PINTOS PROJECT 2 DOCUMENTATION

Group 26

Contents

Implementation by Patrick Robinson, Documentation by James Tuck.

Pintos assignment (project 2) introduction

The purpose of the 2nd pintos project (which is what our assignment is based off) is to allow user programs access to kernel features through system calls.

To run a program in pintos you use the command: pintos -r “program name” in it’s current state pintos will not be able to accept any arguments which is why the first part of project two is to implement argument passing, then after we have implemented argument passing we will be able to start implementing system calls.

Argument passing and setting up stack

In it’s current state pintos will not allow for argument passing meaning 12 of the 13 system calls that are to be implemented in this assignment (the only one that will work is halt) meaning that we will have to implement argument passing before we can consider starting to work on system calls.

We need to take a couple of things into consideration before we can start on argument passing, these are:

There are a few parts of the process_activate function located in process.c that we are going to look at that allow us to implement argument passing

The first addition to process_activate is shown below:

```
int total_length = 0;
char word_align = 0;
char * argvAddr[argc];
```

The reason that we added this is into the function is because we’ll need to keep track of the amount of bytes used when storing the arguments in the stack as we will have to pad it out to 4 bytes using 0s if it isn’t already 4 bytes long.

```
for (int i = (int)argc; i >= 0; i--)
  {
    if (argv[i] != NULL){

      int param_length = strlen(argv[i]) + 1;
       *esp -= param_length;

      memcpy(*esp, argv[i], param_length);
       total_length += param_length;
      argvAddr[i] = *esp;
      }
  }
```

The section that is shown above is used to write each argument in reverse, this is needed because in the pintos stack the least significant bit is stored in the smallest memory address.

```
if (total_length > 0) {
  word_align = 4 - (total_length % 4);
  }
  *esp -= word_align;
  memset(*esp, 0, word_align);
```

the section of the code that's showed above is how we added is used for padding with zeros so arguments would consist of 4 bytes when added to the stack

```
for (int i = (int)argc; i >= 0; i--)
    {
      if (argv[i] != NULL){

        *esp -= sizeof(char*);
        memcpy(*esp, &argvAddr[i], sizeof(char*));
        }
    }
```

The for loop that's shown above is how we wrote the address that point to each of the arguments as char*s

```
  //write the address of argv, type char**
  char ** a = *esp;
  *esp -= (char)4;
  memcpy(*esp, &a, sizeof(char**));

  //write the number of args, taking up 4 bytes
  *esp -= sizeof(uint32_t);
  memset(*esp, argc, sizeof(char));

  //write a null pointer for return
  *esp -=  sizeof(void*);
  void * nullPtr = NULL;
  memcpy(*esp, &nullPtr, sizeof(void*));
               }
       }  else {

       palloc_free_page (kpage);

       }
```

In this snippet of code we achieve a couple of things, the first thing that we achieve is storing the address of argv as a char**, we then write the number of arguments and write a null pointer to return.

System calls

What are system calls?

System calls allow for user programs to access and use kernel features without giving any access to the kernel, this is important because it helps prevents a kernel panic which is implemented as a security measure to stop user programs from accessing the kernel.

What do we have to do before we can start implementing system calls?

The main problem that we must solve before implementing system calls is implementing argument passing. How we plan to approach this, and our actual implementation will be in the argument passing section of this document. We also have to implement a working syscall handler, which will call the correct syscall depending on the syscall code that the user program gives us. To allow for us to use the syscall handler to call the correct syscall we need the syscall codes to be defined somewhere, we did not have to do this ourselves as they are already defined in the file "lib/`syscall-nr.h" as shown below.`

```
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. */

  };
```

There is also the system codes for the project three and four system calls in this file but I haven't included these in the snippet as we don't have to implement these in this assignment. Because no values have been manually assigned to the members of the enum this means values will be assigned automatically starting at 0 (SYS_HALT will equal 0, SYS_EXIT will equal 1, SYS_EXEC will equal 2 and so on)

We also didn't have to make function prototypes in syscall.c because they were already included in lib/user/syscall.h as seen below:

```
void halt (void) NO_RETURN;
void exit (int status) NO_RETURN;
pid_t exec (const char *file);
int wait (pid_t);
bool create (const char *file, unsigned initial_size);
bool remove (const char *file);
int open (const char *file);
int filesize (int fd);
int read (int fd, void *buffer, unsigned length);
int write (int fd, const void *buffer, unsigned length);
void seek (int fd, unsigned position);
unsigned tell (int fd);
void close (int fd);
```

Now we have done everything we need to get system calls working we can start implementing them and this section of the documentation will explain what each system call does and how we planned and implemented them.

SYS_HALT:

Return type: void

Passed Variables: none

Description: halt shuts down pintos by calling the shut_down_power_off() function

Pseudo code:

If the system call code is SYS_HALT:

Then Shut down pintos system

Built in pintos functions used:

shutdown_power_off() (all that needs to be called or done to implement this syscall)

Implementation:

```
case SYS_HALT:
    {

      shutdown_power_off();
      break;
    }
```

SYS_EXIT:

Return type: void

Passed Variables: int status

Description: Exit is used to terminate the current user program, a status of zero means exit has been successful, if status is a number that isn't zero then there's been error.

Design:

Pseudo code:

Get current thread

Set the current thread's exit code to status

Exit thread

Built in pintos functions used:

thread_exit() and thread_current()

Implementation:

```
case SYS_EXIT:
    {
int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;
    copyArgs(esp, argv, argc);
    exit(argv[0]);
 }
```

```
void exit(int status)
{
        struct thread* cur =  thread_current();
        cur->exitcode = status;
        thread_exit();

}
```

SYS_EXEC

Return type: pid_t

Passed variables: const char *cmd_line

Description: exec runs the program that is given in cmd_line and returns the new processes process ID, exec should return -1 if the program that is given in the command line can not be executed.

Design:

Pseudo code:

Get the program name to be executed from the command line

Create a process id

Execute the program that was received from the command line and give it the process ID that was previously created

Return the process ID onto the frame

Implementation:

We made some changes to the file userprog/process.c aswell as userprog/syscall.c

We added the following struct named process_data to process.c to assist us with process management.

```
struct process_data {


  const char* cmdline;

  tid_t pid;

  struct thread * parent_thread;

  struct semaphore * sema_exec;

  struct semaphore init_semaphore;
  int exitcode;

};
```

Then we also made changes to the process_execute(const char *file_name) function to add the functionality needed to have a properly working execute system call. The original process_execute can be seen below.

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);


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

  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

There is a couple of problems with this implementation of process_execute that means it doesn't always return the correct value when SYS_EXEC is called. An example is SYS_EXEC should return -1

Another issue with this implementation is that it may return a value before the program that needs to be executed is even loaded.

Our new version of this function can be seen below with changes shown in red

```
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;
  struct thread *cur = thread_current();
  struct process_data * newPData = NULL;


   newPData = palloc_get_page(0);
  if (newPData == NULL) {
    return -1;
  }

  newPData->parent_thread = cur;
  newPData->cmdline = file_name;

  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);
  tid = thread_create (file_name, PRI_DEFAULT, start_process, newPData);

  newPData->pid = tid;

  sema_init (&newPData->sema_exec, 0);
  sema_down(&newPData->sema_exec);

  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

The implementation of execute in the file syscall.c is as seen below.

```
case SYS_EXEC:
    {
        const char * cmd_line = *((int*)f->esp + 1);
        pid_t pid;
        pid = (pid_t)process_execute(cmd_line);
         f->eax = pid;

        break;
    }
```

As our changes in execute show we ran into a couple of problems while implementing execute, we originally assumed that process_execute would fully fill the requirements without any changes. But it quickly became apparent that not making any changes to process execute would a couple of issues

SYS_WAIT

Return type: int

Passed variables: pid_t pid

Description: Wait is used to wait for a child process and return's the child processes exit code, however in certain circumstances it must return -1, these include: pid is doesn't call exit() and is terminated by the pintos kernel; pid is not a direct child of the calling process or the child process is already being waited on by the parent process.

Design:

Pseudo code:

If pid is not a direct child of the calling process

Or If the child process is already being waited on by the parent process

Or if the child process doesn't call exit and is terminated by the pintos kernel

Then return -1

Else wait for child process to exit and return the exit code of the child process.

We weren't able to get a working implementation of wait so check the "what were we not able to get working section" to see our current implementation aswell as the issues that we had with this implementation.

SYS_CREATE

Return type: boolean

Passed variables: const char *file, unsigned initial_size

Description: Create creates a file that has the name file (the variable that's passed into the system call)and sets it's size to initial_size bytes. If this successful SYS_CREATE will return true, otherwise it will return false. Another factor to take into consideration is that create should not open the file after it's been created

Pseudo code:

Create a file using filesys_create

Check if it's successful by setting a boolean variable to the return value of filesys_create.

Return true if the file was created successfully

Or return false if the file wasn't created

Implementation

```
case SYS_CREATE:
  {
    int argc = 2;
    uint32_t argv[argc];
    void *esp = f->esp;
    copyArgs(esp, argv, argc);
    f->eax = create((char*)&argv[0], argv[1]);
  }
```

```
bool create(const char* file, unsigned initialSize)
{
bool success = filesys_create (file, initialSize);
return success;
}
```

SYS_REMOVE

Return Type: boolean

Passed Arguments : const char *file

Description: Remove deletes the file that's called file(the variable that's passed into the system call). If this successful SYS_REMOVE will return true, otherwise it will return false. Remove should be able to delete an open or closed file. If remove does delete an open file that file should remain open, the way this works is that pintos allows for threads that have a file open to keep it open until the thread is exited.

Implementation:

```
case SYS_REMOVE:
  {
    int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;

    copyArgs(esp, argv, argc);
    f->eax = remove((char*)&argv[0]);
  }
```

```
bool remove(const char* file)
{
  bool success = filesys_remove(file);

  return success;
}
```

SYS_OPEN

Return Type: integer

Passed variables: const char *file

Description: opens the file named file (the argument that's passed into this system call), this should return a positive integer handler called file descriptor. If the file is unable to be opened for whatever reason open should return -1

Pseudo code:

get a file descriptor by opening a file through filesys open and

Implementation:

```
case SYS_OPEN:
  {
    int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;
    copyArgs(esp, argv, argc);

    f->eax = open((char*)argv[0]);   }
```

```
int open(const char* file)
{
  int fd = (int)filesys_open(file);
  thread_current()->fileTable[fd] = file;
  return fd;
}
```

SYS_FILESIZE:

Description: The purpose of file size is self-explanatory, it's used to return the size of a given file.

Pseudo code:

Look up the file with the file descriptor FD using the current threads file table

Then use the kernel function file_length(file) to determine the length of the file that has the passed in file descriptor.

Implementation:

```
case SYS_READ:
  {
    int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;
    copyArgs(esp, argv, argc);

    f->eax = filesize(argv[0]));
 }
```

```
int filesize(int fd)
{
  const char* file = thread_current()->fileTable[fd];
  int length = (int)file_length(file);
  return length;
}
```

SYS_READ

Return type: int

Passed arguements: int fd, void *buffer, unsigned size

Description: Read reads the bytes from the file that is open as fd into a buffer, it then reads the number of byes that were actually read for example it would return 0 if it was at the end of the file or it would return -1 if it couldn't read any bytes for another reason

pseudo code:

Look up the file with the file descriptor FD using the current threads file table

Set a value called bytes read and set it to the return value of file_read

Implementation:

```
case SYS_READ:
  {
    int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;
    copyArgs(esp, argv, argc);

    f->eax = read(argv[0], argv[1], argv[2]);
  }
```

```
int read(int fd, void* buffer, unsigned length)
{

  const char* file = thread_current()->fileTable[fd];
  int bytesRead = (int)file_read(file, buffer, length);
  return bytesRead;
}
```

SYS_WRITE

Return type: int

Passed variables: int fd, void *buffer, unsigned size

Description: write bytes from buffer to the open file. In a normal operating system (or pintos in later projects) writing past the end of the file would extend the file size but because the basic pintos file system doesn't support changing a files size after it's created, how write should handle this is that it should write as many bytes as possible before reaching the end of the file and then return the actual number written.

Implementation:

```
case SYS_WRITE:
  {
    int argc = 3;
    uint32_t argv[argc];
    void *esp = f->esp;
    copyArgs(esp, argv, argc);
    f->eax = write(argv[0], argv[1], argv[2]);
    }
```

```
int write(int fd, const void* buffer, unsigned size)
{

  const char* file = thread_current()->fileTable[fd];
  int bytesWritten= 0;

  if(buffer>=PHYS_BASE){exit(-1);}
  if(fd<0||fd>=128){exit(-1);}
  if(fd==0){exit(-1);}
  if(fd==1)
  {
    int a=(int)size;
    while(a>=100)
    {
      putbuf(buffer,100);
      buffer=buffer+100;
      a-=100;
    }

    putbuf(buffer,a);
    bytesWritten=(int)size;

  } else {
      if(thread_current()->fileTable[fd]==NULL)
        {
       printf("file not found");
       bytesWritten =-1;
       } else {
       bytesWritten =file_write(file, buffer, (uint32_t)size);
        }
  }

  return bytesWritten;
}
```

The first if statement (if(buffer>=PHYS_BASE)) is used to check if the buffer is larger than PHYS_BASE and if it is the function will then return -1, if this if statement is false write then the system call also then also validate that the file descriptor is valid.

SYS_SEEK

Return type: void

Passed variables: int fd, unsigned position

Description: seek changes the next byte that's to be written or read to the byte at the argument position (position is an unsinged integer and a position of 0 refers to the start of the file)

Implementation:

```
case SYS_SEEK:
  {
    int argc = 2;
    uint32_t argv[argc];
    void *esp = f->esp;

    copyArgs(esp, argv, argc);
    seek(argv[0], argv[1]);
  }
```

```
void seek(int fd, unsigned position)
{
  const char* file = thread_current()->fileTable[fd];
  file_seek (file, position);
}
```

SYS_TELL

Return type: unsinged

Passed argument : int fd

Description: Tell returns the position of the next byte in the open file with the file descriptor that's passed into tell that needs to be either read or write.

Implementation:

```
case SYS_TELL:
  {
    int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;

    copyArgs(esp, argv, argc);
    tell(argv[0]);
  }
```

```
unsigned tell(int fd)
{
 const char* file = thread_current()->fileTable[fd];
  file_tell (file);
}
```

SYS_CLOSE

Return type: void

Passed variables: int fd

Description: closes the file that has the file descriptor fd which is passed into the close function.

Pseudo code:

Find file with the file descriptor fd

When found close file

Get file that has the file descriptor that has been passed in.

When found close the file.

Implementation:

```
case SYS_CLOSE:
    {
    int argc = 1;
    uint32_t argv[argc];
    void *esp = f->esp;

    copyArgs(esp, argv, argc);
    close(argv[0]);
    }
```

```
void close(int fd)
{
  const char* file = thread_current()->fileTable[fd];
  file_close (file);
}
```

12 of these system calls can be split into two different categories, these are:

Filesystem: remove, create, open, file size, read, write, seek, tell and close.

Process/ thread control: exec, exit and wait.

The only system call that does not fit into either of these two categories is halt.

Additional functions

We created a few extra functions that helped us implement the system calls.

These are:

CopyArgs:

Description:

The original function calls memcpy and passes in the following arguments: void *esp, uint32_t *argv and int argc. This could have been done without this additional function but we felt it was easier to understand what was happening if we used a function with a name where we could easily see what was happening, and it wouldn't really cause any impact on runtime performance.

Our first Implementation which didn't include any validation is shown below:

```
void copyArgs(void *esp, uint32_t *argv, int argc)
{
memcpy(argv, esp + 4, argc * 4);
}
```

We decided to change this to add some validation, the new version that will be in our final version of pintos is shown below:

```
bool copyArgs(void *esp, uint32_t *argv, int argc)
{
        bool success = true;
        for (int i = 0; i < argc; i++)
        {

                success = check_user(esp+4, argv, 4);

        }
        memcpy(argv, esp + 4, argc * 4);
        return success;
}
```

In this new version copyargs returns a boolean value depending on weather it was successful or not. It also stops pintos from crashing when a system call is given a bad pointer.

A bad pointer can either be NULL or pointers to kernel memory.

Checkuser

Description: Reads a consecutive `bytes` bytes of user memory with the starting address `src` (uaddr), and writes to dst. It also Returns the number of bytes read. In case of invalid memory access, exit() is called and consequently the process is terminated with return code -1.

Implementation:

```
bool
check_user (void *esp, int argc)
{
  int32_t ptr;

  for(int i=0; i<argc; i++) {
    ptr = get_user(esp + (i*4));
    if(ptr == -1) // segfault occured
    {
      printf("DEBUG bad pointer");
      return false;
    }

  return true;
}
}
```

Get user:

Description: Reads a byte at user virtual address UADDR the byte must be memory that is mapped by the MMU

the first part of input validation ensures it is below PHYS_BASE. Get user Returns the byte value if successful or it returns -1 if a segfault occurred.

Implementation:

```
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}
```

File system

Using the filesystem

To allow us to test our programs we have to copy our test programs into the pintos file system. This can be done with the following steps:

- If the filesystem hasn't been made yet then we have to make it using the following command "from userprog/build : pintos-mkdisk filesys.dsk --filesys-size=2"
-

File system Limitations

The pintos file system has multiple limitations when compared to operating systems such as windows and linux. These limitations would be fixed in project 4 but because we are only implementing project 2 of pintos we'll have to keep the following limitations in mind:

Data in a singular file must occupy a singular range of sectors on the virtual hard-disk this means that unlike windows/linux where a larger file would be split into chunks a file in pintos must occupy one chunk.

The size of a file is also limited at the time of creation.

The length of a file name is limited to 14 characters.

We're unable to create directories or subdirectories

There's no built in file system synchronisation this means that we have to ensure we are using proper synchronisation when calling any functions to do with the file system to avoid

Testing

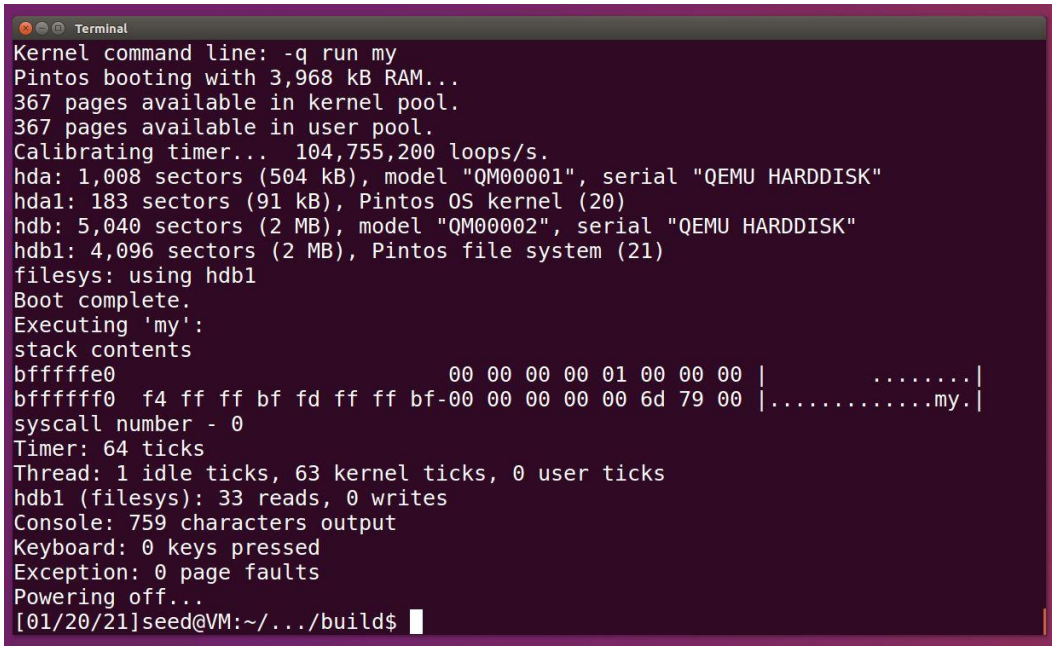
In order to ensure that our system calls are working we needed to create our own user programs that will make use of system calls to allow us to see if these system calls are working correctly.

The first system call that we tested was halt as this was the first system call we created and it would be the easiest to see if it was working as intended as all that we would have to do is to check the system powered off when halt was called.

Our test program to test halt can be seen below:

```
#include <stdio.h>

#include <syscall.h>

Int main(void){

        Halt();

}
```

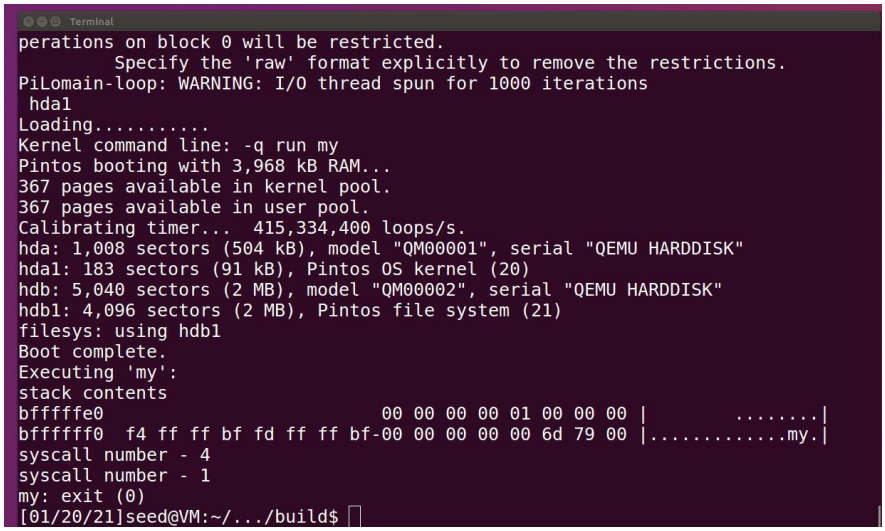
As expected, this test ran as expected and powered the pintos system down as shown below.



The next system call that we tested was create. Our test program was the same format as the one for halt however because create requires arguments to be passed into it we must make some changes as seen below:

```
#include <stdio.h>

#include <syscall.h>

Int main(void) {

        create(“”,0);

}
```

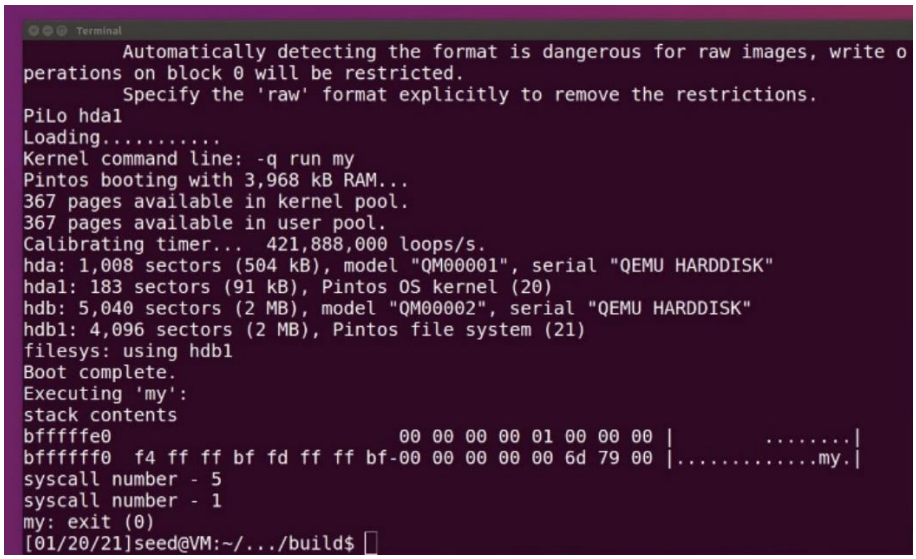
This code would create an empty file with no name and a size of 0 bytes. The output of running this program can be seen below, as you can see in the screenshot the program calls systemcall code 4 (the system call code for create) before calling the exit code to terminate the process



The next test we caried out was remove where we removed the file with the name “” the code we used to check this system call was called correctly is shown below:

```
#include <stdio.h>

#include <syscall.h>

Int main(void) {

        remove(“”);

}
```

As seen below this test program returned the correct system call code for remove and executed the program without any errors.



What were we not able to get working as intended?

There was only one system call that we could not get working which was wait. We encountered a few issues and due to this and time constraints we were not able to implement a working wait system call. Shown below is a screenshot of our current wait implementation.

```
int process_wait (tid_t child_tid)
{
    // FIXME: @bgaster --- quick hack to make sure processes execute!

/*----------unfortunately due to time considerations I wasn't able to solve the list-----
----------processing issues preventing our wait implementation from working----------*/

  for(;;) ;


  // 1. search for child in child list using child_tid
  struct thread * cur =  thread_current();
  struct list_elem *candidate = NULL;
  struct process_data *parentPData = cur ->pData;

  // every (child) process has a process_data, containing its pid
  struct process_data * childPData = NULL;


  // iterate through child processes on child list
  for (candidate = list_front(&(parentPData->child_list)); candidate != list_end(&
(parentPData->child_list)); candidate = list_next(candidate))
  {

    // get the process_data for the specified child process
    struct process_data *pData = list_entry(candidate, struct process_data, elem);

    // found child process
    if(pData->pid == child_tid)
    {
      childPData = pData;
    }
  }


  // 2. if child process is alive, wait for it to exit
  if (childPData->pid== -1)   // not found
    return -1;

  // already waiting?
  if( childPData->status == waiting)
    return -1;

  sema_init(&childPData->sema_wait, 0);  // [in exit, call sema_up]
  sema_down(&childPData->sema_wait);


  // 3. once child exits, de-allocate the descripter of the child process and
  //return its exit status
  int exit_code = childPData->exitcode;

  list_remove(candidate);
  palloc_free_page(childPData);
  palloc_free_page(parentPData);

  return exit_code;

}
```

As you can see by the comments we had issues with processing issues with the child list which caused our wait to not work.