OS Lab Assignment 6.2

---- GROUP ----

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

Ashutosh Varshney 18CS30009 ashutosh.varshney.varshney@gmail.com
Sahil Jindal 18CS10048 sahil132jindal@gmail.com

---- PRELIMINARIES ----

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

timer.c implements the wakeupticks as in Assignment 3. We have submitted the same file as in Assignment 3 for that.

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.

No specific struct is defined for argument passing. We have made changes in the function setup_stack in process.c to account for argument passing which declares local variables argv, argc, Z and copies them at specific locations in esp.

---- 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 setup_stack function inside process.c :

- We first make esp (stack pointer) to point to the PHAS_BASE (begin of stack)
- We split the file_name(command line) on spaces using strtok_r() and count the number of arguments in argc
- We allocate the required size to argv according to argc.
- Then we iterate on the file and copy each word to the argv. We copy the same to the esp after allocating some space.
- Finally we have in our esp: the argv (that contains the words) and the number of arguments(Z)

---- RATIONALE ----

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

The ‘r’ in strtok_r stands for re-entrant. A function is called re-entrant if there is a possibility that it can be stopped at some point and then resume from the point where it stopped earlier. In the implementation of strtok_r() provided by pintOS there is an argument called char **save_ptr which saves the context by keeping track of the tokenizer’s position. So the function can start tokenizing from where it left. This is preferable and safe compared to strtok() in case where another thread gains control as it saves the context and thereby making it possible to be called in nested loops.

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

1. The implementation of the kernel becomes simpler in case of UNIX as it doesn’t have to parse the command string and validate certain things which makes it simpler otherwise the kernel would have to waste some time doing this. Moreover there is no such reason as to why a system program like shell can’t do the same task of separating commands into executable names and arguments.

2. The UNIX approach also takes care of security for otherwise a user can knowingly or unknowingly pass some bad arguments to the kernel which might create some problems. Also, it becomes much more flexible for users to use different shell interpreters for different purposes as the parsing would be done by those shell interpreters and not by the kernel itself.

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 thread.h:

```
struct child {                          //Used as a structure for the child process
    int tid;
    struct list_elem elem;
    int error_exit;
    bool flag;
  };
```

```
  int64_t wake_up_time;   // stores time at which the process should leave block state
   bool success;      //to tell if the thread has successfully executed
   int error_exit;          //exit code
   struct list childs;   //child list
   struct thread* parent;         //pointer to parent
   struct file *self;        // pointer to self file
   struct list files;    //list of files
   int fd_count;   // counter of file descriptor
   struct semaphore child_lock;         //semaphore for child process
   int waitingon;          //child id on which parent is waiting
```

In thread.c:

```
struct lock filesys_lock; // global variable for file system lock to ensure only one process
accesses it
#define INIT_FD 2                       //used in initialisation
#define INIT_ERROR -100                 //used in initialisation
#define INIT_WAIT 0                     //used in initialisation

void acquire_filesys_lock(void)                 //to acquire filesys lock
{
  lock_acquire(&filesys_lock);
}

void release_filesys_lock(void)                 //to release filesys lock
{
  lock_release(&filesys_lock);
}
Initialised the new variables in init_thread
```

In syscall.c

```
struct proc_file {                      // structure for a file process
        struct file* ptr;
        int fd;
        struct list_elem elem;
};
void exit_process(int status)           // to exit a process
int exec_process(char *file_name)               //to execute a process
void* valid_addr(const void *vaddr)             // to check that the given memory address is valid
void close_file(struct list* files, int fd) // to close a file using it's fd and remove from files list
void close_all_files(struct list* files)    // close and remove all files in files list
```

In process.c

No new struct or global variables created. Changes made to the following functions: process_execute, start_process, process_wait, process_exit, setup_stack, load

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

File descriptors are unique within a single process and not within the entire OS. Each process keeps track of the list of files that have been opened which are mapped to a unique descriptor. We have implemented this by making use of a list of files and file descriptor count declared inside struct thread.

---- ALGORITHMS ----

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

For write, we first check whether the file descriptor has a value 1 or not. If yes then we call putbuf() else we search for the file with the same file descriptor value. If we are able to find it then acquire_filesys_lock() is called to ensure that only 1 process accesses this file at a time. After this we call the pre-defined file_write() declared in file.c with the given arguments. Subsequently, we release the lock using release_filesys_lock().

For read, we first check whether the file descriptor has a value 0 or not. If yes, we read the input using input_getc() and set the return value. Otherwise, we iterate on the file list of the current thread to search for a file having the same fd value. If not found, we return -1. Otherwise, acquire the lock of the file and then call file_read(), which is pre-declared in file.c, to set the return value. Before returning, release the file lock.

>> 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?

In the first case, the least possible number of inspections of the page table is 1(when a segment size for ELF in memory is greater than or equal to the segment size in files) and the greatest possible number is 4096 when each ELF segment is only one byte in size ie 1 inspection for each byte.

In the latter case, the least possible number of inspections of the page table is 1 and the greatest possible number is 2.
An improvement can be to allow the file segments to share pages instead of allocating one page per segment when the page size is much more than the segment size.

>> 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.

For each system call all the arguments and other pointers are validated with the help of the esp and their addresses are checked in syscall_handler() before continuing further. We have implemented valid_addr() to make this check which first checks whether the address is less than the physical base of the stack or not using is_user_vaddr() to ensure that it is not a kernel pointer. We also check whether the pointer is NULL or not. If the check fails on the esp itself we call thread_exit() otherwise we proceed according to the syscall number to check the arguments. If the check fails on any of the arguments then we call exit_process(-1). exit_process() sets the status of the process to -1 after doing a linear search, sets flag to true as well as if the parent of the process was waiting for it then the function sema ups the semaphore child_lock. Therefore our strategy of validating everything first, assigning resources after all validation checks and releasing semaphores and other resources on validation fails handles everything correctly.

Example:

Suppose a process P has children C1,C2 and C3 and P is waiting on C3. Suppose now C3 is being executed which asks for a syscall. But due to a bad pointer reference, exit_process() gets called with -1. exit_process() does a linear search over the list of children of P i.e. [C1,C2,C3] and finds C3 using the tid. Then the status of C3 is set to -1 to kill it and the semaphore on which P was waiting for C3 is increased.

---- RATIONALE ----

>> B6: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

We referred to the pintOS documentation Section 3.1.5 which states 2 methods of accessing user memory. We went with the first one as it was easier to implement. Link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html

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

Advantages: The implementation is simple since we are just maintaining a simple list of file descriptors thus insertion is done in O(1) and there is no limit as such to the number of files a process can have. The implementation also makes use of the number of files open to provide a unique mapping for file descriptors within a process.

Disadvantages: The search time for a particular file descriptor is O(n) as the entire list needs to be traversed in worst case. A possible solution is to use array indexing which will allow search in O(1) but this would impose a restriction on the number of files that a process can have opened which is not desirable.