Implementing User Programs on PintOS

Andrew Belcher

November 4, 2020

Contents

1 Introduction

The task of this documentation is to detail the implementation of user land programs within PintOS. A simple instructional operating system for the x86 architecture. Some of this implementation requires input and output to a file system and synchronisation or control of different processes running at one time in order to run various simple programs to their completion. Implementing virtual memory or a file system directory is beyond the scope of this project.

2 Argument Handling

When running PintOS for user programs, we have no way to handle arguments. When we run something like `pintos -q run 'echo hello world!'` our OS will attempt to launch echo if it is inside the file system. However it wont be able to use any of the data we feed it such as the `hello world!`, since the echo program only can process its data via system calls and a userland stack. In attempting to get a basic user program like echo running on PintOS, we need to do 2 things mainly, parse our the arguments fed to our executed program, and populate a visible portion of memory with them so that the kernel can access the necessary data for basic I/O.

To do this we will need to look into `process.c`. The control flow for how our arguments get passed around and ultimately offered up to the a syscall handler looks like the following.

```
process_execute -> start_process -> load -> setup_stack
```

2.1 Arguments Splitting

In order for our syscalls to be passed any data, our arguments needs to be split into separate strings, we can do this with the help of `strtok_r`(a re-entrant version of `strtok`). This function does something known as string tokenization where we can split a string into tokens by using a delimit(in our case `" "` is our delimiter to separate by).

2.1.1 `process_execute`

To make sure our arguments are handled properly, the first step is to ensure they are passed into the thread when creating so they can be accessed later when setting up the stack. The changes to `process_execute` are as follows..

```
char* f_args; /* file name along with args to be
passed to thread_create */
...

/* Make a copy of FILE_NAME.
Otherwise there's a race between the caller and load(). */
f_args = palloc_get_page (0);

if (f_args == NULL)
  return TID_ERROR;
...

// load file and args string into copy ready for parsing
strlcpy (fname, file_name, PGSIZE);
strlcpy (f_args, fname, PGSIZE);
thread_args = f_args;
...

// create the thread with filename and pointer to full
// argument string will be parsed out later in setup_stack
tid = thread_create (fname, PRI_DEFAULT, start_process, thread_args);
```

This code just makes a separate page of memory to then copy both the file-name and arguments too so there isnt confusion when load accesses the page later on. We will mention how we pull out the process name later in chapter `2.2`, but for now the focus is that `thread_args` has been passed to the thread to be accessed later for parsing.

Based on the first example that might look like..

i.e. `thread_args = 'echo hello world!'`

2.1.2 `stack_setup`

Now that we have our arguments accessible by the thread and passed along to the others functions needed for setting up a processs, we need to focus on how this process will get them into its stack for when it needs to make system calls. We do this in `setup_stack` however we need to make some changes to how that function is called. As shown in `chapter 2:Agument Handling`, `setup_stack` is called from `load` which its purpose is to load the executable in memory for running, if it fails to load we wont enter `setup_stack` so we need to find the success related code. `load` takes the full arguments string as its first argument, however we pull out the filename for loading and pass the arguments string onto `setup_stack`.

This code hits if the executable is successfully loaded.

```
/* Set up stack. */
if (!setup_stack (esp, file_name))
  goto done;
```

We also need to change how this function is defined to include 1 extra argument.

```
static bool setup_stack (void **esp, char* args_unparsed);
```

Previously we would set the stack pointer back from the base in order to atleast prevent PintOS from crashing, this would fake a empty frame for the purpose of testing, but now we need to implement it ourselves so we set it back to the base.

```
    *esp = PHYS_BASE;
```

2.1.3 Parse the aruments

In order to get each argument out of our single string being passed in, we need to allocate some memory to store it into and point a data structure for our stack data at it.

First we create a data structure for `process.h`

```
/* Used for pushing arguments to the stack */
struct argument
{
  char* token;
  struct list_elem token_list_elem;
};

/* Where we store the arguments after parsing */
struct stack_data
{
  struct list argv;
  int argc;
};
```

Now we can point `stack_data` at the allocated page, initialise the list within it and then readjust the pointer to the beginning of the structure

```
// Allocate a page for thread data *for our args
char *tmp_page = palloc_get_page (0);
if (tmp_page == NULL)
{
  success = NULL;
  return success;
}

// Initialize that page for our args and make a ptr for it
memset(tmp_page,0,PGSIZE);
char* tmp_page_ptr = tmp_page;

struct stack_data* program_stack_data = NULL;
  program_stack_data = tmp_page_ptr;

// Stack data for this proccess pointing to allocated page
struct stack_data* tmp_stk_data = NULL;
tmp_stk_data = tmp_page_ptr;

/* Move pointer to location in page at top of the
   stack_data structure we are writing in */
tmp_page_ptr += sizeof(struct stack_data);
list_init(&program_stack_data->argv);
```

By using `strtok_r` we can tokenize our arguments until we don't have a tokenized string at the end. We can allocate and point to an argument data structure that just lets us go through a list of our arguments later. Also its important to check that our arguments haven't gone beyond the size of a page. Finally for each tokenized string in our long string of arguments we can increment the count of arguments as well as push each argument to front of the list.

```c
char* token, saveptr;

// Parse file and args
for (token = strtok_r (args_unparsed, " ", &saveptr);
        token != NULL;
        token = strtok_r (NULL, " ", &saveptr)
        )
{
  struct argument* args = NULL;
  args = (struct argument *)tmp_page_ptr;
  tmp_page_ptr += sizeof(struct argument);

  // If we go beyond our limit complain
  ASSERT(tmp_page_ptr - tmp_page <= PGSIZE);
  args->token=token;
  list_push_front(&tmp_stk_data->argv, &args->token_list_elem);
  tmp_stk_data->argc++;
}
```

2.1.4 Setup the Frame

According to PintOS documentation, we should be setting up the stack according to some a detailed below, minor changes were made to this design, such as placing a 32bit buffer at the base of the frame just in case something popped 1 too many times off the stack frame. Following this model we were able to push our arguments onto the stack so that our system calls could process a programs requests.

i.e `pintos -q run '/bin/ls -l foo bar'`

Address	Name	Data	Type
0xbffffffc	argv[3][...]	"bar\0"	char[4]
0xbffffff8	argv[2][...]	"foo\0"	char[4]
0xbffffff5	argv[1][...]	"-l\0"	char[3]
0xbfffffed	argv[0][...]	"/bin/ls\0"	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*)()

Place 32 bit buffer at base of frame.

```c
// Just put a 4 byte buffer at base;
*esp -= 4;
```

Push each tokenized argument onto the stack, counting the length + 1 for the null terminator.

```c
// Copy arg strings onto stack
for (e = list_begin (&program_stack_data->argv);
  e != list_end (&program_stack_data->argv);
  e = list_next (e))
  {
      struct argument *args = list_entry(e, struct argument,
       token_list_elem);
      char *indexed_arg = args->token;
      *esp -= (strlen(indexed_arg) + 1);

      strlcpy (*esp, indexed_arg, strlen(indexed_arg) + 1);
      args->token = *esp;
  }
```

Word align the data

```c
  // align data with 0 byte
  uint8_t alignment_byte = 0;
  *esp -= (sizeof(uint8_t));
  *(uint8_t *)*esp = alignment_byte;
```

One extra argument being set to `NULL` to follow the convention

```c
  // last arg, null pointer for convention *argv[argc] = NULL
  char *last_arg  = NULL;
  *esp-= (sizeof(char *));
  *(int32_t *)*esp = (int32_t)last_arg;
```

..Ok time to go back over each argument and push the pointer to each argument string within the frame onto the stack.

```
// Iterating over the same list pushing the ptrs to the arguments
// strings on the stack
for (e = list_begin (&program_stack_data->argv);
     e != list_end (&program_stack_data->argv);
     )
{
  struct argument *args = list_entry(e, struct argument,
  token_list_elem);
  char *indexed_arg = args->token;
  *esp -= (sizeof(char*));
  *(int32_t *)*esp = (int32_t)indexed_arg;
  e = list_next(e);
}
```

Few more things are remaining, the argument vectors pointer, the argument count ,and the return address which in our case isnt used so we fake it with just setting it to 0.

```
// push argument vector pointer onto stack
char **fst_arg_ptr = *esp;
*esp -= (sizeof(char **));
*(int32_t *)*esp = (int32_t)fst_arg_ptr;

// push argument count onto stack
*esp -=(sizeof(program_stack_data->argc));
*(int32_t *)*esp = program_stack_data->argc;

// fake the return addr in stack frame
void *fake_return_addr  = 0;
*esp -= (sizeof(void *));
*(int32_t *)*esp = (int32_t)fake_return_addr;
```

2.1.5 Result

Now that everything has been parsed and pushed onto the stack, we can dump our stack frame at the end of setting it up to see if it is all arranged correctly in memory.

i.e `pintos -q run 'echo hello world!'`

```
   -------HEX DUMP------
bfffffb0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
bfffffc0 00 00 00 00 00 00 00 00 00 00 00 00 00 aa aa aa |................|
bfffffd0 aa 03 00 00 00 d9 ff ff bf ea ff ff bf ef ff ff |................|
bfffffe0 bf f5 ff ff bf 00 00 00 00 00 65 63 68 6f 00 68 |..........echo.h|
bffffff0 65 6c 6c 6f 00 77 6f 72 6c 64 21 00 00 00 00 00 |ello.world!.....|
```

Bingo! Looks like everything is arranged correctly, we used a test return address of `0xaaaaaaaa` to help spot the end of the frame.

2.2 Application Name Extraction

In order to extract the executables file name for loading we do so in 2 locations, `process_execute` and `load` found in `userprog/process.c`. The first point of entry to `userprog` is `process_execute` where we initially made a copy of the arguments string to be processed later for the stack. We strip our the file name with `strtok_r` in order to pass it as the first argument to thread create, this will be helpful for our exit syscall, where we will need to print out the name of the program being exited.

2.2.1 process_execute

```c
tid_t
process_execute (const char* file_name)
{
  tid_t tid = TID_ERROR; // -1
  char* saveptr; // pointer for context on reentering strtok
  char* f_args; // file name along with args to be passed to thread_create
  char* fname; // file name to be stripped out

  /* Make a copy of FILE_NAME.
  Otherwise there's a race between the caller and load(). */
  f_args = palloc_get_page (0);

  if (f_args == NULL)
    return TID_ERROR;

  // separate page for passing in the stripped filename to thread_create
  fname = palloc_get_page(0);
  if(fname == NULL)
      return TID_ERROR;

  // load file and args string into copy ready for parsing
  strlcpy (fname, file_name, PGSIZE);
  strlcpy (f_args, fname, PGSIZE);
  thread_args = f_args;

  // strip out the filename
  fname = strtok_r(fname," ",&saveptr);

    ...

  // create the thread with filename and pointer to full argument string
  // will be parsed out later in setup_stack
  tid = thread_create (fname, PRI_DEFAULT, start_process, thread_args);

  ...

  // dont forget to free memory when thread_create fails
  if (tid == TID_ERROR)
  {
    palloc_free_page (f_args);
    palloc_free_page (fname);
    return tid;
  }
```

2.2.2 load

```c
// allocate memory for a copy of filename
char* f_namecpy = palloc_get_page(PAL_ZERO|PAL_ASSERT);

if(f_namecpy == NULL)
{
  printf ("load: cant allocate memory\n");
  goto done;
}

// load it into that page
strlcpy(f_namecpy,file_name,PGSIZE);

char* save_ptr;
f_namecpy = strtok_r(f_namecpy," ",&save_ptr);

/* Open executable file. */
file = filesys_open (f_namecpy);

if (file == NULL)
{
  printf ("load: %s: open failed\n", f_namecpy);
  goto done;
}

...

 done:

  // we cant allow program file to be changed
  if(success)
  {
    file_deny_write(file);
    t->cur_exec = file;
  }
  else
    file_close(file);

// dont forget to free the page
palloc_free_page(f_namecpy);

/* We arrive here whether the load is successful or not. */
return success;
```

2.2.3 Result

Now that we have applied the previously shown changes to `load` and `process_execute` we are able to pull the process name from the thread data structure as well as load any executable that is used as the first argument to `process_execute`.

Lets test inside `load` with a simple print statement..

```
printf("proc_name:%s\nfilename:%s\n\n",thread_current()->name, f_namecpy);
```

Now does it work?

```
pintos -q run 'echo hello'
...
proc name:echo
filename:echo
```

Horray!

3 System Calls

System calls are very important for an operating system to have, without the ability to handle any requests between a userland based process and the kernel pretty much any program would not be able to function. To give this functionality to our small operating system, we need to first develop a sycall handler, this is simple a bit of code that acts as a man in the middle between the syscall routines located in the kernel and the userland process. A syscall handler should also do any bounds checking on this information being passed in as to prevent crashes and secure the kernels space.

3.1 Syscall Handler

To develop our syscall handler, we need to be able to access the stack of our userland process, we do this by referencing the frame via `struct intr_frame`. If we look at this struct in `/threads/interrupt.h` we can see it has the current stack pointer made available along with other registers.

```
    /* Interrupt stack frame. */
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */
```

Using a pointer to this structure we can have a handler function access the stack pointer and pull out the syscall number for branching to various syscall routines.

```
static void syscall_handler (struct intr_frame *frame)
{
  // Grab stack pointer
  uint32_t *esp = (uint32_t*)frame->esp;

  // Is the Stack Pointer Valid?
  check_userland_addr(esp);

  // Grab syscall number
  int syscall_num = *esp;

  DEBUG_PRINT("\nCalling syscall:%d\n\n",syscall_num);

  // Check if its a valid syscall
  ASSERT(syscall_num < SYS_NUM_SYSCALLS);

  // Now call our syscall
  int arg[MAX_ARGS];

  // Now Branch
  switch(syscall_num)
  {
  ...
```

3.2 Bounds Checking

Since we are giving userland processes the ability to interact with the kernel, we can get into trouble pretty quickly. If something is not handled correctly in kernel context we will encounter a page fault and depending on the fault can crash our entire OS. Ideally we want to sanitize any input to kernel space from user space and check if pointers are accessing memory they should, along with checking access to any resources that a userland process shouldn't have access to. We also need to protect against a userland process trying to inject code and exploit the kernel or leverage itself to kernel context which would be a huge breach of security.

To do this we can just use a method of evaluating a pointer and a wrapper for this method to check range of this pointer which can be used for checking the bounds of a buffer.

```
// evaluate pointer into userland, if invalid exit the process
static void check_userland_addr (const void *ptr)
{

  // check if pointer is to somewhere and is in the userland address space
  if (ptr == NULL ||
      !is_user_vaddr(ptr) ||
      ptr >= PHYS_BASE ||
      pagedir_get_page(thread_current()->pagedir, ptr) == NULL
      )
  {
    // exit if address is not valid
    sys_exit(-1);
    NOT_REACHED();
  }

}


// evaluate useland buffer pointer by pointer
static void check_userland_buffer (void *buffer, size_t size)
{

  char* tmp_buf = (char *)buffer;

  for(size_t i = 0; i < size; i++)
  {
    check_userland_addr((void*)tmp_buf);
    tmp_buf++;
  }
}
```

3.3 Syscalls

Various syscalls are needed to be completed for basic input/output and file management to be conducted on PintOS via its test programs. There are 3 main categories of syscalls we need to implement: process management, file management, and file/standard I/O.

3.3.1 `sys_halt`

This is the simplest system call, it basically needs to shut the system down when its called.

Our handler case is pretty simple since it returns and takes no arguments.

```
case SYS_HALT: // 0
{
  sys_halt();
    break;
}
```

The code for this syscall is extremely simple...

```
void sys_halt(void)
{
  shutdown_power_off();
}
```

3.3.2 sys_exit

For our exit syscall, we need to pull the status that the process is exiting with, this status is an integer so we will cast the returned pointer to pointer to an int and assign the data to status. We simply call our syscall internally with this data.

```
case SYS_EXIT: // 1
{
  int status = (int)get_stack_argument(frame,0);

  sys_exit(status);
  break;
}
```

As discussed in chapter `2.2` we have assigned the thread its name via the stripped out file name. We are able to print out its name and status now that its exiting in order to help the user understand how well its programs are running. Lastly we need to update the thread with the exit status assigned so that `thread_exit` can determine what to do.

```
void sys_exit(int status)
{
  struct thread* cur = thread_current();

  // print name and status before exit
  printf("%s: exit(%d)\n", cur->name, status);

  // assign the thread the status
  cur->exit_status = status;

  thread_exit();
}
```

3.3.3 sys_exec

The purpose of sys_exec is to redirect to process_execute with a command string sent from a userland process. We also need to check if the pointer to this string is valid before we run away and use it. Also this is the first syscall we encounter that has some form of returned data, we can just assign eax of our current frame to be populated with this data, eax is a 32bit mode x86_64 arch register used for storing return values.

```
case SYS_EXEC: // 2
{
    const void* cmd = get_stack_argument(frame,0);

    check_userland_addr((void*)cmd);

    frame->eax = sys_exec(cmd);
    break;
}
```

The return value in execs case is the same as process_execute's which is the process id, essentially the same as the thread id in our case. We also developed a redefine of printf called DEBUG_PRINT which allows all prints used for debugging to be turned off in a single defines.

```
pid_t sys_exec(const char* cmd)
{
  DEBUG_PRINT("EXEC:%s\n",cmd);

  tid_t tid = process_execute(cmd);

  return tid;
}
```

3.3.4 sys_wait

Most of these sycalls sys_wait, sys_exit, sys_exec are just wrapper functions that call process related functions internally which do all the work, you can see much is the same in sys_wait, which just calls process_wait. We just need to make sure we load eax with the returned status.

```
case SYS_WAIT: // 3
{
    pid_t pid = (pid_t)get_stack_argument(frame,0);

    frame->eax = sys_wait(pid);
    break;
}

int sys_wait(pid_t pid)
{
  return process_wait(pid);
}
```

3.3.5 sys_create

Time for file system management syscalls, we start with sys_create which takes 2 arguments one the name of the file, and the size in which it should be allocated when creating. Due to the file name being a string, we will need to evaluate its pointer. It also returns an integer from filesys_create, we just need to push this to the eax register.

```
case SYS_CREATE: // 4
{
    const char* filename = (const char*)get_stack_argument(frame,0);
    unsigned size = (unsigned)get_stack_argument(frame,1);

    check_userland_addr((void*)filename);

    frame->eax = sys_create(filename, size);
    break;
}
```

Since the file system is a shared resource, we need to make sure access to it is synchronized to prevent corruption of data. We do this by using the semaphore allocated for it via filesys_lock and pass it our file name and size. After returning we need to make sure we release the lock so any other process can access the file system.

```
int sys_create(const char* filename, unsigned size)
{
  int ret = -1;

  lock_acquire(&filesys_lock);
  ret = filesys_create(filename, size);
  lock_release(&filesys_lock);

  return ret;
}
```

3.3.6 sys_remove

Essentially the same as `sys_create` but this time only a single argument of the file name is needed. We also need to evaluate its pointer.

```
case SYS_REMOVE: // 5
{
    const char* filename = (const char*)get_stack_argument(frame,0);
    check_userland_addr((void*)filename);

    frame->eax = sys_remove(filename);
    break;
}
```

The only difference here is to use a different file system call..

```
int sys_remove(const char* filename)
{
  int ret = -1;

  lock_acquire(&filesys_lock);
  ret = filesys_remove(filename);
  lock_release(&filesys_lock);

  return ret;
}
```

3.3.7 sys_open

Previously our file system syscalls have been fairly simple, however in order for our process to start performing I/O with its files things need to get a bit more complicated. To start I/O we need to implement the `open` syscall, this function should take a file name and return a file descriptor. The handler code is the same as other we have seen previously.

```
case SYS_OPEN: // 6
{

    const char* filename = (const char*)get_stack_argument(frame,0);
    check_userland_addr((void*)filename);

    frame->eax =  sys_open(filename);
    break;
}
```

The syscall code itself can be broken down into two parts, one open the file via its file name, and two store the pointer to its file structure in a list of the current processes open files. The second part needs a few prerequisites such as allocating file descriptors as well as creating and maintaining a list of files within our current thread. We can see this discussed in chapter 5, but for now we will assume this is already setup.

```
int sys_open(const char* filename)
{
  struct file *f;
  int status = -1;
  int fd;
  struct thread* cur_td = thread_current();


  // synch filesystem usage
  lock_acquire(&filesys_lock);

  // open file and get the struct file pointer of the file
  f = filesys_open(filename);

  // handle filesys open error
  if(f == NULL)
  {
    lock_release(&filesys_lock);
    return -1;
  }

  // time to add this file to our threads open files
  struct proc_fd_list* fd_list = palloc_get_page(0);

  // incase we cant allocate for our list
  if(!fd_list)
  {
    lock_release(&filesys_lock);
    return -1;
  }

  // grab the current threads file list
  struct list* td_file_list = &cur_td->thread_file_list;

  // add our new fd and file * to our processes list
  fd_list->fd = new_fd();
  fd_list->file = f;
  file_deny_write(f);
```

```
  // now add the elem from our list to the threads list
  list_push_back(td_file_list, &(fd_list->elem));

  fd = fd_list->fd;

  lock_release (&filesys_lock);

  return fd;
}
```

3.3.8 sys_read

The read syscall takes 3 arguments, the file descriptor, the buffer in which to point and pull data from in memory, and the size which is the length in which to iterate on the buffer to. We need to use our wrapper for checking each pointed to memory location in the range that size sets from the start of the buffer. sys_read also returns the length of data written in number of bytes.

```
case SYS_READ: // 8
  {
    int fd = (int)get_stack_argument(frame,0);
    char *buffer = (const char*)get_stack_argument(frame,1);
    unsigned size = (unsigned)get_stack_argument(frame,2);

    check_userland_buffer(buffer,size);

    frame->eax = sys_read(fd, buffer, size);
    break;
  }
```

For the dealing with `sys_read` internally we need to handle both file I/O and standard I/O, we do this by checking the file descriptor to see if it matches `STDOUT` or `STDERR` which our implementation shouldn't do anything with so it just returns `-1` for an error. However if we are reading from `STDIN` we need to grab the character from terminal input via `input_getc` and load it into the buffer, we increment the buffers pointer just to handle overrun. Immediately after we return 1 since we should have only read one character. This functionality is essential for things like `scanf` to work via a shell program for example.

Handling normal files in read is pretty similar to other file system syscalls we have done, we just need to add a method to retrieve the file via its file descriptor and call a file system function internally with the pointer to this file as well as the passed in buffer and size.

```c
int sys_read(int fd, char* buffer, unsigned size)
{
  int ret = -1;

  // why do anything
  if(size < 1)return 0;

  // stdio handling
  if(fd == STDOUT_FILENO || fd == STDERR_FILENO)return ret;

  if(fd == STDIN_FILENO)
  {
    *(buffer++) = input_getc();
    return 1;
  }

  // normal files
  struct file *f;

  lock_acquire(&filesys_lock);

    struct proc_fd_list* proc_file = get_process_file(fd);

    if(proc_file != NULL)
      ret = file_read(proc_file->file, buffer, size);

  lock_release (&filesys_lock);

  return ret;
}
```

3.3.9 sys_write

Essentially the write syscall is the same as the read syscall, but with a few few inverted operations. It takes the same arguments as read so the handler code is pretty much identical.

```
case SYS_WRITE: // 9
  {

    int fd = (int)get_stack_argument(frame,0);
    const void *buffer = (const void*)get_stack_argument(frame,1);
    unsigned size = (unsigned)get_stack_argument(frame,2);

    check_userland_buffer(buffer,size);

    frame->eax = sys_write(fd, buffer, size);
    break;
  }
```

When implementing the code internally, we just need to swap `STDIN` for `STDOUT` and instead of `input_getc` we use `putbuf` to display on screen. Also its not based on a single character and will display the entire size of the buffer, so we return that as well. For writing to files the code is also the same but just with a different file system call, although we have added a protection on writing to the file before and after it has been accessed here. This should protect improper writes that would potentially corrupt files.

```
int sys_write(int fd, const void* buffer, unsigned size)
{
  int ret = -1;

  // why do anything
  if(size < 1)return 0;

  // stdio handling
  if(fd == STDIN_FILENO || fd == STDERR_FILENO)return;

  if(fd == STDOUT_FILENO)
  {
    putbuf(buffer,size);
    return size;
  }

  // normal files
  struct file *f;

  lock_acquire(&filesys_lock);

  struct proc_fd_list* proc_file = get_process_file(fd);

  if(proc_file != NULL)
  {
    file_allow_write(proc_file->file);
    ret = file_write(proc_file->file, buffer, size);
    file_deny_write(proc_file->file);
  }

  lock_release (&filesys_lock);

  return ret;
}
```

3.3.10 `sys_seek`

The seek syscall is much like the other file management syscalls in terms of simplicity to implement, it takes 2 arguments of a file descriptor and position into the file and is a void return so we never push any data to `eax`.

```
case SYS_SEEK: // 10
{
  int fd = (int)get_stack_argument(frame,0);
  unsigned pos = (unsigned)get_stack_argument(frame,1);

  sys_seek(fd, pos);
  break;
}
```

Our internal code just grabs the file pointer from our list of file descriptors in our thread, and checks to see if it has returned it properly, if so it should close the program with `sys_exit`. Else we can use the semaphore from the file system and call the file systems seek function to perform the operation.

```
void sys_seek(int fd, unsigned pos)
{
  struct proc_fd_list* proc_file = get_process_file(fd);

  if(proc_file == NULL || proc_file->file == NULL)
    sys_exit(-1);

  lock_acquire(&filesys_lock);
  file_seek(proc_file->file,pos);
  lock_release(&filesys_lock);
}
```

3.3.11 sys_tell

The tell syscall is paired with the seek syscall in that it will return the current position in a file. Its only argument is the file descriptor and it should return a 32bit signed number which we load our return register `eax` with.

```
case SYS_TELL: // 11
{
  int fd = (int)get_stack_argument(frame,0);

  frame->eax = sys_tell(fd);
  break;
}
```

In our code, we set the offset to `-1` just in case it fails to return a correct index, `-1` being an indicator that an error has occurred. We also need to make sure that seek did to kill the program if we cant find the file, before actually calling `file_tell` on the file. Finally returning the index into the file from `file_tell`.

```
int32_t sys_tell(int fd)
{
  int32_t cur_offset = -1;
  struct proc_fd_list* proc_file = get_process_file(fd);

  if(proc_file == NULL || proc_file->file == NULL)
    sys_exit(-1);

  lock_acquire(&filesys_lock);
  cur_offset = file_tell(proc_file->file);
  lock_release(&filesys_lock);

  return cur_offset;
}
```

3.3.12 sys_close

Finally `sys_close` our last syscall to be implemented, does much of the same that open has done, but will clean up memory as well as remove the file from our threads list of open files. The handler code for it is even simpler with a single argument of a file descriptor as well as the return of an integer to gauge whether it closed correctly or not.

```
case SYS_CLOSE: // 12
{
  int fd = (int)get_stack_argument(frame,0);

  frame->eax = sys_close(fd);
  break;
}
```

In our code we need to access a few things, like the current thread, as well as grab the pointer to the file via our helper function `get_process_file`. Once retrieved we can evaluate if it exists, if not we kill the program. We also need to check to see if the file descriptor is a standard I/O or beyond the currently allocated file descriptors, if it is then we need to return and error. All that is remaining to do is to acquire the semaphore for the file system and close the file before freeing our memory and removing the file from our current list of files.

```
int sys_close(int fd)
{
  struct thread* cur = thread_current();

  if(fd < 3 || cur->fd_next >= fd)return -1;

  struct proc_fd_list* proc_file = get_process_file(fd);

  if(proc_file == NULL || proc_file->file == NULL)
    sys_exit(-1);

  lock_acquire(&filesys_lock);
  file_allow_write(proc_file->file);
  file_close(proc_file->file);

  list_remove(&(proc_file->elem));
  palloc_free_page(proc_file);
  lock_release(&filesys_lock);

  return 0;
}
```

4 Process Management

As seen in our syscalls `sys_exit`, `sys_exec`, `sys_wait` we redirect into `process.c` or order to handle these operations. For us to implement their functionality, we need to modify some of threads data structure in `thread.h`. As seen below we add various information about a thread and its children, such as statuses: alive, exiting, waiting, child loaded. Also information like the condition which is a semaphore used for controlling whether a thread is waiting or not. We also need a way of accessing the parent and children thread structures of a thread to control execution a bit more.

4.1 sys_exec

The changes in this section allow for the `sys_exec` syscalls functionality to be implemented.

4.1.1 thread.h

```
#ifdef USERPROG
/* Owned by userprog/process.c. */
struct list thread_file_list;   /* List of threads opened files */
uint32_t *pagedir;              /* Page directory. */
struct thread* parent;          /* Parent of the thread */
struct list_elem child;         /* Element of parents list of children */
struct list children;           /* Children of the thread */
int exiting;                    /* Is thread exiting */
int alive;                      /* Is thread alive */
int waiting;                    /* Is thread waiting */
int exit_status;                /* Return or exit code from thread */
int child_loaded;               /* Has child of this thread been loaded */
struct file* cur_exec;          /* Pointer to current executeable file */
struct semaphore condition;     /* Semaphore for thread run control */
struct list children_return;    /* Children statuses */
tid_t parent_tid;               /* Thread id of parent thread */
struct lock plock;              /* process lock for while thread is alive */
int fd_next;                    /* Current index on thread file descriptor */
struct stack_data* stack_data;  /* Stack data used for this thread,
 before pushing to stack */
#endif
```

4.1.2 thread.c - init

We just need to make sure we intialise some of this data in side `init_thread` inside `thread.c`..

```
// Our lists/semaphores/setup
list_init(&t->children);
list_init(&t->thread_file_list);
sema_init(&t->condition,0);
t->exiting = NOT_EXITING;
```

4.1.3 process_execute

Now that we have initialised some lists and statuses we need for process control in `init_thread` we can follow the control path in starting a process and see where other locations of using this data is necessary. If we recall back to the beginning of chapter 2 we looked at how things got executed in `process.c`

```
process_execute -> start_process -> load -> setup_stack
```

Using this we can locate some of these locations, starting within `process_execute`.

```
  struct thread* td = thread_current();

  /* Create a new thread to execute FILE_NAME. */
  intr_disable();

  // create the thread with filename and pointer to full argument string
  // will be parsed out later in setup_stack
  tid = thread_create (fname, PRI_DEFAULT, start_process, thread_args);
  thread_block();
  intr_enable();

  // return if thread fails to load after creation
  if(td->child_loaded != CHILD_LOADED)
    tid = TID_ERROR;

  // dont forget to free memory when thread_create fails
  if (tid == TID_ERROR)
  {
    palloc_free_page (f_args);
    palloc_free_page (fname);
    return tid;
  }

  // setup thread info
  struct thread* child = get_thread(tid);

  child->alive = ALIVE;
  child->waiting = NOT_WAITING;
  child->fd_next = INITIAL_FD;

  lock_init(&child->plock);

  return tid;
}
```

Towards the end of `process_execute` we have grabbed the thread structure for the created thread if it does return properly. if not we free the memory used and return the error. At the bottom we can see we need to say the thread we just launched is alive, its not been waited on, and we initialise its next available file descriptor at 3. Its also important to initialise the lock it uses for accessing some of its data.

4.1.4 `start_process`

The next location we need to fix is inside `start_process`. After loading the file we need to reference the current thread and tell its parent thread that its not waiting, we can unblock the parent now that the child has been attempted at being loaded. If it fails to load we can kill the child thread which the parent will know that it is not loaded by default when the child dies. However it it doesnt fail it will hit code that tells the parent thread that the child has been loaded into memory.

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

// Tell the parent about our loaded child
cur->parent->waiting = NOT_WAITING;
thread_unblock(cur->parent);

/* If load failed, quit. */
if (!success)
  sys_exit(ERROR);

// Yay child was loaded
cur->parent->child_loaded = CHILD_LOADED;
```

This code now covers the setup for a process's thread info when starting a process such as when `sys_exec` is called. Now we need to look at the process handling for the wait syscall and its functionality.

4.2 sys_wait

Changes detailed in this section allow for the `sys_wait` syscall to function and keep the process alive while it hasn't been exited or needs to be waited for a period of time.

4.2.1 process_wait

By removing the infinite loop present in `process_wait` we will end up terminating the process before it ever gets to run. To get it actually checking for different thread status and working properly we need to have a method of getting the parent thread and child thread via its id passed into this function. Our `get_thread` function handles this for us by returning a pointer to a threads data structure by using the thread id and internally accessing a list of child threads made available to the current thread.

```
 // Get thread from thread id
struct thread* get_thread(tid_t tid)
{
  // Catch if threa creation failed
  ASSERT(tid != TID_ERROR);

  struct list_elem* e;
  struct thread* td;

  for(e = list_begin(&all_list);
      e != list_end(&all_list);
      e = list_next(e))
  {
    td = list_entry(e, struct thread, allelem);

    // Found it!
    if(td->tid == tid && td->status != THREAD_DYING)
    {
      return td;
    }
  }
  return NULL;
}
```

With the ability to grab both parent and child thread, we can check to see if it has been waited on already, if so we aren't able of telling it to wait again so we just return and error, we do the same if the thread doesn't exist. This code only can trigger a thread to wait if its alive as well. We need a method to detect when its blocked or exiting, in which the wait syscall will return the exit status of the thread instead of actually waiting on it. If everything checks out and the thread is alive and well but needs to wait, we can set its parents semaphore to down which will wait for it to be set to a positive value later on. Other methods could have been used here such as a lock, but `sema_up` and `sema_down` were easier to implement. At the end we need to remember to return the child threads exit code just in case we haven't hit a return already.

```
int
process_wait (tid_t child_tid)
{
  if(child_tid == TID_ERROR)
    return ERROR;

  struct thread* parent = thread_current();
  struct thread* child = get_thread(child_tid);
  int status = ERROR;

  if(child == NULL || child->waiting)
    return status;

  if(child->alive)
  {
    child->waiting = WAITING;

    if(child->exiting == EXITING && child->status == THREAD_BLOCKED)
      return child->exit_status;

    // wait for the condition to be set to positive
    sema_down(&parent->condition);
  }
  return child->exit_status;
}
```

4.3 sys_exit

Code changes in this section help `sys_exit` properly close down a thread and clean up its resources along with signal to wait that a thread is dying.

4.3.1 thread_exit

The first change to get exit code working properly is to modify code in `thead_exit` since its whats called first from `sys_exit`. To get it working we need to signal that the thread exiting is now dead and not waiting anymore, aside from that we can remove its resources but call `process_exit` before hand to handle the userland side of our process dying.

```
void
thread_exit (void)
{
  ASSERT (!intr_context ());

#ifdef USERPROG
  process_exit ();
#endif

  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
  intr_disable ();

  // clean up files and resources
  struct thread* td = thread_current();
  td->alive = DEAD;
  td->waiting = NOT_WAITING;

  struct file* cur_exec = td->cur_exec;
  if(cur_exec)
  {
    file_allow_write(cur_exec);
    file_close(cur_exec);
  }

  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();

  NOT_REACHED ();
}
```

4.3.2 process_exit

The needed changes for process_exit are very simple, we need to set our current threads exit status to be that of exiting, and also after we get rid of its memory resources we need to check for a specific case. This case is when the thread is the main thread, which is the first process loaded via PintOS, if it has been told to exit we simple tell the parent condition to become positive and ours to become negative, telling it essentially to wait. Without this code when our thread exits, PintOS will crash due to it trying to exit without keeping its main program alive long enough to handle closing out.

```
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  // we are exiting now
  cur->exiting = EXITING;

    /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }

  // if we are the root thread then wait
  if(strcmp(cur->name, "main") != 0)
  {
    sema_up(&(cur->parent)->condition);
    sema_down(&cur->condition);
  }
}
```

4.4 Result

Now with everything in place we should be able to execute a program like echo in PintOS and have it cleanly exit and close down for us if we give it the option too.

```
pintos -q run 'echo hello world!'
```

```
Loading............
Kernel command line: -q run 'echo hello world!'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 419,020,800 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 193 sectors (96 kB), Pintos OS kernel (20)
hdb: 21,168 sectors (10 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 20,480 sectors (10 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo hello world!':
echo hello world!
echo: exit(0)
Execution of 'echo hello world!' complete.
Timer: 70 ticks
Thread: 1 idle ticks, 69 kernel ticks, 0 user ticks
hdb1 (filesys): 38 reads, 0 writes
Console: 683 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Horray! We just executed, waited, and exited from program in PintOS!

5 File Management

One of the things briefly brushed upon in our file I/O syscall implementations was how we managed to store a list of and access our current files being used by a process. To do so we needed a data structure for this information about a file such as its pointer, a file descriptor allocated, and a way to index it in a list. We need a way to set up the next available file descriptor for a thread, as well as a way to get a pointer for this file via a file descriptor like needed for syscalls like read, write, and close.

5.1 process.h

To start we create a simple data structure in `process.h` but it really could of been anywhere that was visible to us in userprog.

```
/* List of open files in a process */
struct proc_fd_list{
  int fd;                  // file descriptor for the file
  struct file* file;       // pointer to the file in fs
  struct list_elem elem;   // element for access in a list
};
```

5.2 get_process_file

We wrote a simple function that used a list we had in `thread.h` that gets pushed to every time a file is opened. When we need to access a file, we simply locate this list in our current thread, and check to see if it exits first. If it does, we can go through our list and see if any of the file descriptors match the one we are looking for, if not we simply return nothing, but if we do find a match, we return a pointer to this list entry which contains its file descriptor and file pointer that we can use in our syscalls.

```
// Get a list entry of a file via its file descriptor
struct proc_fd_list* get_process_file(int fd)
{
  struct thread* cur_td = thread_current();
  struct list_elem* e;

  if(list_empty(&cur_td->thread_file_list))
  {
    return NULL;
  }
  else
  {
    for(e = list_begin(&cur_td->thread_file_list);
        e != list_end(&cur_td->thread_file_list);
        e = list_next(e))
    {
      struct proc_fd_list* proc_file = list_entry(e,
       struct proc_fd_list, elem);

      if(proc_file->fd == fd)
      {
        return proc_file;
      }
    }
    return NULL;
  }
}
```

5.3 new_fd

Whenever we open up a file, we need to allocate it a new file descriptor. We can just do this with a function that when called returns an incremented file descriptor from the threads data structure. To prevent our file descriptor from being a standard I/O reserved number, we simple set `fd_next` to be 3 when we create its thread.

```
// generates a new file descriptor reserving 0,1,2 for stdio
static int new_fd()
{
  // update thread +1 after return
  return thread_current()->fd_next++;
}
```