```
+--------------------------------------+
|             CS 3204                  |
| PROJECT 2: USER PROGRAMS |
|      DESIGN DOCUMENT        |
+--------------------------+
```

---- GROUP ----
Ahmed Khalil ElZainy (20010087)
Ahmed Samir Said Ahmed (20010107)
Mohamed Wael Fathy (20011752)
Youssef Hossam AboElwafa (20012263)
Youssef Mohamed ElMedany (20012293)

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.
None.

---- 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?

1. **The implementation of argument passing :-**
   - In the Process_execute() function the file_name is provided as a parameter, including the command and its arguments. So use the total file_name as the new thread's name,pass it to start_process(),load(). load() function loads an ELF executable from file_name into the current thread. It also sets the entry point (eip) and initial stack pointer (esp) for the user process and separated the first token and the rest, which are command and arguments then pass the command, stack_ptr (esp) and save_ptr to push_into_stack() function.
   - `void push_into_stack(char *file_name, void **esp, char **save_ptr)`
     in this function push all arguments onto the stack by iterating over the filename using the save_ptr and taking a token which represents the next argument. To ensure proper alignment, the function calculates the (align) value based on the size of all arguments. It then pushes the necessary number of zeros to the stack to align it to a word size (4 bytes). Next pushed the addresses of the arguments onto the stack in reverse order. Finally, it pushed the address of the number of the arguments , pushing Null onto stack and update and return esp.
2. **How do you arrange for the elements of argv[] to be in the right order?**
   We scan through the argument string backwards, so that the first token we get is the last argument, the last token we get is the first argument. We can just keep decreasing the esp pointer to set up the argv[] elements.
3. **How do you avoid overflowing the stack page?**
   We decided not to check the esp pointer for overflowing until it fails. As we go through everything and make changes whatever we want like add another argument and don't count how much space we reserved. However, it leaves us with two choices ,first is to check the validity of esp every time before using it ,but it comes with more overhead and may terminate a process if it invalid. So we take the second choice which includes letting it fail and handle the page fault exception which is exit(-1) the running thread whenever the address is invalid.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?
strtok_r() function is similar to strtok() but it takes an extra argument which is a pointer to a char pointer which is used to store the position of the next token. The save_ptr in strtok_r() is provided by the caller. In pintos, the kernel separates commands executable name and arguments. So we need to put the address of the arguments somewhere we can reach later.

>> 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.
In UNIX:-

1. Users can combine different executables with various arguments to create complex commands or execute multiple commands in a single line.
2. Shorting the time passed inside the kernel
3. Checking whether the arguments are over the limit, and checking whether the executable is there before passing it to the kernel to avoid kernel failure.
4. Users can construct commands by specifying the executable name followed by its argument which eases command line completion.

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.c:

we added a global variable lock_for write that is used to synchronize writing requests among different processes and threads.

```
struct lock lock_for_fileaccess;
```

In struct thread:

`struct thread *parent;` a parent element representing the parent of the thread.

`struct list children;` a list of children of the current thread.

`bool create_child_done;` a boolean to check the creation of children.

`int child_exit_status;` a boolean to save the exit status of its child.

`tid_t waiting_for_child;` the id of the child i am waiting for.

`struct semaphore wait_for_child;` a lock is used to enable the child to execute its process.

`struct semaphore synchronized_wait_for_child;` a lock to synchronize among parent and child to enable the child load its data

`struct list user_files;` a list of opened executable files.

`int exit_status;` the exit status of the running thread.

`struct file *executable;` the currently executed executable file.

`struct list_elem child_elem;` list element to point to the child on the children list.

struct user_file:

`struct list_elem elem;` element for user-file to point for in the user-file list of the thread.

`int fd;` an integer descriptor for the file.

`struct file *file;` a pointer to the current executable file.

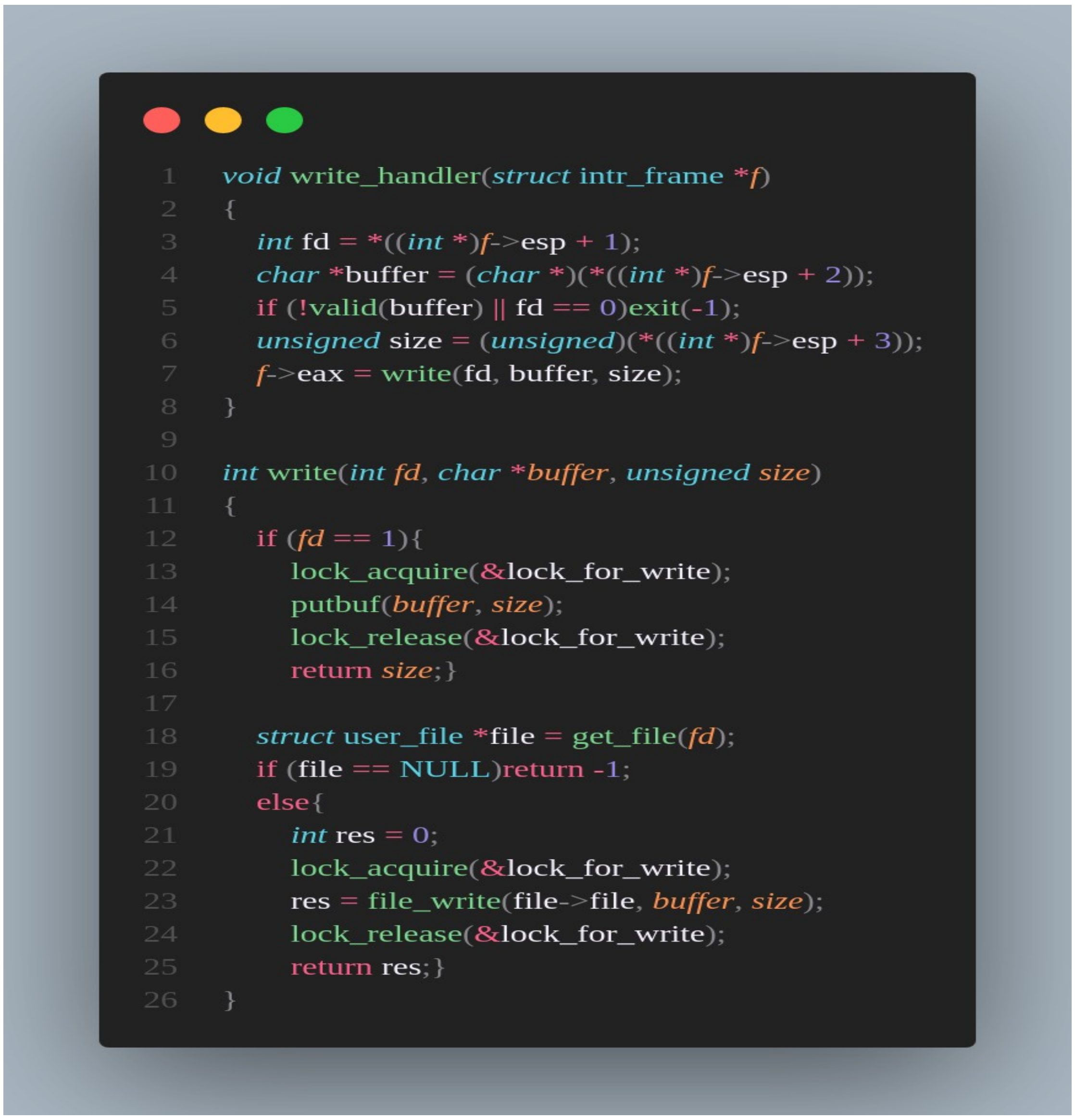
>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?

in our implementation we created a list of files for each process inside the thread struct to maintain the open files by the current thread which means that the file descriptor isn’t unique for all opened files in the system so that a single file may be opened by two or more different processes with different file descriptors but they are unique inside each single process.

---- ALGORITHMS ----

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

- First, get the file pointer (fd) ,buffer pointer and its size pointer from the stack. Then check if the buffer to write in is valid and if the fd is stdin (not equal to 1), if not exit(-1).Next call the read() function and store the return value in eax which is the return value of stack handler.
- read() function acquires lock for reading(lock_for_fileaccess) and if it is a stdin then retrieves keys from standard input using input_getc() function ,after that, releases the lock and returns 0. Otherwise, find the open file according to the fd number from the user_file list. Then check if the user_file is Null and return -1 otherwise acquiring lock for reading (lock_for_fileaccess) and use file_read() function in filesys to read the file, get status. Release the lock and return the status.

- 

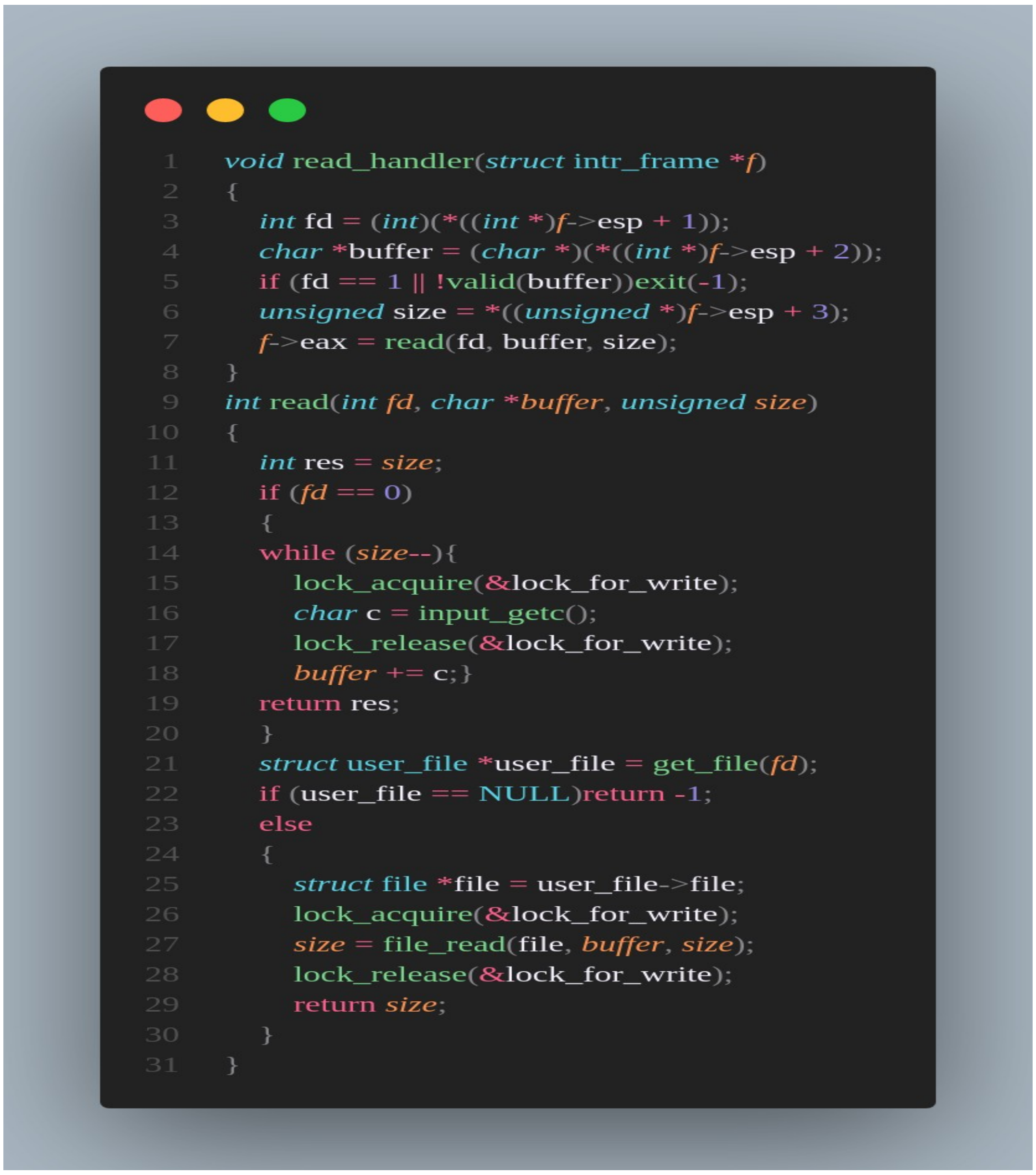
```c
void write_handler(struct intr_frame *f)
{
   int fd = *((int *)f->esp + 1);
   char *buffer = (char *)(*((int *)f->esp + 2));
   if (!valid(buffer) || fd == 0)exit(-1);
   unsigned size = (unsigned)(*((int *)f->esp + 3));
   f->eax = write(fd, buffer, size);
}

int write(int fd, char *buffer, unsigned size)
{
   if (fd == 1){
      lock_acquire(&lock_for_write);
      putbuf(buffer, size);
      lock_release(&lock_for_write);
      return size;}

   struct user_file *file = get_file(fd);
   if (file == NULL)return -1;
   else{
      int res = 0;
      lock_acquire(&lock_for_write);
      res = file_write(file->file, buffer, size);
      lock_release(&lock_for_write);
      return res;}
}
```

-Writing:

- First, get the file pointer (fd) ,buffer pointer and its size pointer from the stack. Then check if the buffer to write in is valid and if the fd is stdout (not equal to 0), otherwise exit(-1).Next call the write() function and store the return value in eax which is the return value of stack handler.
- write() function acquires lock for writing (lock_for_fileaccess) and if it is a stdout, prints the content of the buffer to the console using putbuf() function ,after that, releases the lock and returns the size printed . Otherwise, find the open file according to the fd number from the user_file list. Then check if the user_file is Null and return -1 otherwise acquiring lock for writing (lock_for_fileaccess) and use file_write() function in filesys to read the file,get status. Release the lock and return the status.

- 

```
void read_handler(struct intr_frame *f)
{
   int fd = (int)(*((int *)f->esp + 1));
   char *buffer = (char *)(*((int *)f->esp + 2));
   if (fd == 1 || !valid(buffer))exit(-1);
   unsigned size = *((unsigned *)f->esp + 3);
   f->eax = read(fd, buffer, size);
}
int read(int fd, char *buffer, unsigned size)
{
   int res = size;
   if (fd == 0)
   {
   while (size--){
      lock_acquire(&lock_for_write);
      char c = input_getc();
      lock_release(&lock_for_write);
      buffer += c;}
   return res;
   }
   struct user_file *user_file = get_file(fd);
   if (user_file == NULL)return -1;
   else
   {
      struct file *file = user_file->file;
      lock_acquire(&lock_for_write);
      size = file_read(file, buffer, size);
      lock_release(&lock_for_write);
      return size;
   }
}
```

>> 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?

- For the full page (4,096 bytes) of data:
The least number is 1.if the first inspection gets a page head and the data are stored exactly in a single page therefore only one page needs to be verified. As the pointer returning from pagdir_get_page is enough to get the whole stored data with no need to inspect more.

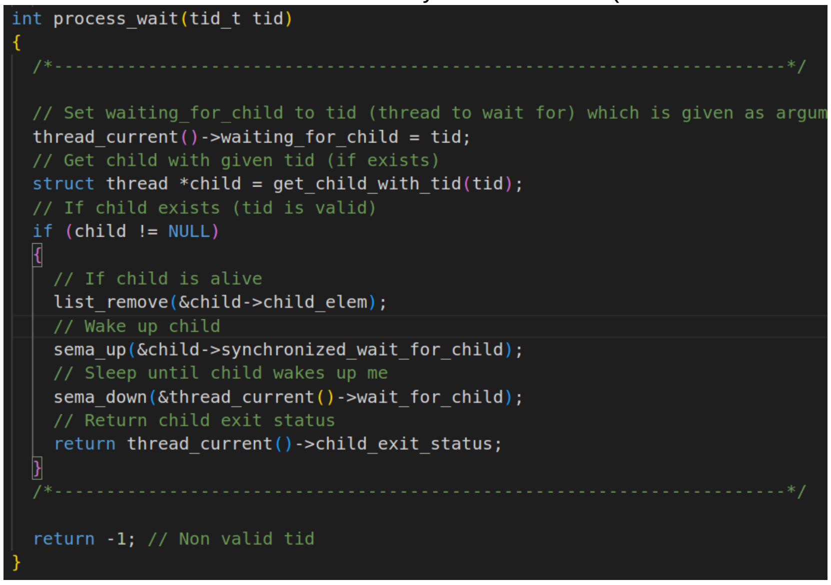
The largest number is 4096, if it's not contiguous, in that case we have to check every address to ensure a valid access.As the inspected data are sparse in a byte-size level across 4096 pages, thus pagedir_get_page is called once per byte.

-For 2 bytes of data, the least number is 1: the two bytes are existent in the same page. The largest number is 2: the two bytes are existent in separate pages.

-It is challenging to make an improvement in these numbers as it should be inspected for non-contiguous pages of data. However, there is a way to improve it by making full use of pages when loading data as the inspections occur when data lies on different pages.

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

Firstly, It calls the **Waiting_handler** function which is used to check for the validity of the process that we want to wait on by checking its name doesn't equal null and that it has a valid virtual address.
The **Waiting_handler** function sets the status of the frame(which holds our desired process) with the return value of the **wait** function.
The wait function takes tid and calls **process_wait** function with it.
Lastly, The **process_wait** function: It checks for the existence of child of the process which is supposed to wait for it,
if there is a child to wait for then it removes the child from the list , wakes up the child and block the parent on the **wait_for_child** lock.
If the child was terminated by the kernel (i.e. killed due to an exception), returns -1



```c
int process_wait(tid_t tid)
{
  /*---------------------------------------------------------------------*/

  // Set waiting_for_child to tid (thread to wait for) which is given as argum
  thread_current()->waiting_for_child = tid;
  // Get child with given tid (if exists)
  struct thread *child = get_child_with_tid(tid);
  // If child exists (tid is valid)
  if (child != NULL)
  {
    // If child is alive
    list_remove(&child->child_elem);
    // Wake up child
    sema_up(&child->synchronized_wait_for_child);
    // Sleep until child wakes up me
    sema_down(&thread_current()->wait_for_child);
    // Return child exit status
    return thread_current()->child_exit_status;
  }
  /*---------------------------------------------------------------------*/

  return -1; // Non valid tid
}
```

>> 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 this part of the phase Our design depends on having a multiple of system call handlers that handles the type of system call inside the "sys_call handler" function. this syscall handler at the beginning validates the pointer to the stack of the interrupt frame that currently holding the data of the executing thread by passing the frame as an argument to a function called valid_esp() which does the previous logic of validation. after validating the process stack, we pull the top of the stack which should represents one of the 12 sys_call numbers which should be pushed previously when interrupt handler is called. after getting the syscall number, we switch on it to determine its type and perform the corresponding system call. after defining the type of system call, we call the corresponding handler as mentioned above. The role of each handler is first validate the pointer to the file you need to access either by checking if it exists or not and in addition if it has access to it or not. by validating these previous inside the handler before calling the kernel function itself, it prevent from having any non released-resources if it fails at any of the previous validation as it exit(-1) if it fails to validate and inside the exit_function a call to process_exit is invoked to release all resources on exiting the process. process exit is called in the two cases of success and failure. in addition, if the error still happens we handle it again in page fault exception by calling exit(-1) to make sure that all resources are freed on issuing an error.

---- SYNCHRONIZATION ----

>> B7: The "exec" system call returns -1 if loading the new executable
>> fails, so it cannot return before the new executable has completed
>> loading. How does your code ensure this?
>> How is the load success/failure status passed back to the thread that calls "exec"?

Our design is to have the child_load_status **recorded in the parent's thread.** Child is responsible to set child_load_status.

thread.h in the **struct thread**:

```
bool child_creation_success;
int child_status;
```

The reason we choose this design is that the child thread **can exit anytime** due
to some reasons.
So, if we save it in the child process, there is **no way to retrieve the status** if it exits before the parent checking on it.
A thread can only wait a thread to load at a time, so use only one variable inside the parent thread is enough.

>> B8: Consider parent process P with child process C. How do you
>> ensure proper synchronization and avoid race conditions when P
>> calls wait(C) before C exits? After C exits? How do you ensure
>> that all resources are freed in each case? How about when P
>> terminates without waiting, before C exits? After C exits? Are
>> there any special cases?

We use a child_status flag to represents each child process's status, and a
list of child_status inside **parent's struct** to represent all the children that
the process has. And use a **monitor** to prevent race condition.

thread.h in the **struct thread**:

```
struct thread *parent;
struct list children;
bool child_creation_success;
int child_status;
tid_t waiting_for;
struct semaphore wait_for_child;
struct semaphore parent_child_sync;
```

Child is responsible to set **it’s status in parent’s thread struct**.
When parent exits, the list inside it will be free and will remove all children

```
struct list *children = &thread_current()->children;
struct list_elem *iter = list_begin(children);
while (iter != list_end(children))
{
  struct thread *child = list_entry(iter, struct thread, child_elem);
  iter = list_next(iter);
  child->parent = NULL;
  sema_up(&child->parent_child_sync);
  list_remove(&child->child_elem);
}
```

So, in the cases above:

P calls wait(C) after C exits:
P will acquire the monitor and found out C already exits and check it’s exit status directly.

P calls wait(C) before C exits:
P will acquire the monitor and wait until it exits by checking the child thread’s existence through a function (child_with_id) we wrote, which checks all-thread-list. Then parent retrieves the child’s exit status.

P terminates without waiting before C exits:
The list inside P will be free, the lock and resources will be released, since no one will wait a signal except parent, condition don’t need to be signaled. When C tries
to set it’s status and find out parent has exited, it will ignore it and continue to execute.

P terminates after C exits:
The same thing happen to P, which is free all the resources P has.

---- RATIONALE ----

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

We chose to check only that the pointer is below PHYS_BASE and deference the pointer using the get_user() method. In the Pintos manual, it is said that this method is faster than the method that verifies the validity of the user-provided pointer, so we chose to implement this way. This method did not seem too complicated, and since it is the faster option of the two, it seemed to be a better choice.

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

Advantages:
1) Thread-struct’s space is minimized
2) Kernel is aware of all the open files, which gains more flexibility to manipulate the opened files.

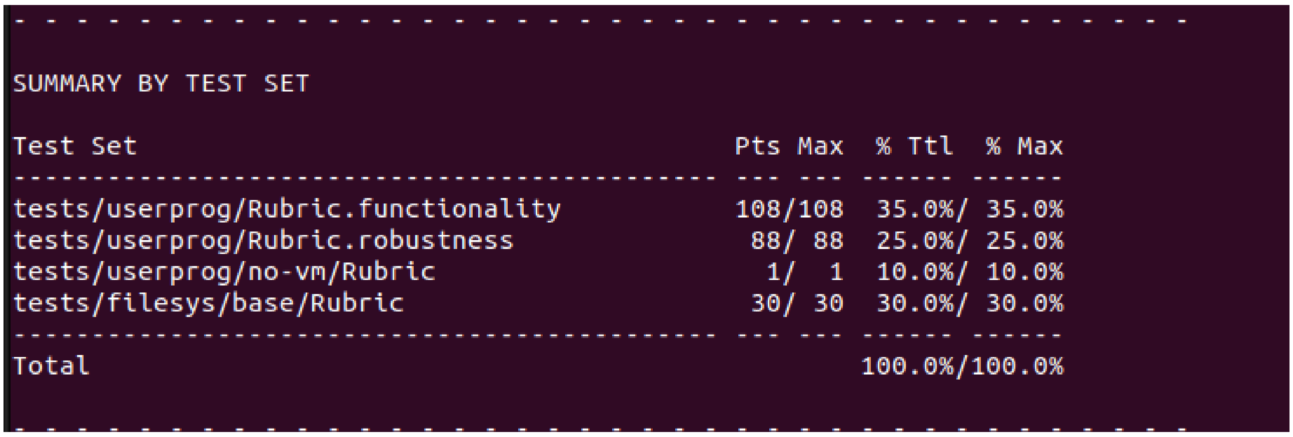
Disadvantages:
1) Consumes kernel space, user program may open lots of files to crash the

```
    kernel.
    2) The inherits of open files opened by a parent require extra effort to
    be
    implemented.
```

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

We didn’t change it. We think it’s reasonable and implementable.

Tests Check:



```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 

SUMMARY BY TEST SET

Test Set                                      Pts Max  % Ttl  % Max
--------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality           108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness               88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                     1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                      30/ 30  30.0%/ 30.0%
--------------------------------------------- --- --- ------ ------
Total                                                 100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - 
```