Group Project
UFCFW-15-2
pintOS – Implementing argument passing and system calls

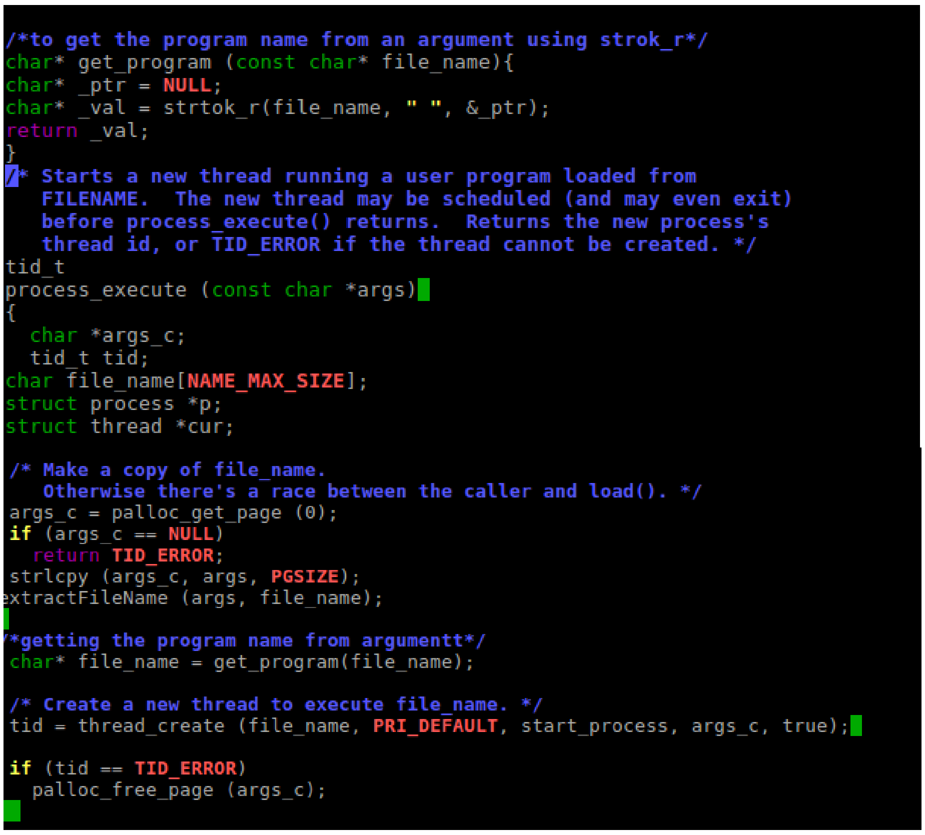
Shadhan Sidique - S2200235
Juwairiya Hussain Rasheed – S2200605

Group Assignment
pintOS – Implementing argument passing and system calls

1. Argument Passing:

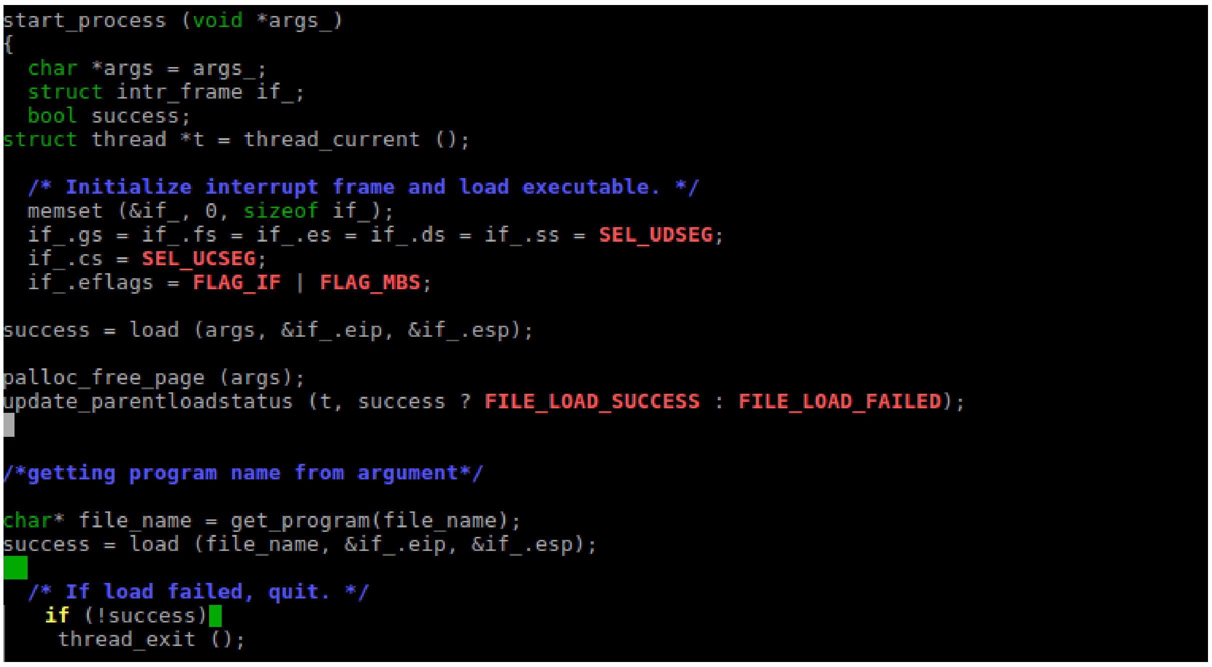
Currently, Pintos does not have the capability to pass arguments to newly created processes. This means that running commands like 'echo' or 'echo x y z' results in the same output, because the current code doesn't handle passing multiple arguments and treats them as part of the filename. The objective of the argument passing task is to address this limitation.

To tackle this issue, we begin by modifying the `process_execute` function. This involves utilizing the `strtok_r()` function, to split the file name into different parts. These split values are then passed appropriately when creating the thread.



```
/*to get the program name from an argument using strok_r*/
char* get_program (const char* file_name){
char* _ptr = NULL;
char* _val = strtok_r(file_name, " ", &_ptr);
return _val;
}
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   thread id, or TID_ERROR if the thread cannot be created. */
tid_t
process_execute (const char *args)
{
  char *args_c;
  tid_t tid;
char file_name[NAME_MAX_SIZE];
struct process *p;
struct thread *cur;

/* Make a copy of file_name.
   Otherwise there's a race between the caller and load(). */
args_c = palloc_get_page (0);
if (args_c == NULL)
  return TID_ERROR;
strlcpy (args_c, args, PGSIZE);
xtractFileName (args, file_name);

/*getting the program name from argumentt*/
char* file_name = get_program(file_name);

/* Create a new thread to execute file_name. */
tid = thread_create (file_name, PRI_DEFAULT, start_process, args_c, true);

if (tid == TID_ERROR)
  palloc_free_page (args_c);
```

The next step involves making adjustments to the `load` function and the `setup_stack` function. These modifications are necessary to enable the processing of programs with multiple arguments.



```
start_process (void *args_)
{
  char *args = args_;
  struct intr_frame if_;
  bool success;
struct thread *t = thread_current ();

  /* Initialize interrupt frame and load executable. */
  memset (&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;

success = load (args, &if_.eip, &if_.esp);

palloc_free_page (args);
update_parentloadstatus (t, success ? FILE_LOAD_SUCCESS : FILE_LOAD_FAILED);


/*getting program name from argument*/

char* file_name = get_program(file_name);
success = load (file_name, &if_.eip, &if_.esp);

  /* If load failed, quit. */
   if (!success)
    thread_exit ();
```

2. System Calls

The second task of the group assignment involves implementing a syscall handler for Pintos, encompassing various system calls:

void halt(void): This function terminates Pintos by invoking the "shutdown_power_off" function (defined in "threads/init.h"). It should be used sparingly, as it can result in the loss of some information about the current state of the running Pintos system.

void exit(int status): Terminates the calling user program and returns the status to the kernel. The exit status returned to the parent process waiting for it (see the wait syscall below) can differ.

pid_t exec(const char *cmd_line): Executes the executable specified in cmd_line, passing any given arguments, and returns the new process' program ID (pid). It should return -1 if the program fails to load or run for any reason. Synchronization mechanisms are necessary to ensure that the parent process waits until it knows whether the child process has successfully loaded its executable.

int wait(pid_t pid): Waits for a child process identified by its pid and retrieves the child's exit status. If the process with the specified PID is still running, the function waits until it terminates and then returns the status returned by the exited process. If the process was terminated by the kernel (e.g., due to an exception), the function returns -1. It fails and returns -1 immediately if certain conditions are met, such as if the specified PID does not refer to a direct child of the calling process.

bool create(const char *file, unsigned initial_size): Creates a new file named file with a size of initial_size bytes. Returns true if successful, otherwise false.

bool remove(const char *file): Deletes the file named file. Returns true if successful, otherwise false.

int open(const char *file): Opens the file named file. Returns a non-negative integer handle known as a "file descriptor" (fd), or -1 if the file couldn't be opened. File descriptors 0 and 1 are reserved for the console: fd 0 (STDIN_FILENO) is standard input, and fd 1 (STDOUT_FILENO) is standard output.

int filesize(int fd): Returns the size, in bytes, of the file opened with the given file descriptor (fd).

int read(int fd, void *buffer, unsigned size): Reads size bytes from the file associated with the file descriptor fd and places the read content into buffer. Returns the number of bytes actually read, or 0 if reaching the end of the file, or -1 if the file couldn't be read.

int write(int fd, const void *buffer, unsigned size): Writes size bytes from buffer to the file associated with the file descriptor fd. Returns the number of bytes actually written, which may be less than size.

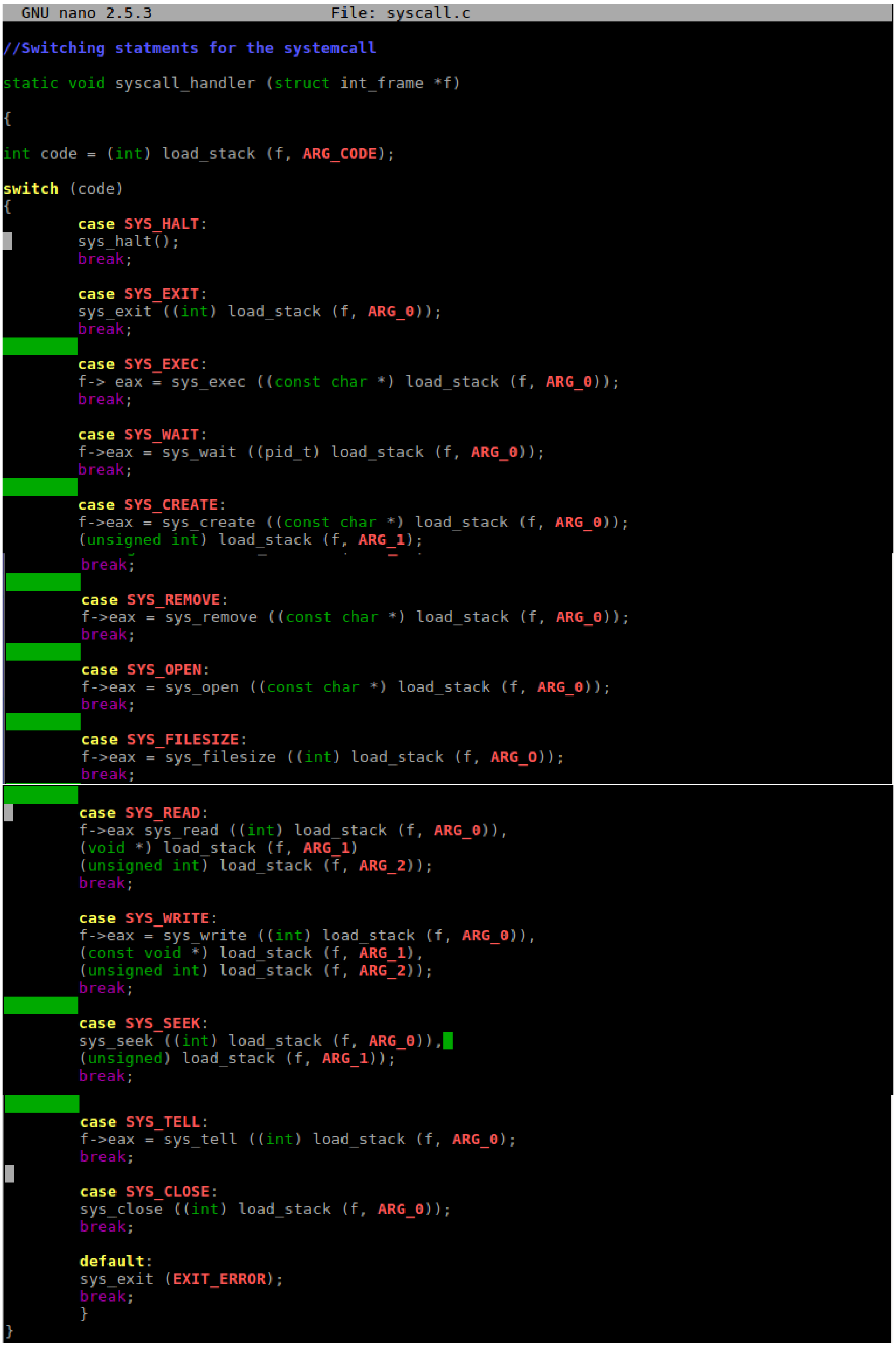
void seek(int fd, unsigned position): Changes the next byte to be read or written in the file associated with fd to the specified position.

unsigned tell(int fd): Returns the position of the next byte to be read or written in the file associated with fd.

void close(int fd): Closes the file descriptor fd.

These syscall implementations collectively enhance the functionality of Pintos and enable various operations to be performed by user programs.

Below are the custom sub functions of the system call functions in the system_handler()



```
  GNU nano 2.5.3                   File: syscall.c

//Switching statments for the systemcall

static void syscall_handler (struct int_frame *f)

{

int code = (int) load_stack (f, ARG_CODE);

switch (code)
{
        case SYS_HALT:
        sys_halt();
        break;

        case SYS_EXIT:
        sys_exit ((int) load_stack (f, ARG_0));
        break;

        case SYS_EXEC:
        f-> eax = sys_exec ((const char *) load_stack (f, ARG_0));
        break;

        case SYS_WAIT:
        f->eax = sys_wait ((pid_t) load_stack (f, ARG_0));
        break;

        case SYS_CREATE:
        f->eax = sys_create ((const char *) load_stack (f, ARG_0));
        (unsigned int) load_stack (f, ARG_1);
        break;

        case SYS_REMOVE:
        f->eax = sys_remove ((const char *) load_stack (f, ARG_0));
        break;

        case SYS_OPEN:
        f->eax = sys_open ((const char *) load_stack (f, ARG_0));
        break;

        case SYS_FILESIZE:
        f->eax = sys_filesize ((int) load_stack (f, ARG_0));
        break;

        case SYS_READ:
        f->eax sys_read ((int) load_stack (f, ARG_0)),
        (void *) load_stack (f, ARG_1)
        (unsigned int) load_stack (f, ARG_2));
        break;

        case SYS_WRITE:
        f->eax = sys_write ((int) load_stack (f, ARG_0)),
        (const void *) load_stack (f, ARG_1),
        (unsigned int) load_stack (f, ARG_2));
        break;

        case SYS_SEEK:
        sys_seek ((int) load_stack (f, ARG_0)),
        (unsigned) load_stack (f, ARG_1));
        break;

        case SYS_TELL:
        f->eax = sys_tell ((int) load_stack (f, ARG_0);
        break;

        case SYS_CLOSE:
        sys_close ((int) load_stack (f, ARG_0));
        break;

        default:
        sys_exit (EXIT_ERROR);
        break;
        }
}
```