our grading scripts.

- System Call: void **seek** (int fd, unsigned position)

Changes the next byte to be read or written in open file fd to position, expressed in bytes from the beginning of the file. (Thus, a position of 0 is the file's start.) A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until project 4 is complete, so writes past end of file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

- System Call: unsigned **tell** (int fd)

Returns the position of the next byte to be read or written in open file fd, expressed in bytes from the beginning of the file.

- System Call: void **close** (int fd)

Closes file descriptor fd. Exiting or terminating a process implicitly closes all its open file descriptors, as if by calling this function for each one.

The file defines other syscalls. Ignore them for now. You will implement some of them in project 3 and the rest in project 4, so be sure to design your system with extensibility in mind.

2.System Call Details

The first project already dealt with one way that the operating system can regain control from a user program: interrupts from timers and I/O devices. These are "external" interrupts, because they are caused by entities outside the CPU.

The operating system also deals with software exceptions, which are events that occur in program code. These can be errors such as a page fault or division by zero. Exceptions are also the means by which a user program can request services ("system calls") from the operating system.

In the 80x86 architecture, the int instruction is the most commonly used means for invoking system calls. This instruction is handled in the same way as other software exceptions. In Pintos, user programs invoke int $0x30 to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt.

Thus, when the system call handler syscall_handler() gets control, the system call number is in the 32-bit word at the caller's stack pointer, the first argument is in the 32-bit word at the next higher address, and so on. The caller's stack pointer is accessible to syscall_handler() as the esp member of the struct intr_frame passed to it. (struct intr_frame is on the kernel stack.)

The 80x86 convention for function return values is to place them in the EAX register. System calls that return a value can do so by modifying the eax member of struct intr_frame.

You should try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or a pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call's arguments from the stack.

3. Assignment

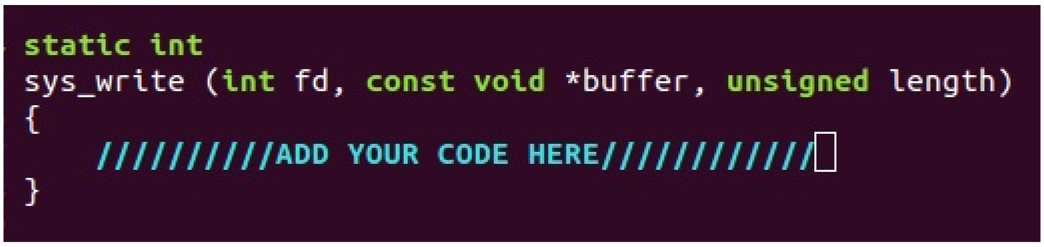
3.1 Descprition

We have alreay written most of the code for you. You need to download the new SRC code to replace the code in your computer.

Your task is to fill the empty function in **syscall.c** such as **void halt ()**.

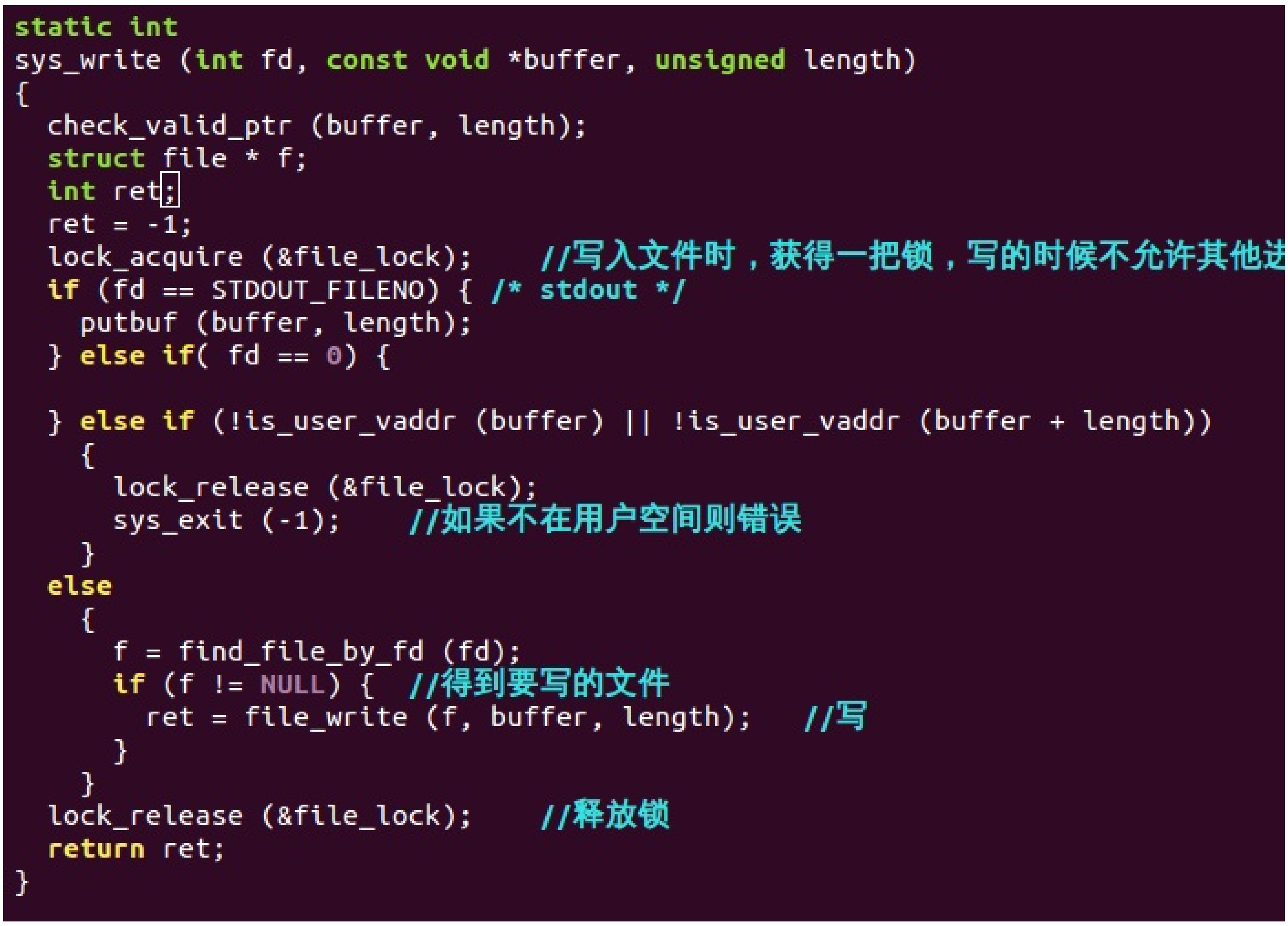
There is a example to help you to understand.

Before you fill empty function:



```
static int
sys_write (int fd, const void *buffer, unsigned length)
{
    //////////ADD YOUR CODE HERE////////////
}
```

After you do that:



```
static int
sys_write (int fd, const void *buffer, unsigned length)
{
  check_valid_ptr (buffer, length);
  struct file * f;
  int ret;
  ret = -1;
  lock_acquire (&file_lock);    //写入文件时，获得一把锁，写的时候不允许其他进
  if (fd == STDOUT_FILENO) { /* stdout */
    putbuf (buffer, length);
  } else if( fd == 0) {

  } else if (!is_user_vaddr (buffer) || !is_user_vaddr (buffer + length))
    {
      lock_release (&file_lock);
      sys_exit (-1);    //如果不在用户空间则错误
    }
  else
    {
      f = find_file_by_fd (fd);
      if (f != NULL) {  //得到要写的文件
        ret = file_write (f, buffer, length);   //写
      }
    }
  lock_release (&file_lock);    //释放锁
  return ret;
}
```

4.2 Requirement

After you have finished this assignment, you should pass the tests in **src/userprog/** as much as possible.

Run `make check` in ***src/userprog/*** and take a screenshots of your test result.

You should download and **fill** the **DESIGNDOC** and archive your code(**the whole pintos folder**) and DESIGNDOC into a **zip** file **named sid_nameInPinYin_vid.zip(e.g 12330441_zhangsan_v0.zip)** and upload to the ftp.

4. Some function you will use.

- bool filesys_create (const char *, unsigned);

Creat a file and return true if successful, false otherwise.

- bool filesys_remove (const char *);

Remove a file and return true if successful, false otherwise.

- void shutdown_power_off();

This function will terminate Pintos.

- struct file * filesys_remove (const char *);

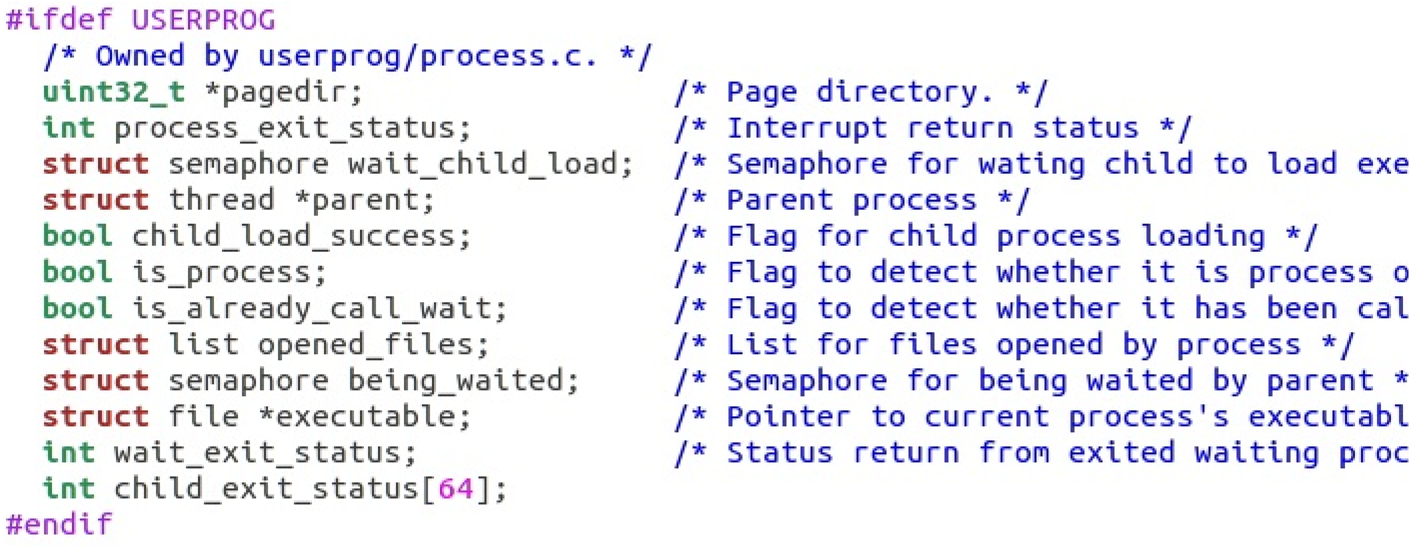
Open a file and return the pointer point to the file.

- int file_length(struct file *);

This function will return file size.

You need to read ***/src/filesys/file.h*** and ***/src/filesys/filesys.h*** to get more information by yourself.

5. New elements in thread



```
#ifdef USERPROG
  /* Owned by userprog/process.c. */
  uint32_t *pagedir;                   /* Page directory. */
  int process_exit_status;             /* Interrupt return status */
  struct semaphore wait_child_load;    /* Semaphore for wating child to load exe
  struct thread *parent;               /* Parent process */
  bool child_load_success;             /* Flag for child process loading */
  bool is_process;                     /* Flag to detect whether it is process o
  bool is_already_call_wait;           /* Flag to detect whether it has been cal
  struct list opened_files;            /* List for files opened by process */
  struct semaphore being_waited;       /* Semaphore for being waited by parent *
  struct file *executable;             /* Pointer to current process's executabl
  int wait_exit_status;                /* Status return from exited waiting proc
  int child_exit_status[64];
#endif
```

In this homework, maybe you only need to use:

- int process_exit_status;
- struct list opened_files;
- bool child_load_success;