CS39002
Operating Systems Laboratory
Spring Semester 2020-2021

ASSIGNMENT 6: Extending PintOS to run user programs with arguments (by changing program stack structure) and implementing (mostly file system-related) system calls
PART 1 + PART 2
DESIGN DOCUMENT

GROUP 8		
Atharva Naik Roshan	18CS10067	atharvanaik2018@gmal.com
Radhika Patwari	18CS10062	rsrkpatwari1234@gmail.com

ARGUMENT PASSING
========================

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

Answer:
a)Included header files in `$HOME/pintos/src/threads/thread.h`

#include <kernel/list.h>
#include <threads/synch.h>

-> Including required header files

b)Included variables to the definition of thread in
`$HOME/pintos/src/threads/thread.h`

bool success;
int error_code;
struct list child_processes;
struct thread* parent;

int fd_count;
struct semaphore sema_child;
int waiting_on_child;

-> Assigning variables to the definition of thread. `success` denotes if the thread is successfully executed. `error_code` stores the status of the thread while returning. `child_processes` is the list of child processes of the thread. `parent` is the parent process that created this thread. `fd_count` stored the number of files opened. `sema_child` is a semaphore that stores the threads which are waiting for this thread to complete. `waiting_on_child` indicates the thread is waiting for some other process to complete.

c)Included structure for a child process in `$HOME/pintos/src/threads/thread.h`

```
struct child {
  int tid;
  struct list_elem elem;
  int error_code;
  bool used;
};
```

-> Storing required values for the child like process id, element list, exit status and if it is used.

d)Using a global filesys semaphore in `$HOME/pintos/src/threads/thread.c`

```
struct lock filesys_lock;
```

-> used to synchronise the file sys and avoid race conditions if multiple concurrent processes try to access the same file.

e)initializing semaphore in thread_inti() in `$HOME/pintos/src/threads/thread.c`

```
 lock_init(&filesys_lock);
```

f)Creating a child of current thread in thread_create() in `$HOME/pintos/src/threads/thread.c`

```
 struct child* c = malloc(sizeof(*c));
 c->tid = tid;
 c->error_code = t->error_code;
 c->used = false;
 list_push_back (&running_thread()->child_processes, &c->elem);
```

-> creating a child list of the current thread and initialising its corresponding values.

g)Adding loop in thread_exit() in `$HOME/pintos/src/threads/thread.c`

```
  while(!list_empty(&thread_current()->child_processes)){
          struct proc_file *f = list_entry
(list_pop_front(&thread_current()->child_processes), struct child, elem);
          free(f);
  }
```

-> The current thread is about to exit. So we free up the memory of all the child processes created by this thread.

h) initialising new variables of the current thread in init_thread() in `$HOME/pintos/src/threads/thread.c`

```
  list_init (&t->child_processes);
  t->parent = running_thread();
  t->fd_count=2;
  t->error_code = -100;
  sema_init(&t->sema_child,0);
  t->waiting_on_child=0;
```

-> initialising child list, parent, file descriptors count, exit status of the current thread

i)Creating file locking functions in `$HOME/pintos/src/threads/thread.c`

```
void acquire_filesys_lock()
{
  lock_acquire(&filesys_lock);
}
void release_filesys_lock()
{
  lock_release(&filesys_lock);
}
```

->Acquiring and releasing the `filesys_lock` to ensure synchronisation and avoid race condition

j)Adding argument passing in process_execute() in `$HOME/pintos/src/userprog/process.c`

```
  char *save_ptr;
  char *f_name;
  f_name = malloc(strlen(file_name)+1);

  strlcpy (f_name, file_name, strlen(file_name)+1);

  f_name = strtok_r (f_name," ",&save_ptr);

  tid = thread_create (f_name, PRI_DEFAULT, start_process, fn_copy);
  free(f_name);
```

->Parsing the arguments passed during execution of the process using `strtok_r()` and creating a thread of name same as the file executable

k)ensuring semaphores in `$HOME/pintos/src/userprog/process.c`

```
  sema_down(&thread_current()->sema_child);

  if(!thread_current()->success)
```

```
        return -1;
```

-> Waiting for other sibling processes to complete if there are any. Ensuring current thread is created successfully by checking `success` value

l)Including the following in start_process() in
`$HOME/pintos/src/userprog/process.c`

```
if (!success)
{
        thread_current()->parent->success=false;
        sema_up(&thread_current()->parent->sema_child);
        thread_exit();
}
else
{
        thread_current()->parent->success=true;
        sema_up(&thread_current()->parent->sema_child);
}
```

-> If the current thread is created successfully, set `success` as true and release the semaphore of its parent. If the current thread faces error and `success` value is false, then simply release the lock and exit the process

m)Included the following in process_wait() in
`$HOME/pintos/src/userprog/process.c`

```
  struct list_elem *e;

  struct child *ch=NULL;
  struct list_elem *e1=NULL;

  for (e = list_begin (&thread_current()->child_processes); e != list_end
(&thread_current()->child_processes);
        e = list_next (e))
        {
        struct child *f = list_entry (e, struct child, elem);
        if(f->tid == child_tid)
        {
        ch = f;
        e1 = e;
        }
        }


  if(!ch || !e1)
        return -1;

  thread_current()->waiting_on_child = ch->tid;

  if(!ch->used)
        sema_down(&thread_current()->sema_child);
```

int temp = ch->error_code;
list_remove(e1);

return temp;

-> The above functionality checks if the current process needs to wait for some child process to finish. In case such a child process exists, the current process waits on the `sema_child` semaphore until it is released and the child process successfully terminates.

n)Checking exit status in process_exit() in `$HOME/pintos/src/userprog/process.c`

```
if(cur->error_code==-100)
        syscall_exit(-1);
int exit_code = cur->error_code;
```

-> checking exit status of the current thread

o)Preventing race condition in load() in `$HOME/pintos/src/userprog/process.c`

```
  acquire_filesys_lock();

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
        goto done;
  process_activate ();

  /* Open executable file. */

  char * fn_cp = malloc (strlen(file_name)+1);
  strlcpy(fn_cp, file_name, strlen(file_name)+1);

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

  file = filesys_open (fn_cp);

  free(fn_cp);

  /* Set up stack. */
  if (!setup_stack (esp, file_name))
        goto done;
  /* Assignment 6 : 2.4 ended */

  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;

 done:
  /* We arrive here whether the load is successful or not. */
  file_close (file);
```

```
release_filesys_lock();
```

->Synchronising file sys while accessing executable files to avoid race condition

p)Performing argument passing in setup_stack() in
`$HOME/pintos/src/userprog/process.c`

```
char *token, *save_ptr;
int argc = 0,i;

char * copy = malloc(strlen(file_name)+1);
strlcpy (copy, file_name, strlen(file_name)+1);


for (token = strtok_r (copy, " ", &save_ptr); token != NULL;
        token = strtok_r (NULL, " ", &save_ptr))
        argc++;


int *argv = calloc(argc,sizeof(int));

for (token = strtok_r (file_name, " ", &save_ptr),i=0; token != NULL;
        token = strtok_r (NULL, " ", &save_ptr),i++)
        {
        *esp -= strlen(token) + 1;
        memcpy(*esp,token,strlen(token) + 1);

        argv[i]=*esp;
        }

while((int)*esp%4!=0)
{
        *esp-=sizeof(char);
        char x = 0;
        memcpy(*esp,&x,sizeof(char));
}

int zero = 0;

*esp-=sizeof(int);
memcpy(*esp,&zero,sizeof(int));

for(i=argc-1;i>=0;i--)
{
        *esp-=sizeof(int);
        memcpy(*esp,&argv[i],sizeof(int));
}

int pt = *esp;
*esp-=sizeof(int);
memcpy(*esp,&pt,sizeof(int));

*esp-=sizeof(int);
memcpy(*esp,&argc,sizeof(int));
```

```
*esp-=sizeof(int);
memcpy(*esp,&zero,sizeof(int));

free(copy);
free(argv);
```

->Storing the arguments in the memory stack in user space after parsing the argument list along with a NULL value.

q)Included header files in `$HOME/pintos/src/userprog/exception.c`

```
// Set eip and eax values if in kernel mode as the test that memory is valid is unsuccessful
if(!user) { // kernel mode
 f->eip = (void *) f->eax;
 f->eax = 0xffffffff;
 return;
}
```

-> Checking if the user address is valid in order to avoid page fault condition

---- ALGORITHMS ----

>> A2: Briefly describe how you implemented argument parsing. How do
>> you arrange for the elements of argv[] to be in the right order?
>> How do you avoid overflowing the stack page?

Implementation of argument parsing:
----

1) The primary changes are inside the setup_stack function, to place the arguments correctly on the stack of the process thread.
2) Then process_execute is changed to parse the name of the executable, while creating the thread for the process. Also the load function is changed as start_process now passes the filename+the command line arguments. strtok_r is used to get the name of the executable inside the load function.
3) The detailed working of setup_stack is explained below.

Way of arranging for the elements of argv[] to be in the right order:
----

1. A loop relying on strtok_r is used to get count of arguments, and then the argv array is declared accordingly.
2. strtok_r is used to parse each argument, and the pointer pointing to each argument is stored in the argv array. A sentinel pointer is stored at the end of the array.
3. word-align zeroes are added to round down the value of the stack pointer to a multiple of 4 for faster access (as it is word-aligned).
4. Then the pointers to the arguments are stored in backwards order, (right to left), onto the stack, followed by the argv pointer (pointer to the first element of the argv array), the argc (argument count) integer and in the end a return value of 0 of void* type.

Avoiding overflow of the stack page:

The thing is we decided not to check the esp pointer until it fails. Our implementation didn't pre-count how much space do we need, just go through everything, make the change, like add another argv element, when necessary. But this leaves us two ways to deal with overflowing, one is checking esp's validity every time before use it, the other one is letting it fail, and handling it in the page fault exception, which results in exit(-1) for the running thread whenever the address is invalid. We chose the latter approach since the first approach seems have too much burden and it make sense to terminate the process if it provides too many arguments.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?

Answer:
The thing different about strtok_r is that the caller supplies the save pointer, which contains the current location of the tokenizer while parsing the string. This is because in PintOS the kernel parses the command into the executable name, and the command line arguments. Thus the placeholder is needed to put the address of arguments at a location which can be accessed later. Saving the context of the pointer makes the function thread safe.

>> A4: In Pintos, the kernel separates commands into a executable name
>> and arguments. In Unix-like systems, the shell does this
>> separation. Identify at least two advantages of the Unix approach.

Answer:
1) Parsing the command inside the kernel wastes kernel time, compared to the UNIX approach
2) Robust checking can be done if the parsing is done inside the shell. e.g. checking for existence of the file, checking whether the arguments exceed the size limit etc. These help in reducing the chances of the kernel failing.
3) The ability of the UNIX shell to parse commands also makes it possible for multiple commands to be passed together. This makes the shell act like a powerful interpreter, and not just a mere interface to the kernel.

SYSTEM CALLS
======================

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

Solution:

a)Included header files and macros in `$HOME/pintos/src/userprog/syscall.h`

```
#include "threads/thread.h"
#define ERROR -1

void syscall_exit (int status);
```

-> Including `thread.h` header file and macro `ERROR` as -1. Defining function `syscall_exit()` used by user process to exit execution

b)Included header files and macros in `$HOME/pintos/src/userprog/syscall.c`

```
#include "threads/vaddr.h"
#include "userprog/pagedir.h"
#include "userprog/process.h"
#include <user/syscall.h>
#include "list.h"

#define STD_INPUT 0
#define STD_OUTPUT 1

void syscall_exit(int status);
int syscall_write (int filedes, int buffer, int byte_size);
int syscall_exec(char *file_name);
void* check_addr(const void*);
```

-> Including the header files and defining macros along with function definitions

c) Handling various syscalls in syscall_handler() in `$HOME/pintos/src/userprog/syscall.c`

```
 int* p = f->esp;
 check_addr(p);
 int system_call = *p;

 switch (system_call)
 {
        case SYS_EXIT:
        check_addr(p+1);
        syscall_exit(*(p+1));
        break;

        case SYS_WRITE:
        check_addr(p+7);
        check_addr(p+6);
        f->eax = syscall_write(*(p+5), *(p+6), *(p+7));
        break;

        case SYS_EXEC:
        check_addr(p+1);
        f->eax = syscall_exec(*(p+1));
```

```
break;

case SYS_WAIT:
check_addr(p+1);
f->eax = process_wait(*(p+1));
break;

case SYS_CREATE:
 check_addr(p+5);
 check_addr(*(p+4));
 f->eax = syscall_create(*(p+4),*(p+5));
 break;

case SYS_REMOVE:
 check_addr(p+1);
 check_addr(*(p+1));
 f->eax = syscall_remove(*(p+1));
 break;

case SYS_OPEN:
 check_addr(p+1);
 check_addr(*(p+1));
 f->eax = syscall_open(*(p+1));
 break;

case SYS_FILESIZE:
 check_addr(p+1);
 acquire_filesys_lock();
 f->eax = file_length (get_file(&thread_current()->files, *(p+1))->file_ptr);
 release_filesys_lock();
 break;

case SYS_READ:
 check_addr(p+7);
 check_addr(*(p+6));
 f->eax = syscall_read(*(p+5), *(p+6), *(p+7));
 break;

case SYS_WRITE:
 check_addr(p+7);
 check_addr(*(p+6));
 f->eax = syscall_write(*(p+5), *(p+6), *(p+7));
 break;

 case SYS_SEEK:
 check_addr(p+5);
 acquire_filesys_lock();
 file_seek(get_file(&thread_current()->files, *(p+4))->file_ptr,*(p+5));
 release_filesys_lock();
 break;

case SYS_TELL:
 check_addr(p+1);
 acquire_filesys_lock();
```

```
        f->eax = file_tell(get_file(&thread_current()->files, *(p+1))->file_ptr);
        release_filesys_lock();
        break;

       case SYS_CLOSE:
        check_addr(p+1);
        syscall_close(*(p+1));
        break;

       default:
       printf ("Not defined system call!\n");
       break;
  }
```

-> `f->esp` gives the syscall number which is stored in system_call. Depending on the value of `system_call`,`syscall_write()`,`syscall_exit()`, `syscall_create()`,`syscall_read()`,`syscall_open()`,`syscall_remove()`,`syscall_write()`,`syscall_filesize()`,`syscall_seek()`,`syscall_tell()`,`syscall_close()` and `syscall_exec()` functions are called. `check_addr()` checks if pointer stores the address of user memory space. In case of any other system calls, an appropriate message is displayed.

d) System call exit implemented in `$HOME/pintos/src/userprog/syscall.c`

```
void syscall_exit(int status)
{
  struct list_elem *e;

  for (e = list_begin (&thread_current()->parent->child_processes); e != list_end
(&thread_current()->parent->child_processes);
       e = list_next (e))
  {
       struct child *f = list_entry (e, struct child, elem);
       if(f->tid == thread_current()->tid)
       {
       f->used = true;
       f->error_code = status;
       }
  }

  thread_current()->error_code = status;

  // parent process unblock
  if(thread_current()->parent->waiting_on_child == thread_current()->tid)
       sema_up(&thread_current()->parent->sema_child);

  thread_exit();
}
```

-> The above function is used by the user program to exit the process. The for loop checks if the current process is the child of some parent. If it has a parent, it sets its `used` value as true and `error_code` as `status`. It also checks if the parent is waiting for this child process to complete. If true, then it unlocks the parent which is currently waiting at the `sema_child` semaphore for the child process to finish.

e)System call write implemented in `$HOME/pintos/src/userprog/syscall.c`

```
/* syscall_write */
int syscall_write (int filedes, int buffer, int byte_size)
{
	if (byte_size <= 0)	// don't do anything for non-positive byte-size.
	{
	return byte_size;
	}
	if(filedes == STD_OUTPUT)	// check if the file descriptor is for STD_OUTPUT
i.e. the console
	{
	putbuf(buffer, byte_size);
	return byte_size;
	}
	else
	{
		struct opened_file* fptr = get_file(&thread_current()->files, filedes);
	 if(fptr==NULL)
		return ERROR;
	 else
	 {
		int status;
		acquire_filesys_lock();
		status = file_write (fptr->file_ptr, buffer, byte_size);
		release_filesys_lock();
		return status;
	 }
	}
	return ERROR;
}
```

-> The functions take in file descriptor `filedes`, `buffer` denoting content to write and `byte_size` denoting the number of bytes to write. If `byte_size <= 0`, then it returns without writing. The function currently is used to write only on console. In case of writing to files, we use `get_file()` to check if the file exists and then use `file_write()` to write to that file. To avoid race conditions,we use file synchronisation.

f)System call exec is implemented in `$HOME/pintos/src/userprog/syscall.c`

```
int syscall_exec(char *file_name)	// inside ths we ensure that the executable exists
and can be opened even before calling process_execute.
{
  acquire_filesys_lock();						// because
filesys_open is a critical step.
  char * fn_cp = malloc (strlen(file_name)+1);	// save a copy of the filename, which
we use for parsing the true filename/executable name and command line args.
  strlcpy(fn_cp, file_name, strlen(file_name)+1);

  char * save_ptr;							// required by
```

```
strtok_r to keep track of current location.
  fn_cp = strtok_r(fn_cp," ",&save_ptr);              // get name of the executable.

  struct file* f = filesys_open (fn_cp);               // open executable file.

  if(f == NULL)                                                                   // in case
file open fails, e.g. file doesn't exist.
  {
        release_filesys_lock();                                         // release lock.
        return ERROR;                                                          // exit with error
in case file can't be loaded.
  }
  else
  {
        file_close(f);
        release_filesys_lock();                          // release the file system.
        return process_execute(file_name);          // execute the file process (we now
know that the executable can be loaded).
  }
}
```

-> The above function takes the exec `file_name` as input and parses it to know the process by which the current process needs to be replaced. It calls `process_execute()` for the new file descriptor. To maintain synchronisation while accessing the file system, `acquire_filesys_lock()` and `release_filesys_lock()` are used.

g)Checking if file descriptor is valid in `$HOME/pintos/src/userprog/syscall.c`

```
void* check_addr(const void *vaddr)
{
  if (!is_user_vaddr(vaddr))
  {
        syscall_exit(ERROR);
        return 0;
  }
  void *ptr = pagedir_get_page(thread_current()->pagedir, vaddr);
  if (!ptr)
  {
        syscall_exit(ERROR);
  }
  return ptr;
}
```

-> The above function takes `vaddr` as input and checks if it is a valid user address(< `PHYS_BASE`). It also checks if the address corresponds to a valid page of the current process.

h)wait , create and open system calls in `$HOME/pintos/src/userprog/syscall.c`

```
int syscall_wait(tid_t child_tid)
{
   return process_wait(child_tid);
}
```

```
int syscall_create(const char *file, unsigned initial_size)
{
    acquire_filesys_lock();
    int status = filesys_create(file, initial_size);
    release_filesys_lock();
    return status;
}

int syscall_open(const char *file)
{
    acquire_filesys_lock();
    struct file* fptr = filesys_open (file);
    release_filesys_lock();

    if(fptr!=NULL)
    {
        struct opened_file *pfile = malloc(sizeof(*pfile));
        pfile->file_ptr = fptr;
        pfile->fd = thread_current()->fd_count;
        thread_current()->fd_count++;
        list_push_back (&thread_current()->files, &pfile->elem);
        return pfile->fd;
    }
    return -1;
}
```

-> `syscall_wait()` is used by parent processes to wait for the child processes to finish. `syscall_create()` is used to create a new file of the given size. `syscall_open()` is used to open the required file and then add it to the list of open files of the current thread. Returns -1 in case the file to be opened does not exist.

i)Read system call in `$HOME/pintos/src/userprog/syscall.c`

```
int syscall_read (int fd, uint8_t *buffer, unsigned size)
{
    if(fd == 0)
    {
        for(int i = 0;i < size;i++)
            buffer[i] = input_getc();
        return size;
    }
    else
    {
        struct opened_file* fptr = get_file(&thread_current()->files, fd);
        int val;
        if(fptr==NULL)
            return -1;
        else
        {
            acquire_filesys_lock();
            val = file_read (fptr->file_ptr, buffer, size);
```

```
                release_filesys_lock();
          }
          return val;
    }
    return -1;
}
```

-> reads the file pointed by the file descriptor upto the data size of `size` and then writes the data in the `buffer`. If the file does not exist, just return -1. Used file synchronisation to avoid race conditions.

j)close and remove system calls in `$HOME/pintos/src/userprog/syscall.c`

```
void syscall_close(int fd)
{
    acquire_filesys_lock();
    struct list* files = &thread_current()->files;
    close_file(files, fd);
    release_filesys_lock();
}

int syscall_remove(const char *file)
{
    acquire_filesys_lock();
    int status =  filesys_remove(file);
    release_filesys_lock();
    return status;
}
```

-> `syscall_close()` closes the file given by the file descriptor and uses file synchronisation to avoid race condition. `syscall_remove()` removes the file from the current list.

k) functions for closing files in `$HOME/pintos/src/userprog/syscall.c`

```
void close_file(struct list* files, int fd)
{
    struct list_elem *e;
    struct opened_file *f;
    for (e = list_begin (files); e != list_end (files); e = list_next (e))
    {
          f = list_entry (e, struct opened_file, elem);
          if(f->fd == fd)
          {
                file_close(f->file_ptr);
                list_remove(e);
          }
    }
         free(f);
}

void close_files(struct list* files)
{
```

```
    struct list_elem *e;
    while(!list_empty(files))
    {
          e = list_pop_front(files);
          struct opened_file *f = list_entry (e, struct opened_file, elem);
                  file_close(f->file_ptr);
                  list_remove(e);
                  free(f);
    }
}
```

-> `close_file()` closes the file with the given file descriptor and removes it from the file list. `close_files()` closes all the opened files and free up the space taken by them

l) function to get desired file in `$HOME/pintos/src/userprog/syscall.c`

```
struct opened_file* get_file(struct list* files, int fd)
{
    struct list_elem *e;
    for (e = list_begin (files); e != list_end (files); e = list_next (e))
    {
          struct opened_file *f = list_entry (e, struct opened_file, elem);
          if(f->fd == fd)
                  return f;
    }
   return NULL; // return NULL if file not found.
}
```

-> given the list of files and the file descriptor, this function returns the address of the file in the list. If the given file descriptor is not found, it returns NULL.

m) Extra definitions to the struct in `$HOME/pintos/src/threads/thread.h`

```
struct file *process_file;      // name of executable being run by the process.
struct list files;              // list of files needed by the process.
int fd_count;                   // count of file descriptors.
```

-> `files` maintains the list of the files used by the current thread and keeps the count in `fd_count`. `process_file` is the pointer to the struct storing the name of the executable being run by the current thread.

n)Struct definition in `$HOME/pintos/src/userprog/syscall.c`

```
struct opened_file { // to store details about files opened by a process.
         int fd; // file descriptor.
         struct file* file_ptr; // pointer to the file struct.
         struct list_elem elem; // list element for storing pointer to next and previous
element of the list of files.
};
```

-> struct definition added to maintain the list of files and executables being processed along with the file pointers when required.

>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?

Solution :
The file descriptors are unique only within the process. We store an **opened_file struct**. The structure has the pointer to the file struct returned by **filesys_open**, the file descriptor and list element **elem** used to store the file in the list of files belonging to the process. Since the list of files is associated with the thread struct we assign file descriptors using an **fd_count** variable which is initialized as 2 (as 0 and 1 are reserved for standard I/O) is incremented every time a new file is opened for a process. Thus file descriptor values are assigned as 2,3.. and so on for each of the files.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel

Solution :
Read:

- Check if both buffer and size are valid pointers present in the user virtual memory address space, if not, exit(-1)
- Check the value of file descriptor (fd)
- If fd == STD_INPUT, use input_getc() to read size bytes from the console
- Read from a file pointed by fd
- If no such file exits, return -1
- Acquire a file system lock on the filesystem semaphore to avoid race condition
- Read the file using the file_read() in src/filesys/file.c
- Total byte size read is returned (can be less than the size that was supposed to be read in case end of file occurs while reading)

Write:

- First we use check_addr to see if both the buffer and size variables stack pointers are valid addresses in the USER_ADDR space. Then the syscall_write function is invoked.
- Then inside the syscall write function we check if the argument **size** is non-negative
- Then we check if the file descriptor is set to STD_OUTPUT. In that case we invoke the putbuf function for writing byte_size number of characters from the buffer to the console.
- In case the file descriptor is for a file we use the **get_file** function to fetch the file struct. If null is returned we return ERROR.
- We use file_write to write byte_size number of characters from the buffer to the console and return the status returned by the function.

>> B4: Suppose a system call causes a full page (4,096 bytes) of data
>> to be copied from user space into the kernel. What is the least
>> and the greatest possible number of inspections of the page table
>> (e.g. calls to pagedir_get_page()) that might result? What about
>> for a system call that only copies 2 bytes of data? Is there room
>> for improvement in these numbers, and how much?

Solution :
For a full page of data:

- least number is 1
    - pagedir_get_page() returns the page head
    - In best case, page is contiguous in the memory
    - Entire page can be retrieved using this page pointer
- Greatest number is 4096
    - Page is not provided contiguous allocation
    - Worst case : 1 byte per frame of memory
    - Each pagedir_get_page() returns pointer to a 1 byte page

For 2 bytes of data:

- least number is 1
    - 2 bytes present in one page
    - This page pointer is returned by the pagedir_get_page()
- greatest number is 2
    - Page is not stored in contiguous location
    - Worst case : 1 byte per frame
    - Each pagedir_get_page() returns pointer to a 1 byte page

>> B5: Any access to user program memory at a user-specified address
>> can fail due to a bad pointer value. Such accesses must cause the
>> process to be terminated. System calls are fraught with such
>> accesses, e.g. a "write" system call requires reading the system
>> call number from the user stack, then each of the call's three
>> arguments, then an arbitrary amount of user memory, and any of
>> these can fail at any point. This poses a design and
>> error-handling problem: how do you best avoid obscuring the primary
>> function of code in a morass of error-handling? Furthermore, when
>> an error is detected, how do you ensure that all temporarily
>> allocated resources (locks, buffers, etc.) are freed? In a few
>> paragraphs, describe the strategy or strategies you adopted for
>> managing these issues. Give an example.

Solution :
We validate every single pointer we encounter by calling pagedir_get_page(), and checking that the returned value is not null. This ensures that this pointer references a valid address that is mapped the page directory. We also make sure every pointer that we access is a virtual address, using the is_user_vaddr function. This tests to make sure that the pointer is less than

PHYS_BASE, and therefore not a kernel pointer.

For every system call that takes an argument that is a pointer argument, such as a buffer or char*, we validate that pointer in the same manner as described above. Each of these types of arguments are passed with size, so we can check the pointer to the end of the buffer or char * argument and ensure that pointer is also valid.

This approach gets validation out of the way immediately, and therefore the function our code is not particularly obscured.

---- RATIONALE ----

>> B6: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?

Solution:
We validate the user memory by calling check_addr, to see if the stack pointer lies inside the user virtual address space. In case it doesn't we terminate the process with ERROR exit status (-1). We also check if the virtual address is actually mapped to a physical address by seeing whether or not pagedir_get_page returns NULL. If it does we terminate the process with syscall_exit(ERROR). A NULL pointer is an indicator that the particular user space address is unmapped or illegal.

>> B7: What advantages or disadvantages can you see to your design
>> for file descriptors?

Solution :
Advantages:

- The same structure opened_file can store the necessary information, and be used in essentially the same way, irrespective of how the file is opened.
- Because each thread has a list of its file descriptors, there is no limit on the number of open file descriptors (until we run out of memory).

Disadvantages:

- There exist many duplicate file descriptor structs, for stdin and stdout - each thread contains structs for these fds.
- Accessing a file descriptor is O(n), where n is the number of file descriptors for the current thread (have to iterate through the entire fd list). Could be O(1) if they were stored in an array.