+-----------------------+
| CS 212 |
| PROJECT 1: Threads |
| DESIGN DOCUMENT |
+-----------------------+

---- GROUP ----

Karim Tarek Ibrahim
Hussein Khaled
Mohamed Anwar
Nagui Mostafa
Abdullah Taman

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

---- DATA STRUCTURES ----

>> A1:

```
struct list_elem ch_elem;       // To put it in children
struct semaphore parent_wait_till_child_exit;
struct semaphore child_parent_relation; // Parent forces child to work here
struct file *exe;
```

- The list children :

It's used to detect the children of the parent whether it's an initial or a parent in the shell waiting for some child.

- The semaphores :

The first one is used in the exec_process function to force the parent to wait until the child is created, then it's reused in the start_process function to free the parent and if the child creates success it blocks the child till the parent wakes up it in the wait function.

---- ALGORITHMS ----

>> A2:

First of all we get the line written in the command line and passed in exec_process from the parameter of the function then we create a thread to execute the start_process function, there we call the function load. The load is the place where we get the name of the program and open it, if successful we continue to read the rest of the arguments passed in the cmd. We do so in the push_stack_arguments, we use the strtok_r function which cuts the input line by line.

---- RATIONALE ----

>> A3:

Strtok cannot be used in multiple thread environments because it saves the state in a global variable, but strtok_r and the _r versions of functions are reentrant: you can call them from multiple threads simultaneously, or in nested loops, et cetera.. Therefore it is implemented this way so that commands may be separated at command line and arguments may be put into store for later on in use.

>> A4:

1. It shortens the time a user is inside a kernel

2. Checks to ensure the executable is there before pushing to the kernel to avoid kernel Panic. If the user program fails, we shall not worry. But if the kernel panics it’s much worse.

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

---- DATA STRUCTURES ----

>> B1:

In thread.h:

```
struct files_opened
{
    struct file *f;
    struct list_elem elem;
    int file_descriptor;
};
```

This struct is used to detect files opened by the thread, so it points to the opened files by the thread

Added new data structures to the thread struct :

```
tid_t certain_tid_parent_wait_for;
bool child_success;      // depend on the load function
int child_result_status; // My child failed or what
int exit_status;         // I failed or what
struct thread * parent;
struct list children;
struct list files_opened_by_me; // Important for removing
struct list_elem ch_elem;       // To put it in children
struct semaphore parent_wait_till_child_exit;
struct semaphore child_parent_relation; // Parent forces child to work here
struct file *exe;
```

>> B2:

- Using files_opened struct which encapsulates both the file pointer and the file descriptor and the list_elem which is important for the thread to be able to carry a list of file_opened struct so at any moment it can access the list and get the file pointer itself through the file descriptor or the opposite.
- File descriptors are unique within a single process, not the entire OS.

---- ALGORITHMS ----

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

The kernel calls the sys_call handler where we validate the stack value passed, then if it is valid, then we extract the value used in the switch statement. With the values found in syscall-nr.h

```
/* System call numbers. */
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. */

   /* Project 3 and optionally project 4. */
   SYS_MMAP,                   /* Map a file into memory. */
   SYS_MUNMAP,                 /* Remove a memory mapping. */

   /* Project 4 only. */
   SYS_CHDIR,                  /* Change the current directory. */
   SYS_MKDIR,                  /* Create a directory. */
   SYS_READDIR,                /* Reads a directory entry. */
   SYS_ISDIR,                  /* Tests if a fd represents a directory.
*/
```

```
  SYS_INUMBER                 /* Returns the inode number for a fd. */
};
```

We choose to read or write according to this value so suppose the value is SYS_WRITE.

```
case SYS_READ:
{
  system_read_wrapper(f);
  break;
}
case SYS_WRITE:
{
  system_write_wrapper(f);
  break;
}
```

Then we go to the system_write_wrapper where we validate the ptr to point to user space and not kernel space and also validate if the ptr mapped to a virtual address and if the value of fd=0 then we want to read from stdin which is not acceptable. If the conditions are not passed then we call sys_exit(-1). Otherwise, go to the sys_write where we check if fd=1 then we write to stdout so we call putbuf method and make the proper synchronization. Otherwise, we write to a file using the file_write method and make the proper synchronization.

```
void system_write_wrapper(struct intr_frame *f)
{
 int fd = *((int *)f->esp + 1);
 char *buffer = (char *)(*((int *)f->esp + 2));
 // fd must not be 0 because zero is stdin, will be used in read
 if (fd == 0 || !validate_address_in_virtual_memory(buffer))
 { // fail, if fd is 0 (stdin), or its virtual memory
   sys_exit(-1);
 }
 // Pull the fourth parameter which is size of the output
 unsigned size = (unsigned)(*((int *)f->esp + 3));
 f->eax = sys_write(fd, buffer, size);
}

int sys_write(int fd, const void *buffer, unsigned size)
{
 if (fd == 1)
 { // fd is 1, writes to the stdout
   lock_acquire(&global_lock);
   putbuf(buffer, size);
   lock_release(&global_lock);
   return size;
```

```
 }

 struct files_opened *file = sys_file_helper(fd);
 if (file == NULL)
 { // fail
   return -1;
 }
 else
 {
   int ans = 0;
   lock_acquire(&global_lock);
   ans = file_write(file->f, buffer, size);
   lock_release(&global_lock);
   return ans;
 }
}
```

In case of read we will make the same steps of write but if fd=1 then we will not validate this. Otherwise, we will call sys_read function and if fd=0 then we take input from stdin then we call input_getc function. Otherwise, we call file_read function and both with proper synchronization.

```
void system_read_wrapper(struct intr_frame *f)
{

 int fd = *((int *)f->esp + 1);
 char *buffer = (char *)(*((int *)f->esp + 2));
 // fd must not be 1 because zero is stdin, will be used in write
 if (fd == 1 || !validate_address_in_virtual_memory(buffer))
 { // fail, if fd is 1 (stdin)
   sys_exit(-1);
 }

 unsigned size = (unsigned)(*((int *)f->esp + 3));
 f->eax = sys_read(fd, buffer, size);
}

int sys_read(int fd, void *buffer, unsigned size)
{
 int size_of_file = size;
 if (fd == 0)
 {

   while (size--)
   {
```

```
    lock_acquire(&global_lock);
    char ch = input_getc();
    lock_release(&global_lock);
    buffer += ch;
  }
  return size_of_file;
}
else if (fd == -1)
{
  // negative area
}
else
{
  struct files_opened *file = sys_file_helper(fd);
  if (file == NULL)
  { // fail
    return -1;
  }
  else
  {
    lock_acquire(&global_lock);
    size_of_file = file_read(file->f, buffer, size);
    lock_release(&global_lock);
    return size_of_file;
  }
}
}
```

>> B4:

The least number of inspections on the page table would only be 1, as it would only need 1 inspection in order to have the page table organized. The most that would need is 4096 as it would then have to inspect every byte of data. For 2, the greatest number would be 2, since there are only 2 bytes. As far as it goes, it executes in n time so there is no reason to improve this.

>> B5: Briefly describe your implementation of the "wait" system call
>> and how it interacts with process termination.

We validate the address in the system wrapper then we make the parent point to this child that we waited on and if the child is not null then we remove the child from the parent's child

list and then we wake up the child and block the parent and we return the current child status. Otherwise, we return -1.

```
int process_wait(tid_t child_tid) // Shell comes here to wait  for
child
{
 thread_current()->certain_tid_parent_wait_for = child_tid;
 struct thread *child = find_child(child_tid);
 if (child != NULL)
 { // Execute it
   list_remove(&child->ch_elem);
   sema_up(&child->child_parent_relation);                    // Go
child and execute
   sema_down(&thread_current()->parent_wait_till_child_exit); // Wait
for your child
   return thread_current()->child_result_status;
 }

 return -1;
}
```

At the Process Termination when calling the process_exit we check if there is a parent thread that is waiting on me then we set the child status of the parent to the child exit status and we make the parent is not waiting on this child and then make the parent wake up.

```
 if (cur->parent != NULL) // I'm a child
 {
   struct thread *parent = cur->parent;
   if (parent->certain_tid_parent_wait_for == cur->tid) // if parent is
waiting on me
   {
     parent->child_result_status = cur->exit_status;
     // Now parent not wating for anybody
     // Let's reset everything
     parent->certain_tid_parent_wait_for = -1;
     parent->child_success = false;
     // Free the parent
     sema_up(&parent->parent_wait_till_child_exit);
   }
 }
```

>> B6: 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 we implemented there is a wrapper function that calls the sys_call function and gives it the required parameters , the wrapper is responsible for pulling the sys_call function arguments from the stack and validating the pointers then calling the function after that so it decouples the validation code from the system call code.
- If any error occurred after validating the pointer the sys_exit() function is called which is responsible for shutting down the thread/process and deallocating the semaphores , lists and anydata allocated for the thread.

sys_exit and system_exit_wrapper

```
static void
syscall_handler(struct intr_frame *f UNUSED)
{
  if (!valid_esp(f))
  {
    sys_exit(-1);
  }

  // We will read only one integer telling me what operation is to be executed
  switch (*(int *)f->esp)
  {

  case SYS_HALT:
  {
    system_halt_wrapper(f);
    break;
  }
  case SYS_EXIT:
  {
    system_exit_wrapper(f);
    break;
  }
```

```
void system_exit_wrapper(struct intr_frame *f)
{

  int *status_pointer = (int *) ((int *)f->esp + 1);
  if (!validate_address_in_virtual_memory(status_pointer))
  {
    f->eax = -1;
    sys_exit(-1);
  }

  f->eax = *status_pointer;
  sys_exit(*status_pointer);
}
```

```
void sys_exit(int status)
{
  struct thread *t = thread_current();
  char* name= t->name;
  char * save_ptr;
  char * exe = strtok_r (name, " ", &save_ptr);


  t->exit_status = status;

  printf("%s: exit(%d)\n",exe,status);

  thread_exit();
}
```

---- SYNCHRONIZATION ----

>> B7:

Firstly, in the process_execute function we force the parent to wait for the child on the semaphore child_parent_relation, and we call the function thread create that goes to the start process function. Then we try to load the parent in the load function and call it the start_process. Load returns a Boolean indicating the result of the operation. In this part of the code we report the parent the result of the operation whether it's a success or fail, then we sem_up the parent and sema_down the child if successful. We set the child_success to true or false in the parent thread indicating the result of the operation.

>> B8:

- If P calls wait(C) before c exits, p is blocked on the semaphore waiting for the C. After C exits this is not possible because when c exits P no longer sees it after C exits because c deletes itself from the parent so this is not possible. When any process exits we free its resources with a while loop in the exit function.
- If P terminates without waiting before C, this is not a problem because a parent frees the children's relation to this parent and they become normal processes. If after C exits this is not a problem also because a child free itself from the parent on exiting.
- No special cases.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?

We did it that way because:

- It was suggested we use palloc and malloc to get a user stack reference to the kernel stack.

- It helped create efficient communication between the user stack and the kernel stack

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?

Advantage:
- The struct makes it easy to get the file descriptor if you have the file pointer or the struct itself.

DisAdvantage:
- It takes O(n) time to access the file pointer from the file descriptor , another implementation would use a hashmap for better performance.

>> B11: The default tid_t to pid_t mapping is the identity mapping.
>> If you changed it, what advantages are there to your approach?

- Used tid_t for processes and threads (kept the one to one mapping between tid_t and pid_t).

- If it was changed, we would have to change how we implemented the current thread and child thread functions since both rely on the tid_t in order to tell where exactly they are in the process.