Assignment - 6
Modifying PintOS to run user programs with arguments and system calls.

---- GROUP 7 ----

>> Fill in the names, roll numbers and email addresses of your group members.

Gurjot Singh Suri 17CS10058 anshusuri123@gmail.com
Kumar Abhishek 17CS10022 ohyesabhi398@gmail.com

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

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.

static bool setup_stack (void **esp,char *file_name);
We didn't declare new struct or change struct member or use global
or static variable for argument passing. We changed static bool setup_stack (void **esp) to static bool setup_stack (void **esp,char *file_name), the new argument is used for parsing the arguments using strtok_r() which are used to fill stack.

---- 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?

in load function :
stack is set up and arguments are filled up by calling setup_stack() with stack pointer and filename's copy as arguments.

in setup_stack() :
esp (stack pointer) is set to point to the PHYS_BASE (begin of stack)
check to see if install_page fails to return
a copy of file_name from the command line is split on spaces using strtok_r() and argc is set appropriately until token is NULL, then argv[] is created which is of size argc
argv is filled up with arguments by splitting the copy of filename again
arg_ptr[] array is created to store addresses of arguments of the stack, it size if argc
arguments are stored on the stack in reverse order using argv[]
for each argument, the stack pointer is decremented, then stored on the stack, the stack address is stored in arg_ptr[], total_leng is incremented by arg size
word alignment is added and then esp(stack pointer) is updated by checking whether total_leng is a multiple of 4
null character is added and esp is updated by subtracting the sizeof(char*)
argument addresses are added to the stack in reverse order like before from arg_ptr[] and then esp is updated by subtracting the size of char *
this is repeated for all the argc addresses in arg_ptr[]
size of char** is subtracted from esp(stack pointer)
stack address of the first argument is added on the stack, i.e the last address stored on stack.
fake return address is added and esp is updated

How do you arrange for the elements of argv[] to be in the right order?
The command is parsed to get the number of arguments in argc,
then we construct and argv array with datatype char *, and of size argc,
then arguments are stored in argv by again parsing the same command,
then arguments are stored on the stack in the right to left order, addresses in arg_ptr[]
by traversing the argv[] array in reverse order. Wordalign, the stack addresses from arg_ptr[] on the stack, address of first argument on stack, argc, null char and the fake return address are stored.

How do you avoid overflowing the stack page?

Checking the esp pointer is not required until it fails, like add another argv element, when necessary.
Overflowing is dealt by letting it fail and then it is handled by the page fault exception,
which then further calls sys_exit(-1) for the running thread whenever the address is invalid.
Also the process is terminated if it provides to much arguments.

---- RATIONALE ----

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

strtok_r() and strtok() has only 1 difference which is that the save_ptr (placeholder) in strtok_r() is provided by the caller. Commands are separated into command line (executable name) and arguments by kernel of Pintos. Addresses of the arguments are to be stored somewhere such that they can be reached later such that we are sure if there were more than one thread call strtok()_r each thread have their own pointer (save_ptr) which is independent from the caller and can remember their position.

>> 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.

It is cleaner to separate the executable name from the arguments before passing it off to the kernel, since they represent different things.
It should be parsed by a user program rather thant by kernel.

Validation of the input is done by the shell more safely than by the kernel.
If someone entered a very large amount of text, perhaps it would cause the kernel a problem if the kernel tried to parse it, whereas if the shell takes care of it, worst case is the shell crashes.

It can separate the commands, it can do advanced pre-processing, acting more like an interpreter not only an interface. Like passing more than 1 set of command line at a time

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.

In syscall.h:

```
/*we add new struct for storing open file descriptors of thread in fd_list*/
struct fd_elem
{
    int fd;                     /*file descriptors ID*/
    struct file *myfile;        /* the real file*/
    struct list_elem elem;      /*list elem to add fd_element in fd_list*/
}
/*Global variable Used to ensure that only one process at a time
```

```
	is executing file system code. *
	struct lock file_lock;
```

In thread.h

```
	/*we add new struct for storing children of thread in child_list*/
	struct child_elem
	{
	   bool waited;               /*to check if wait() is called before?*/
	   struct list_elem elem;    /*list elem used to add in child_list */
	   bool load_success;           /*to check if load success*/
	   int exit_status;               /*the status the child thread exit with*/
	   int cur_status;                 /*the child thread current status*/
	   int child_pid;                 /*pid of this child*/
	   struct semaphore sem;     /*parent wait child to load or exit*/
	};

	/*adding some change in struct thread*/
	struct thread{
		struct list child_list;         /*list of children this thread have*/
		struct thread * parent;         /*pointer to this thread's dad*/
		/*file file descriptors*/
		struct list fd_list;                                        /*list of file descriptors*/
		int fd_size;                                                          /*size of the file descriptors*/
		/*executable file should not edited while running*/
		struct file *exec_name;                                          /*executable held by this thread*/
	};
```

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

Within a single process file descriptors are unique. File_descriptors list fd_list (list of struct fd_elem) is being tracked by process, as well as its next available descriptor number in fd_size which is incremented every time. Our fd_elem struct is what associates the file descriptor numbers with the corresponding file

---- ALGORITHMS ----

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

Addresses while reading or writing are validated by calling check_valid_ptr to check if it is not null and that it points below PHYS_BASE using is_user_vaddr

and if it is mapped using pagedir_get_page functions, otherwise it calls sys_exit(-1);
Addresses having syscall number are validated and obtained. Then the three arguments are obtained by dereferencing the addresses after validating them. Addresses buffer and buffer+size-1 are also validated by calling check_valid_ptr. If a page fault is caused by an invalid user pointer, it is handled in userprog/exception.c and then terminates the process with sys_exit(-1).
Read is done by calling read function, write is done by calling write function, if the addresses are valid,

in read :
int read(int fd, void *buffer, unsigned size) is called
lock_acquire(&file_lock) is called to ensure at a certain time
only one process is using the file.
if fd is greater than 0 we call extract_fd() which iterates on the fd_list
of the current thread and get the file which have the same fd
if not found return NULL
file_read() which is present in file.c is called and
its return value - the number of bytes actually read or returns -1 in case of an error, i.e set eax to return val.
lock_release(&file_lock) is called before any return.

in write :
int write (int fd, const void *buffer, unsigned size) id called
lock_acquire(&file_lock) is called to ensure at a certain time
only one process is using the file.
fd is checked if it is equal to 1 then putbuff() is called to write to console
and return the size else extract_fd() is called which iterates on the fd_list
of the current thread and used to get the file which have the same fd if not found NULL is returned.
file_write() is called which is present in file.c and it returns the value of the number of bytes
actually written or -1 in case of error, i.e set eax to return val.
lock_release(&file_lock) is called before any return.

>> 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?

If the user space data spans over two pages then it would take max 2 calls to pagedir_get_page because it may be held over two pages. If it's all on one page then it would take 1 call to pagedir_get_page.

If a system call causes a full page of data to be copied, the least possible
number of inspections of the page table is 1 and the greatest possible number

is 2,this depends on if the data spans 1 page or 2 page.

For a system call that copies 2 bytes of data, the least possible number
of inspections of the page table is 1 and the greatest possible number
is 2 too (although much more likely to only require 1).

There's no way to avoid the second lookup so this can not be improved.

>> 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.

We first check the frame's stack pointer esp to be valid using check_valid_ptr then
dereference it if valid and check the system call number.

Secondly, we check the pointers to arguments using check_valid_ptr() and obtain them,
if the arguments are themselves pointers like char *, we also validate them, (like in OPEN and CREATE).

In case of WRITE and READ system calls, we additionally also validate buffer+size-1 address
by calling check_valid_ptr() to check the span.

If any pointer is invalid we call sys_exit(-1) that call thread_exit
which in turn call process_exit where we release all the resources acquired by
the thread.

If an invalid user pointer still causes a “page fault”, it is handled in userprog/exception.c’ and
terminates the process with sys_exit(-1).

We ensure all resources are freed by calling sys_exit(-1) that call thread_exit
which in turn call process_exit where we release all the resources acquired by
the thread.

Example: Dealing with bad-ptr during READ (for example), so we call

check_valid_ptr(), If the bad-ptr is NULL or greater than PHYS_BASE this
function catch it and call sys_exit(-1)
Then, we call check_valid_ptr() to check if the buffer (second argument)
spans in user page by checking buffer and buffer+size using check_valid_ptr()

---- RATIONALE ----

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

We chose to implement access to user_memory by verifying the validity
of a user-provided pointer by calling check_valid_ptr which checks if
it is not null and that it points below PHYS_BASE using is_user_vaddr and
if it is mapped using pagedir_get_page functions, otherwise it calls exit(-1).
Only then we dereference it.
We chose it because it is a simpler way to handle user memory access.
It's more difficult to handle freeing of resources if an invalid pointer causes a page fault,
because there's no way to return an error code from a memory access.
We ensure all resources are freed by calling sys_exit(-1) that call thread_exit
which in turn call process_exit where we release all the resources acquired by
the thread. It decreases some performance in case of valid pointers but
handling memory access is easier.

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

Advantages:
Regardless of whether our file descriptors are created by pipe or open, the
same structure can store the necessary information, and be used in essentially
the same way.
Dynamic list is used to store the current process open files, so we need not worry about its size.
Kernel is aware of all the open files, which gains more flexibility to manipulate the opened files.
Thread-struct’s space is minimized.

Disadvantages:
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, but then array size would not be fixed.
The inherits of open files opened by a parent require extra effort to be implement.
Consumes kernel space, user program may open lots of files to crash the kernel.