Operating Systems Coursework Design Documentation

This is a variant of the Pintos Operating System for use as part of Operating Systems module at the University of the West of England (UWE).

This report is for the group members:

Callum Duncan - Student ID: 21047076

Adam Selman - Student ID: 20049296

Sandra Sahnoune - Student ID: 21039366

Wali Shaikh - Student ID: 21039370

GitLab Repository:

https://gitlab.uwe.ac.uk/a2-selman/o.s-group-project

PintOS Documentation

Individual Contributions of Group Members

All descriptions written here have been written by the individual.

Callum Duncan

My contribution to PintOS was the implementation of Task 1's argument passing and setting up the stack within the user space. I also helped out with implementation of the functioning system calls. I also wrote the documentation for how I went about implementing argument passing and setting up the stack.

Adam Selman

My contribution towards this project was primarily in correspondence to implementing system calls and the writing the documentation for all features I worked on (system call handler, `halt`, `exit` and `write`).

The first task I contributed to was figuring out how to access the system call code number and pass it though the system call handler to decide which system call to use.

The system calls I implemented were the `halt` and `exit` system calls. A start was made with `write` as well but it was unfortunately unfinished. I wrote the documentation for these 3 system calls.

Sandra Sahnoune

My contribution towards this project was the implementation of multiple Syscalls `write`, `open`, `close` and `create`.
I also supported the syscalls with other functions and structures such as 'Get user', 'Validate pointer', Validate Buffer' and 'get file' as well as writing a detailed explanation of them on the report.

Wali Shaikh

My contribution towards this project was implementing open close and create system calls.

String Tokenisation and Argument Passing (*Implemented by Callum Duncan*)

Setup

Before any of the argument passing and stack implementation can be introduced, the `setup_stack()` function needs to understand that it's receiving a file name. To do this, `file_name` needs to be implemented into the function.
At lines `206` & `452`, passing `file_name` allows the file name to pass through the function:

```
206: static bool setup_stack (void **esp,void **eip,char *file_name);
452: setup_stack (void **esp, void **eip, char *file_name){...}
```

At the top of the `setup_stack` function there are a number of variables declared to be used for tokenisation.

- A `char` pointer `sptr` keeps track of the stack pointers position, sptr is an abbreviation for "save pointer".
- A `char` pointer `tkn` which refers to each individual argument/token as tokenisation occurs.
- A 64 bit array `argv` which holds the arguments passed by a command.
- An `int` value `argc` has a dual purpose, being used to add arguments to the next available position in `argv` as well as maintaining a total count of how many arguments have been passed.
- A copy of the filename is saved under `fn_copy` from `palloc_get_page`. This uses a predefined block of code previously used in process.c.

Rationale

Below is the code for the copying of a file, to prevent any race conditions from occurring. This code was taken from lines `37-40` from `process_execute()` and re-implemented to be used with the tokenization loop.

```
fn_copy = palloc_get_page (0);
 if (fn_copy == NULL)
   return TID_ERROR;
 strlcpy (fn_copy, file_name, PGSIZE);
```

First, we check that the file_name, `fn_copy`, is unallocated (`NULL`) before proceeding. Then we use `strlcpy` to allocate the `file_name` and filesize, defined as `PGSIZE`, to the `fn_copy` variable.

```
for (tkn = strtok_r (fn_copy, " ", &sptr); tkn != NULL;tkn = strtok_r (NULL, " ", &sptr))
 {
      argv[argc] = tkn;
      argc++;
   }
```

To tokenise, the input string is looped over utilising the `strtok_r` function with the delimiter containing a space `" "`. Each tokenized argument is then added to `argv` by setting the value at index `argc` to value `tkn`. The value of `argc` is then incremented. This continues until `tkn` takes a null value.

Now that this is complete, there are now tokenized arguments to pass to the stack. To start this, variables are initialized:

```
int count = argc-1;
int currentlen=0;
int totallen;
char *argvptr[argc];
char *esp_memblock;
```

The variables listed above are responsible for setting up the dynamic allocation of the stacks. With `count = argc-1`, the `-1` is needed as one of the arguments is equal to null, which does not need to be passed to the stack.

```
while(count >= 0)
  {
     currentlen = ((int)strlen(argv[count]))+1;
     totallen = totallen + currentlen;

     esp_memblock = *esp - currentlen;
     argvptr[count] = esp_memblock;

       *esp -= 1;
     memset(*esp, 0 , 1);

       *esp -= strlen(argv[count]);
       memcpy(*esp, argv[count], strlen(argv[count]));
```

```
        count--;
    }
```

We will be separating the code into "chunks" for easier explanation:

```
currentlen = ((int)strlen(argv[count]))+1;
totallen = totallen + currentlen;
```

this is responsible for tracking the total length of the arguments passed to stack, which will be needed later.
`((int)strlen(argv[count]))+1;` uses the `strlen` function to save the length of the current argument being passed into the loop, the `(int)` specifies to the variable that it needs to be saved as an integer, the `+1` is adding an extra byte to allow for the escape character `\0`.
`totallen` is concatenated with `currentlen` to increase the total size for each loop.

```
esp_memblock = *esp - currentlen;
argvptr[count] = esp_memblock;
```

As this is before the stack pointer `esp` has moved, we can manipulate this within the context of the loop to find where the stack pointer *will* be, and we want to save these addresses to another array to pass to the stack later.
Therefore, subtracting the stack pointer by the current length and saving this in the `argvptr[]` array gives us an array of the addresses that we can later pass to the stack.

```
esp -= 1;
memset(*esp, 0 , 1);
*esp -= strlen(argv[count]);
memcpy(*esp, argv[count], strlen(argv[count]));
count--;
```

This block of code will now set and copy the elements into the stack. Firstly, to account for the "gap" between each argument, the stack pointer gets moved back once, and sets a "nop" value to the stack to divide the arguments.
Next, the stack pointer will be decremented by the string length of the current argument in the loop. `memcpy()` then places the argument into the area the stack pointer is currently pointing to, by the length of that argument.
The `count` is then decremented. This is done because we are using the same `argc` variable from the tokenization stage, as it is already holding the highest value that the `strtok_r` loop reached. Therefore, the stack can be implemented backwards as intended.

This is the output for the example of “pintos -q run ‘echo x y z’ ”:

```
=== INITIAL ARGUMENTS ===
bffffff0                                               7a 00 |              z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0 2c d6 00 f0 |........,...,...|
c0000030  2c d6 00 f0 2c d6 00 f0-57 ef 00 f0 2c d6       |,...,...W...,.  |

=== INITIAL ARGUMENTS ===
bffffff0                                         79 00 7a 00 |            y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0 2c d6 00 f0 |........,...,...|
c0000030  2c d6 00 f0 2c d6 00 f0-57 ef 00 f0             |,...,...W...    |

=== INITIAL ARGUMENTS ===
bffffff0                                   78 00 79 00 7a 00 |          x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0 2c d6 00 f0 |........,...,...|
c0000030  2c d6 00 f0 2c d6 00 f0-57 ef                   |,...,...W.      |

=== INITIAL ARGUMENTS ===
bffffff0                    65 63 68-6f 00 78 00 79 00 7a 00 |     echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0 2c d6 00 f0 |........,...,...|
c0000030  2c d6 00 f0 2c                                  |,...,           |
```

The `totallen` variable can now be used to determine the amount of nop's needed to align before pushing the extra 4 bytes:

```
switch(totallen%4)
   {
  case 0:
    break;
  case 1:
    *esp -= 3;
    memset(*esp, 0, 3);
  break;
  case 2:
      *esp -= 2;
        memset(*esp, 0, 2);
  break;
  case 3:
        *esp -= 1;
      memset(*esp, 0, 1);
  break;
}
```

This switch statement takes the total length of the argument, to the modulus of 4. There is a case statement for the values of 1, 2 or 3. This is determined on what the modulus *is going to be*. if the modulus is 1, the stack pointer will input 3 nops, if the case is 2, it will move and set 2 nops and so forth.

```
esp -= 4;
memset(*esp,0,4);
```

After the arguments have been placed and aligned to the stack, another 4 bytes of nop's are necessary. This is as simple as moving the stack pointer back by 4 and setting the empty space as 0's.

This is the `hex_dump` of the outputs using the same example previously:

```
===WORD ALIGN===
bffffff0              00 65 63 68-6f 00 78 00 79 00 7a 00 |    .echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0 2c d6 00 f0 |........,...,...|
c0000030  2c d6 00 f0                                     |,...            |

=== ALIGN 4 BYTES ===
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0 2c d6 00 f0 |........,...,...|
```

```
int c = argc-1;
while(c >= 0)
{
    *esp -= sizeof(argvptr[c]);
    memcpy(*esp, &argvptr[c], sizeof(argvptr[c]));
    c--;
}
```

Now that the previous steps have been implemented, the `argvptr[]` variable will now be used within another decrementing `while` loop to store the addresses of the initial arguments.
The loop demonstrated here works the same as the previous while loop, decrementing using `argc-1`. This time, placing the addresses of the arguments into the stack with `memcpy`. The `&` sign is used to retrieve the address of the arguments within the array.

```
char *adr;
adr = *esp;
*esp -= 4;
memcpy(*esp, &adr, 4);
```

Because the stack needs to push a double pointer to the first argument within the stack, a variable `*adr` is declared and equalled to `esp` current location. This is taking advantage of the fact that the stack is implemented in reverse order, and that the current stack pointer location is pointing to the first argument's address. As this is the case, `adr` stores where the stack is pointing to *before* moving `esp` backwards. Now that this is stored, `esp` is moved backwards and the address of esp's (using `&adr`) previous location is placed in stack with `memcpy`.

```
esp -= 3;
memset(*esp, 0, 3);
*esp -= 1;
memset(*esp, argc, 1);
```

To write the argument count to the stack, the stack pointer is decremented by 3 bytes which is set to 0. Then, the stack pointer moves back once more, this time placing the argument count (`argc`) in one byte.

```
void *ptr2 = NULL;
*esp -= 4;
memcpy(*esp, &ptr2, sizeof(void*));
```

Finally, to place the null pointer, `*ptr2` is set to `NULL` and this address is then placed within the stack using the `memcpy` function.

Shown below is the `hex_dump` for the previously discussed sections:

```
=== ADDRESS ALLOCATION ===
bfffffe0                                      fe ff ff bf |            ....|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-2c d6 00 f0             |........,...    |

=== ADDRESS ALLOCATION ===
bfffffe0                          fc ff ff bf fe ff ff bf |        ........|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-                        |........        |

=== ADDRESS ALLOCATION ===
bfffffe0              fa ff ff bf-fc ff ff bf fe ff ff bf |    ............|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|
c0000020  a5 fe 00 f0                                     |....            |

=== ADDRESS ALLOCATION ===
bfffffe0  f5 ff ff bf fa ff ff bf-fc ff ff bf fe ff ff bf |................|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...S...S...S...|

===ARG[0]===
bfffffd0                                      e0 ff ff bf |            ....|
bfffffe0  f5 ff ff bf fa ff ff bf-fc ff ff bf fe ff ff bf |................|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-53 ff 00 f0             |S...S...S...    |

===ARGC===
bfffffd0                          04 00 00 00 e0 ff ff bf |        ........|
bfffffe0  f5 ff ff bf fa ff ff bf-fc ff ff bf fe ff ff bf |................|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 53 ff 00 f0-                        |S...S...        |

===NULL POINTER===
bfffffd0              00 00 00 00-04 00 00 00 e0 ff ff bf |    ............|
bfffffe0  f5 ff ff bf fa ff ff bf-fc ff ff bf fe ff ff bf |................|
bffffff0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0                                     |S...            |
```

Additional Utility Functions

Some additional functions were implemented to enable the features required to implement some system calls.

Get User

The get user function is utilised by the `validate_pointer` utility function. It has been taken from the official pintOS documentation Ben Pfaff (2009).

Its usage is defined as follows: *"Reads a byte at user virtual address UADDR. UADDR must be below PHYS_BASE. Returns the byte value if successful, -1 if a segfault occurred."* Ben Pfaff (2009).

Validate Pointer (*Implemented by Sandra Sahnoune*)

This functions checks if the pointer is pointed on the correct position in the stack and if `VADDR` is a user virtual address. If not it exits and returns an error.

Validate Buffer (*Implemented by Sandra Sahnoune*)

This function checks the length of the `buffer` in order to create a temporary copy of the inputs in it. If the buffer has to save a maximum of bytes and if the buffer's pointer is valid
it returns a true value.

Get File (*Implemented by Sandra Sahnoune*)

The Structure file points to the `get_file` function that gets the integer value of the file descriptor. This function aims to find files that match andshare the same file descriptor of the current file inside this function, we assign the function of `thread_current` to the structure of the current thread running.
A `process_file` structure is initialised in order to get the address of the file to be processed.
A `list_elem` is initialised to get the address of the list declared in `thread` .

A for loop is used to get each element of the file from the start and assign the list value to `list_begin` function declared in `lib/kernel/list.h` in order to get the address of the current running process and store its values in `file_list` .
If the file does not end yet the current process will keep reading its elements till it reaches the end and move to the next list. The processing file will get the next entry in the `file_list` as the `list_entry` will check its list, structure `process_file` and elements then it will compare the `process_file` file descriptor to the current file descriptor returning true if the file was found then `process_file` will be assigned to the current file. If not, it will return a `NULL` value.

System Calls

System Call Handler (*Implemented by Adam Selman*)

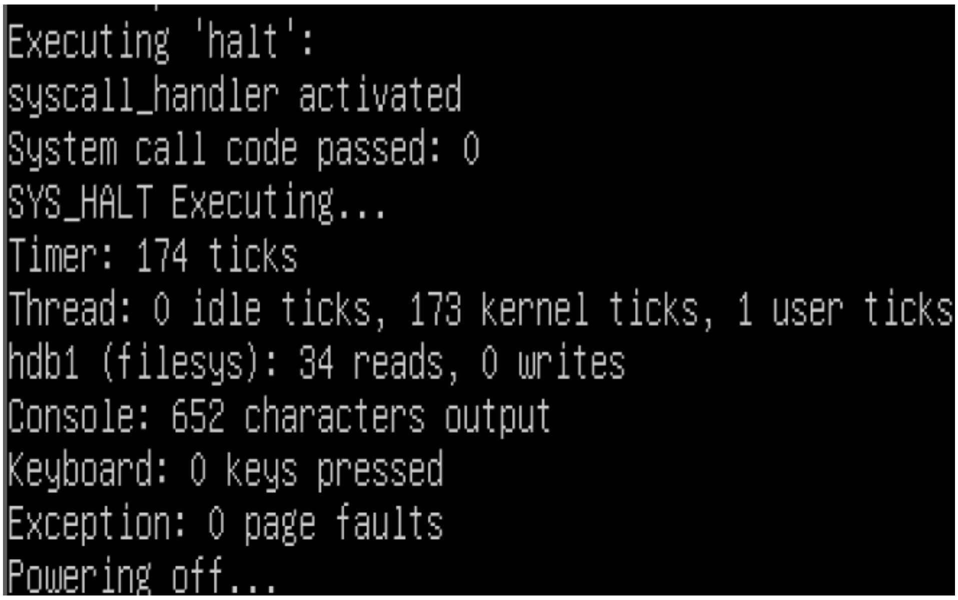
Before we could execute any system calls, a method of determining which system call was needed had to be made. To do this we added functionality to the pintOS `syscall_handler`. We first accessed the system call code via the interrupt frame, `intr_frame *f`. This was done by accessing the data stored at the stack pointer (`f -> esp`) and storing it in a variable. A `switch` statement could then be used to compare this value to the pre-defined system call codes defined in `syscall-nr.h`.

```
syscall_handler (struct intr_frame *f)
{
  int syscall_code = *(int *) f -> esp; // gets stack pointer from interrupt frame  | A.S
  switch (syscall_code)
  {
  case SYS_HALT:  // Implemented by Adam S
...
```

Halt (*Implemented by Adam Selman*)

The halt system call, assigned to system call code 0, utilised an in-built function of pintOS to cause an immediate shutdown. This can be seen in the `sys_halt ()` utilising the `shutdown_power_off` function.

```
case SYS_HALT:  // Implemented by Adam S
    printf("SYS_HALT Executing...\n");
    sys_halt();
    break;
...
void sys_halt(void)
  {
    shutdown_power_off(); // “Terminates Pintos by calling 'shutdown_power_off'"
  }
```



Results of running the halt system call test program

To test, an in built example file was used to utilise the `halt` system call. The result was as expected, the syscall handler is activated, the halt system call code is detected (0) and then `SYS_HALT` case from the switch statement is run.

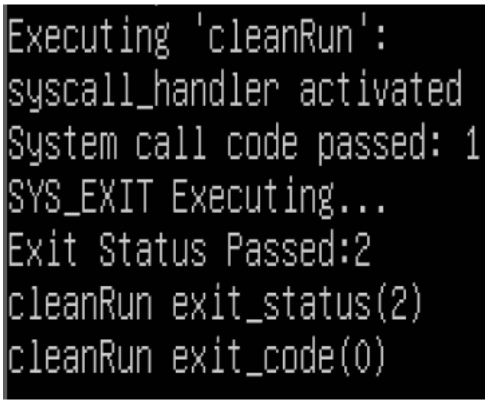
Exit (*Implemented by Adam Selman*)

To setup for this system call it was necessary to add an `exit_status` value to the thread structure of `thread.h`. This could be passed the value to return when exiting the thread.

The exit system call, assigned to system call code 1, terminates the currently running process and returns the status code of the process. To do this, we need to access the arguments via the interrupt frame `intr_frame *f`. We get the status by adding 1 to the interrupts stack pointer value, `f -> esp + 1`, and assigning it to an `int` variable `status`.

We then created a function called `sys_exit` to do the processing for the exit call. We accessed the currently running thread, naming it `*cur`, using the pintOS `thread_current()` function. We can then set the `exit_status` value to the `status` value we retrieved and exit the thread using the pintOS function `thread_exit()`.

```
case SYS_EXIT:  // Implemented by Adam S
    printf("SYS_EXIT Executing...\n");
    int exit_status = *(int *) f -> esp + 1; // getting exit_status from stack
    sys_exit(exit_status); // calling sys_exit function with the exit status
    break;
...
void sys_exit(int status)
  {
    struct thread *cur = thread_current(); // get the current running thread
    cur->exit_status = status; // set its exit status to the new status
    thread_exit();
  }
```



Result of running a simple program which does 1 + 1 and exits called "cleanRun"

To test the exit call, I created a test C file called "cleanRun" which did a simple calculation of 1 + 1 and then ends, which invoked the `exit` system call. The above results show the system call handler being called to exit, the `exit_status` retrieved from the stack and the result being displayed to the user with the process name and the `exit_status`.

Write (Implemented by Adam Selman, Callum Duncan & Sandra Sahnoune)

The first thing we do is get the necessary arguments to write to a file. These are the file descriptor, `fd`, the `buffer`, and the `size` of the buffer. These arguments are each accessed in a slightly different way.

```
int fd = *((int*)f->esp + 1); // file descriptor
void* buffer = (void*)(*((int*)f->esp + 2));  // buffer of values
unsigned size = *((unsigned*)f->esp + 3); // size of buffer
```

The file descriptor is accessed in the same way as is shown previously as it is of type `int`

The reasoning for the void casting for the `buffer` is explained by Stephen Tsung-Han Sher (2018) in section 4.4.3. First, we need to utilise the stack pointer as an integer first to increment it with a +2 as "f->esp can only be incremented with +1, +2, +3 if and only if it is of an int* type" Stephen Tsung-Han Sher (2018). The stack pointer will then get the values stored in the stack using `f -> esp + 2` and needs to be cast into a `void` type. "If we directly cast this int* into a void*, you will be getting the address of the buffer, not the buffer itself, therefore we need to dereference the int* in order to get the contents, then cast it into a void*" Stephen Tsung-Han Sher (2018).

The buffer `size` is of type `unsigned` so we cast the value to the unsigned type when accessing it with an increment of +3 to the stack pointer.

```
case SYS_WRITE:  // system call 9
    printf("SYS_WRITE Executing...\n");
    int fd = *((int*)f->esp + 1); // file descriptor
    void* buffer = (void*)(*((int*)f->esp + 2));  // buffer of values
    unsigned size = *((unsigned*)f->esp + 3); // size of buffer
    printf("File Descriptor:%d\nBuffer: %s\nSize:%d\n", fd, (char*) buffer, size);
    off_t bytes_written = sys_write(fd, *buffer, size);
    break;
...
int sys_write(int fd, void *buffer, unsigned size)
{

  if (validate_buffer(buffer, size) == false) // validate the buffer
  {
    return EXIT_ERROR; // return -1 if invalid
  }

  // get the file relating to the file descriptor
  struct file *file_to_write = get_file(fd);


  sema_down(&fileSema); // mark file resource as in use
  file_write(file_to_write, buffer, size);
  sema_up(&fileSema);  // release the resource

  off_t bytes_written = size;
```

```
    return bytes_written;
}
```

Once these arguments have been stored in variables they are passed to the `sys_write` function.

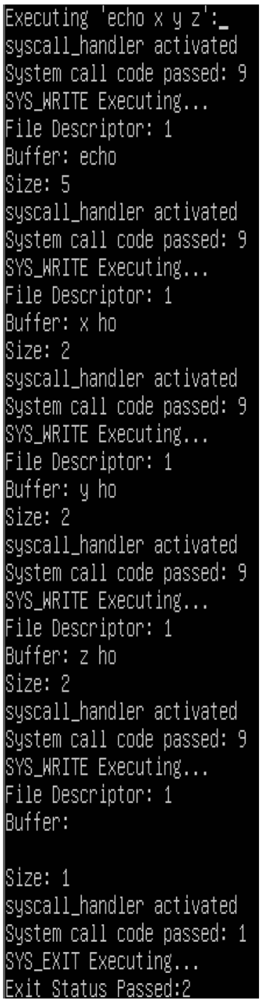
First, we need to validate whether the contents stored inside of the buffer contain valid pointers. This is done with the `validate_buffer` function we implemented which returns a Boolean value.

Next, we looked at retrieving the file using its file descriptor `fd` with the `get_file` function we implemented.

Once we have all of this information we can then utilise the built in pintOS functionality of `file_write` to perform the writing of the data.

```
sema_down(&fileSema);
file_write(*file_to_write, (void*)buffer, size);
sema_up(&fileSema);
```

For thread security, we implemented the use of semaphores to prevent any other processes from accessing a file at the same time as seen above. This is done using the built in `sema_down` and `sema_up` functions in pintOS to lock the file and then release it after writing.



Result of running “echo x y z”

To test, we tried running the echo example with the arguments “x y z” following it. As can be seen above, the write system call is utilised for each tokenised argument. For each, the file descriptor “1” is shown to represent standard output. The buffer shows the argument currently being processed followed by a NOP which is reflected in the size of 2 for the “x”, “y” and “z” arguments and is 5 for “echo” as they have the argument length + 1 for the NOP.

Unfortunately we were unable to fully implement this system call and have it properly function, but the above explained the reasoning and theory behind what we were able to write.

Open (Implemented by Sandra Sahnoune & Wali Shaikh)

The `open` function aims to open a file. The structure file goes to fetch the file and get its address. `filesys_open` is used which is declared in `filesys.c/h` file.

A condition is implemented to check if the file is not open correctly and display an ERROR and returns -1.
If the file is valid, the semaphore will wait and set to up to store the file and display its directory.

Close (Implemented by Sandra Sahnoune & Wali Shaikh)

This function aims to close an opening file by calling an external function `close_file(fd);`
This function application is to close a file , it gets the current running thread and the address of the current file processing and its elements.
A for loop is implemented to read all the elements of the file from the beginning using `list_begin` by going through `file_list` declared in `file.h` and get each elements in the list.
`file_to_process` get the elements of the list and from the `process_file` structure declared in `file.h` it executes the file name.
If both file descriptors are the same the current file process will close using `file_close` declared in `file.h`

Create (Implemented by Sandra Sahnoune & Wali Shaikh)

This function creates a file and return a Boolean value depending on the success of creating it.
It checks if the file exists , if it does the `filesys_create` function declared in `filesys.h` will create a file checking its name and size , the semaphore will turn to 1 and it displays a Boolean value if the file is created correctly. It will return the `isCreated` value of false if created and true if not created.

Other System Calls

For the remaining system calls all that was implemented was a basic “EXECUTING <system call name>” message that could be used for debugging.

References:

Stephen Tsung-Han Sher (2018) *CSCI 350: Pintos Guide* Available from:
https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Gu
[Accessed 13 January 2021]

Ben Pfaff (2009) *Stanford's CS140 in Winter 2009 Pintos Guide* Available from:
https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html#SEC39 [Accessed 2 January]