Project-2 Review

(Extending the functionality of PintOS user program implementation)

Summer 2018

Overview

- Necessities and Organization
- Argument passing in PintOS
- Implementation of System Calls

Metedata

- Sub-checkpoint 1
  - Implement Argument Passing
- Sub-checkpoint 2
  - Implement System Calls

Only after you implement sub-checkpoints 1 and 2 properly, your tests would begin to pass

Overview

- Necessities and Organization
- Argument passing in PintOS
- Implementation of System Calls

Necessities and Organization

[Recall Project1 Approach]

Step 1: Familiarize with the tests associated with the component to implement.

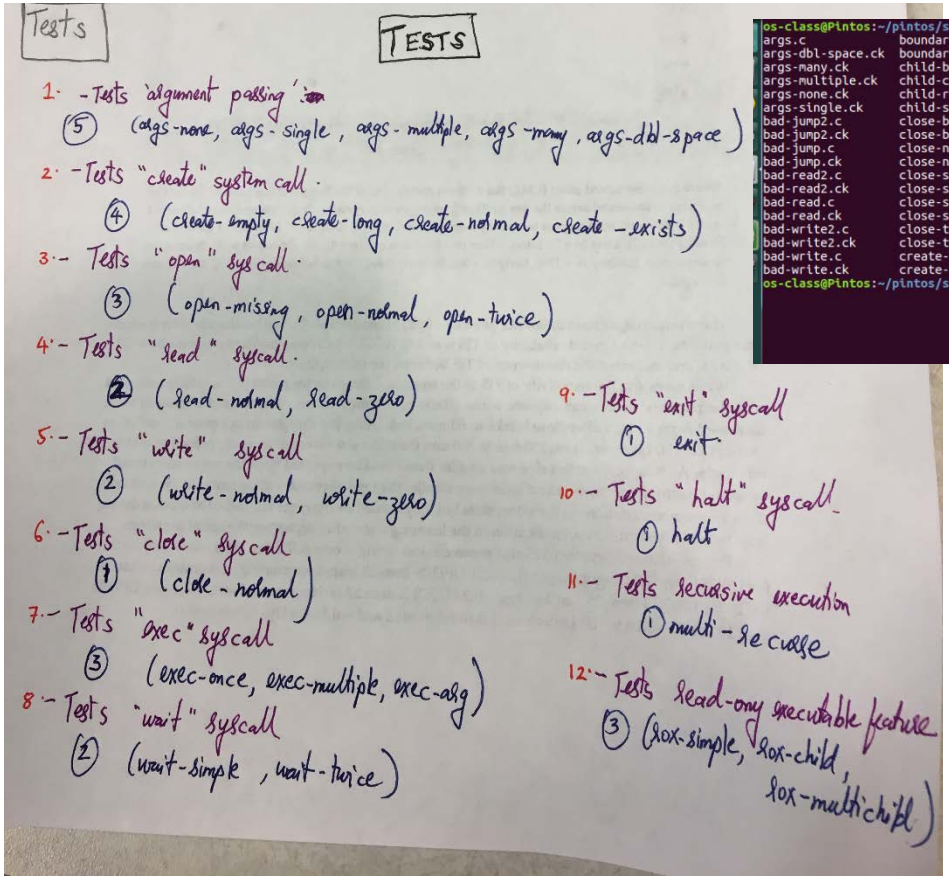
Step 2: Use gdb to identify the code that is invoked during test execution.

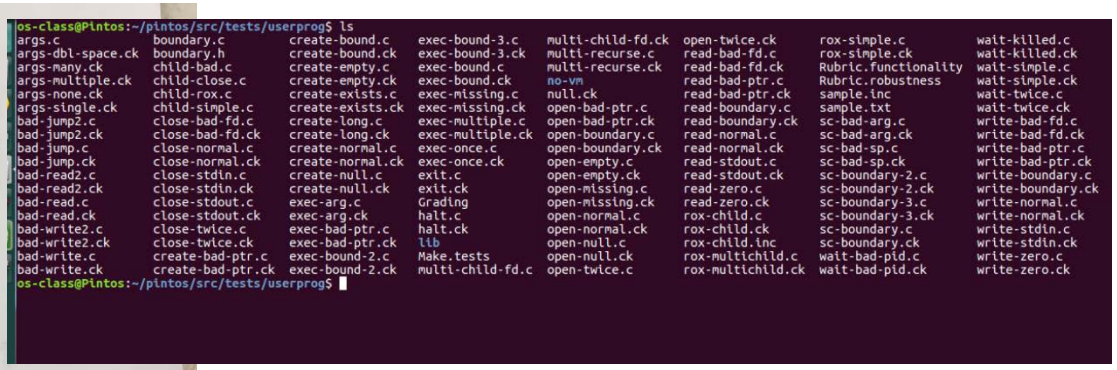
Step 3: Note the functions that you feel need to be extended.

Step 4: Implement different combinations of functions written down in paper to the functions noted in Step 3.

Necessities and Organization: Step 1

STEP1 : Familiarize with the tests associated with the component to implement.





```
os-class@Pintos:~/pintos/src/tests/userprog$ ls
args.c              boundary.c          create-bound.c      exec-bound-3.c      multi-child-fd.ck   open-twice.ck       rox-simple.c        wait-killed.c
args-dbl-space.ck   boundary.h          create-bound.ck     exec-bound-3.ck     multi-recurse.c     read-bad-fd.c       rox-simple.ck       wait-killed.ck
args-many.ck        child-bad.c         create-empty.c      exec-bound.c        multi-recurse.ck    read-bad-fd.ck      Rubric.functionality wait-simple.c
args-multiple.ck    child-close.c       create-empty.ck     exec-bound.ck       no-vm               read-bad-ptr.c      Rubric.robustness   wait-simple.ck
args-none.ck        child-rox.c         create-exists.c     exec-missing.c      null.ck             read-bad-ptr.ck     sample.inc          wait-twice.c
args-single.ck      child-simple.c      create-exists.ck    exec-missing.ck     open-bad-ptr.c      read-boundary.c     sample.txt          wait-twice.ck
bad-jump2.c         close-bad-fd.c      create-long.c       exec-multiple.c     open-bad-ptr.ck     read-boundary.ck    sc-bad-arg.c        write-bad-fd.c
bad-jump2.ck        close-bad-fd.ck     create-long.ck      exec-multiple.ck    open-boundary.c     read-normal.c       sc-bad-arg.ck       write-bad-fd.ck
bad-jump.c          close-normal.c      create-normal.c     exec-once.c         open-boundary.ck    read-normal.ck      sc-bad-sp.c         write-bad-ptr.c
bad-jump.ck         close-normal.ck     create-normal.ck    exec-once.ck        open-empty.c        read-stdout.c       sc-bad-sp.ck        write-bad-ptr.ck
bad-read2.c         close-stdin.c       create-null.c       exit.c              open-empty.ck       read-stdout.ck      sc-boundary-2.c     write-boundary.c
bad-read2.ck        close-stdin.ck      create-null.ck      exit.ck             open-missing.c      read-zero.c         sc-boundary-2.ck    write-boundary.ck
bad-read.c          close-stdout.c      exec-arg.c          Grading             open-missing.ck     read-zero.ck        sc-boundary-3.c     write-normal.c
bad-read.ck         close-stdout.ck     exec-arg.ck         halt.c              open-normal.c       rox-child.c         sc-boundary-3.ck    write-normal.ck
bad-write2.c        close-twice.c       exec-bad-ptr.c      halt.ck             open-normal.ck      rox-child.ck        sc-boundary.c       write-stdin.c
bad-write2.ck       close-twice.ck      exec-bad-ptr.ck     lib                 open-null.c         rox-child.inc       sc-boundary.ck      write-stdin.ck
bad-write.c         create-bad-ptr.c    exec-bound-2.c      Make.tests          open-null.ck        rox-multichild.c    wait-bad-pid.c      write-zero.c
bad-write.ck        create-bad-ptr.ck   exec-bound-2.ck     multi-child-fd.c    open-twice.c        rox-multichild.ck   wait-bad-pid.ck     write-zero.ck
os-class@Pintos:~/pintos/src/tests/userprog$
```

Necessities and Organization: Step 2

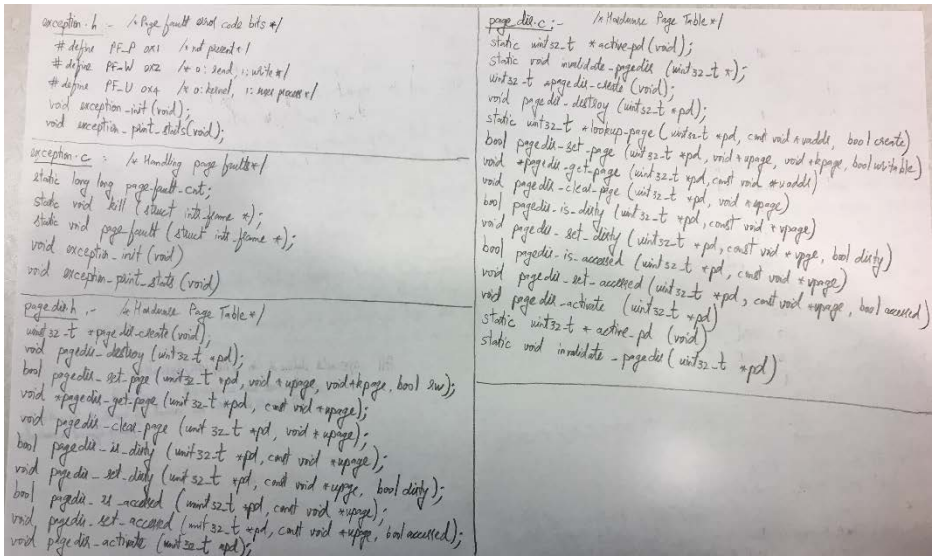
Step 2 : Use ~~gdb~~ normal execution to identify the code that is invoked during test execution.

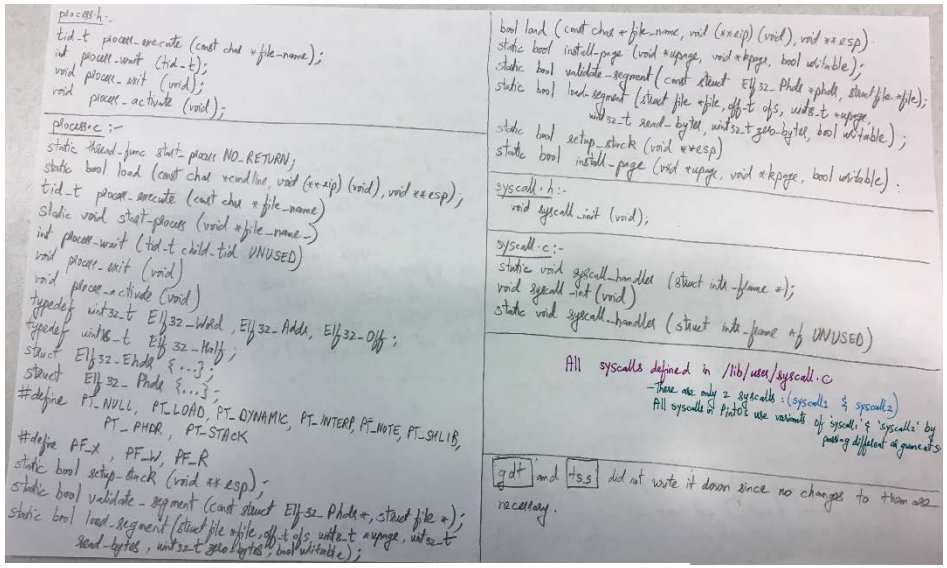
```
os-class@Pintos:~/pintos/src/userprog/build$ pintos -q run 'insult -n 4'
squish-pty bochs -q
00000000000i[APIC?] local apic in  initializing
========================================================================
                        Bochs x86 Emulator 2.2.6
              Build from CVS snapshot on January 29, 2006
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] user_shortcut: old-style syntax detected
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading..........
Kernel command line: -q run 'insult -n 4'
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 172 sectors (86 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'insult -n 4':
Execution of 'insult -n 4' complete.
Timer: 103 ticks
Thread: 0 idle ticks, 102 kernel ticks, 4 user ticks
hdb1 (filesys): 19 reads, 0 writes
Console: 628 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off..========================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
========================================================================
os-class@Pintos:~/pintos/src/userprog/build$
```

```
os-class@Pintos:~/pintos/src/userprog/build$ pintos -q run 'echo x'
squish-pty bochs -q
00000000000i[APIC?] local apic in  initializing
========================================================================
                        Bochs x86 Emulator 2.2.6
              Build from CVS snapshot on January 29, 2006
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] user_shortcut: old-style syntax detected
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading..........
Kernel command line: -q run 'echo x'
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 172 sectors (86 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo x':
Execution of 'echo x' complete.
Timer: 103 ticks
Thread: 0 idle ticks, 101 kernel ticks, 5 user ticks
hdb1 (filesys): 19 reads, 0 writes
Console: 613 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off..========================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
========================================================================
os-class@Pintos:~/pintos/src/userprog/build$
```

Necessities and Organization: Step 3

<u>Step 3</u> : Note the functions that you feel need to be extended.

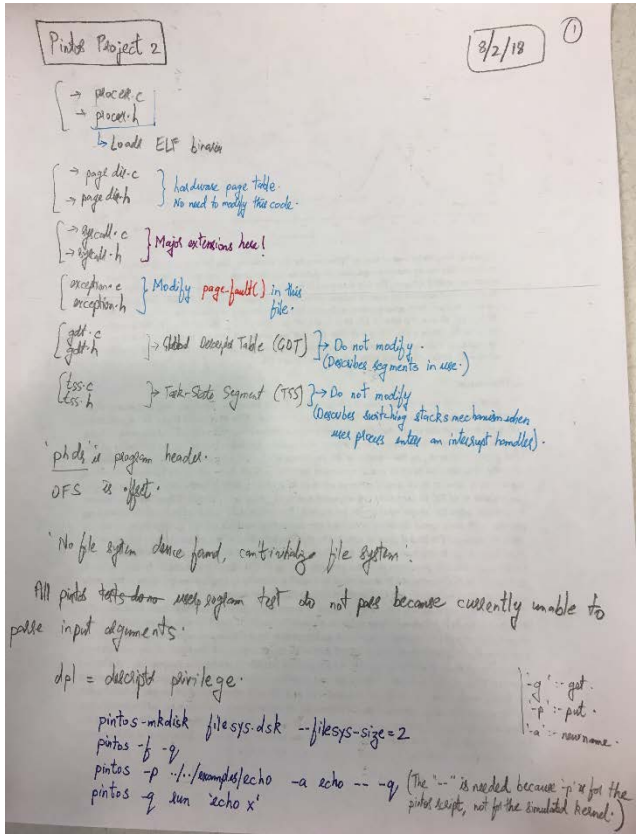




Function	Description
`process_execute`	Executes the user program from the designated file in the argument
`process_wait`	Waits for the child process with designated tid to finish before continuing execution
`process_exit`	Terminates user program currently running
`process_activate`	Sets up CPU to run user program in current thread

Necessities and Organization: Step 4

<u>Step 4</u>: Implement different combinations of functions written down in paper to the functions noted in Step 3.



My scratch paper regarding possible places for extensions.

In Project 2, rather than just applying different combinations, we need to create our own functions. (and thus challenging)

Overview

- Necessities and Organization
- Argument passing in PintOS
- Implementation of System Calls

Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack

Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack

Argument passing in PintOS : Where a User Program Starts

You create a filesystem disk and run a program called echo

```
pintos-mkdisk filesys.dsk --filesys-size=2
pintos -f -q
pintos -p ../../examples/echo -a echo -- -q
pintos -q run 'echo x'
```

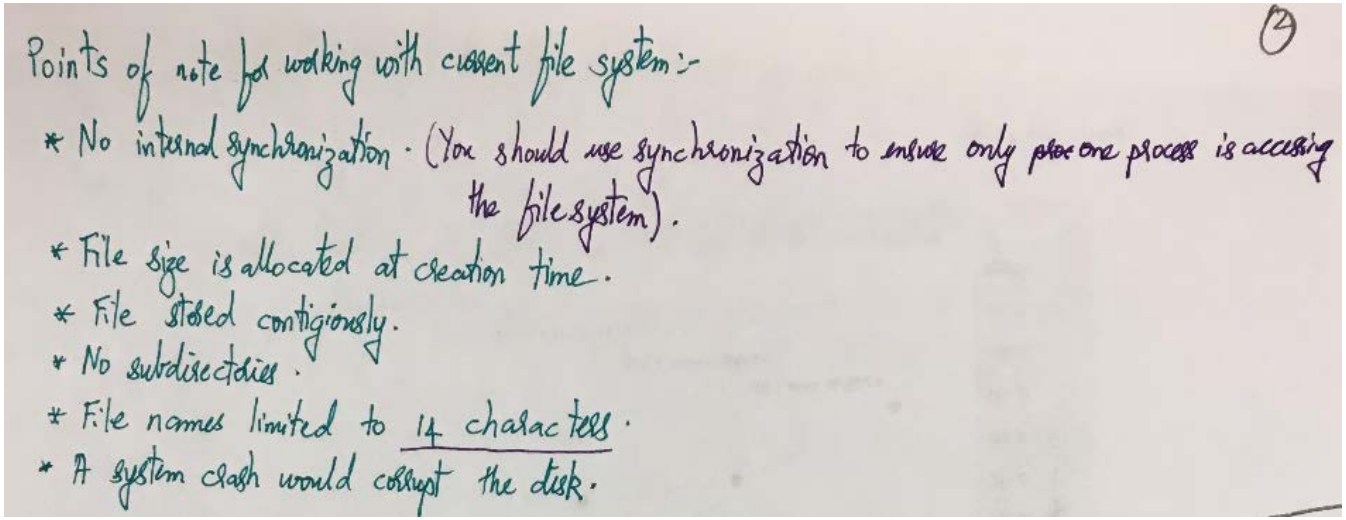
The program internally starts in `process_execute()`

```
tid = thread_create(file_name, PRI_DEFAULT, start_process, fn_copy);
```

In `load()` function in `process.c`, you'll find function called `setup_stack`. You'll need to setup stack here.

Argument passing in PintOS : Where a User Program Starts

Keep in mind these constraints.



Points of note for working with current file system :-

* No internal synchronization. (You should use synchronization to ensure only ~~prev~~ one process is accessing the file system).
* File size is allocated at creation time.
* File stored contigiously.
* No subdirectories.
* File names limited to 14 characters.
* A system crash would corrupt the disk.

Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack

Argument passing in PintOS : Emulate process_wait()

`process_wait()`. Returns immediately with processing input arguments

```
int
process_wait(tid_t child_tid)
{
    return -1;
}
```

You need to change `process_wait()` to `wait` for the child test case that has spawned.

```
int
process_wait(tid_t child_tid)
{
    while(true)
    {
        thread_yield();
    }
}
```

Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack

Argument passing in PintOS : Setup Stack

`setup_stack` only takes the stack pointer `void** esp`

`setup_stack` doesn't have access to filename.
You'll need to pass in the filename and file arguments to setup_stack using `strtok_r`

Make sure you account for *NULL terminated arguments* and *word alignments*.
Otherwise, input arguments would not be read properly.

```
char argument[] = "arg1\0"
*esp -= strlen(argument);
memcpy(*esp, argument, strlen(argument));
```

```
int word_align = 0, 1, 2, or 3
*esp -= word_align;
memset(*esp, 0, word_align);
```

Argument passing in PintOS: Using hexdump( ) for debugging

Format

```
static void hex_dump((uintptr_t)**, void**, int, bool);
```

Example

```
hex_dump((uintptr_t)*esp, *esp, sizeof(char) * 8, true);
```

Hex Dump for args-none

```
bffffe0  00 00 00 00 01 00 00 00-ec ff ff bf f6 ff ff bf |................|
bffffff0  00 00 00 00 00 00 61 72-67 73 2d 6e 6f 6e 65 00 |......args-none.|
```

Hex Dump for args-single

```
bffffd0                 00 00 00 00-02 00 00 00 e0 ff ff bf |    ............|
bffffe0  ed ff ff bf f9 ff ff bf-00 00 00 00 00 61 72 67 |.............arg|
bffffff0  73 2d 73 69 6e 67 6c 65-00 6f 6e 65 61 72 67 00 |s-single.onearg.|
```

Hex Dump for args-multiple

```
bffffb0                 00 00 00 00-05 00 00 00 c0 ff ff bf |    ............|
bffffc0  da ff ff bf e8 ff ff bf-ed ff ff bf f7 ff ff bf |................|
bffffd0  fb ff ff bf 00 00 00 00-00 00 61 72 67 73 2d 6d |..........args-m|
bffffe0  75 6c 74 69 70 6c 65 00-73 6f 6d 65 00 61 72 67 |ultiple.some.arg|
bffffff0  75 6d 65 6e 74 73 00 66-6f 72 00 79 6f 75 21 00 |uments.for.you!.|
```

Overview

- Necessities and Organization
- Argument passing in PintOS
- Implementation of System Calls

Implementation of System Calls

1. All 13 syscalls
2. The create syscall
3. The wait syscall
4. The exit syscall

Implementation of System Calls: All 13 syscalls

`halt, exit, exec, wait, create, remove, open, filesize, read, write, seek, tell, close.`

- When user program calls one of the functions in `lib/user/syscall.h`, a software interrupt happens and an interrupt frame is created.
- The frame is dispatched to `syscall_handler(struct intr_frame* f);`
- The type of syscall to execute is stored in `(f->esp)`

```
int sys_code = *(int*)f->esp;
```

Implementation of System Calls: All 13 syscalls

syscall codes in `src/lib/syscall-nr.h`

```
/* System call numbers. */
enum
{
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
    SYS_FILESIZE,               /* Obtain a file's size. */
    SYS_READ,                   /* Read from a file. */
    SYS_WRITE,                  /* Write to a file. */
    SYS_SEEK,                   /* Change position in a file. */
    SYS_TELL,                   /* Report current position in a file. */
    SYS_CLOSE,                  /* Close a file. */
};
```

Implementation of System Calls: All 13 syscalls

Parsing the `sys_code` in `src/userprog/syscall.c`

```
static void
syscasll_handler(struct intr_frame* f)
{
    //first check if f->esp is a valid pointer)
    if (f->esp is a bad pointer)
    {
        exit(-1);
    }

    //cast f->esp into an int*, then dereference it for the SYS_CODE
    switch(*(int*)f->esp)
    {
        case SYS_HALT:
        {
            //Implement syscall HALT
            break;
        }
        case SYS_EXIT:
        {
            //Implement syscall EXIT
            break;
        }
        (...)
```

Implementation of System Calls: All 13 syscalls

Extracting arguments from user program and passing to syscalls

```
static void
syscall_handler(struct intr_frame* f)
{
    switch(*(int*)f->esp)
    {
        case SYS_WRITE:
        {
            int fd = *((int*)f->esp + 1);
            void* buffer = (void*)(*((int*)f->esp + 2));
            unsigned size = *((unsigned*)f->esp + 3);

            //run the syscall, a function of your own making
            //since this syscall returns a value, the return value should be
                stored in f->eax
            f->eax = write(fd, buffer, size);
        }
    }
}
```

Implementation of System Calls: The create syscall

- All file system calls will be using function from either `src/filesys/file.h` or `src/filesys/filesys.h`
- All file-related syscalls are rather straightforward.

Prototype for `create` syscall

```
bool create (const char* file, unsigned initial_size)
{
    check to see if valid file pointer
    using synchronization constructs:
        //bool filesys_create (const char *name, off_t initial_size);
        bool = filesys_create(file pointer, initial size);
    return bool
}
```

Implementation of System Calls: The wait syscall

`wait` is the trickiest syscall to implement.

Points of note:

Careful interactions between the parent and the child process. Possible scenarios:

- Child becoming an orphan.
- Child becoming a zombie.
- Resume execution of parent thread after child completes.
- Processes sharing their child information to all other processes.
- Which situations should wait fail?
- etc.
- etc…….. (much more scenarios need to be self-discovered along the project.)

Implementation of System Calls: The exit syscall

PintOS requires a specific format for `exit` syscall.

```
<thread_current()->name>: exit(<exit status>)
```

For example:

```
Main Thread: exit(1)
```

Passing Tests

Passing Tests

1. Setup the stack properly.
2. Implement `process_wait`
3. Implement `write syscall` for `STDOUT_FILENO` with `putbuf`
4. Implement `exit syscall`

References

1. https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html
2. http://bits.usc.edu/cs350/assignments/Pintos_Guide_2016_11_13.pdf