+-------------------------+
| CS 140 |
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+-------------------------+
---- GROUP ----
>> Fill in the names and email addresses of your group members.
Amira Mohamed <amirafathi3695@gmail.com>
Gehad Fathy <gehafathy99@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.
- nothing is new, as the argv array and addresses arrays are defined for each new process to be loaded.

----- 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?
Implementing argument passing :

1-split the file name in process_execute function and store them in
an array that is passed as argument to *start_process* function.
2-start_process () will take the first argument and also load()
function .
3-In load() we build our new stack :
to push new element decrement stack pointer and push them according
to the following order : the argument --> word align –-> their
addresses --> store the last address--> finally the size.
(To store them we use memcpy function)
arranging elements in the right order :
---
we loop on the array from the last element to the first .
Then store the addresses also in the same way also
to avoid overflowing the stack page:

We check the total size of the arguments passes if it will cause
overflow we free the page and exit .
>> A3: Why does Pintos implement strtok_r() but not strtok()?
Because strtok_r() takes save_ptr as parameter so it is provided by
the caller so if another thread call this function it sends another
save_ptr so it's more safe .
>> 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- decrease kernel time .
2-it's safer to parse and validate outside the kernel to avoid

problems that may happen if kernel take wrong arguments.

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.
struct child {
char* name;
tid_t tid ;
bool exited;
int exit_status ;
struct list_elem tid_elem ;}
we need it mainly in process_wait() as when the child exit its parent
can't find it's tid to return its status so we make this struct to
store tid and status as not to lose them if the thread exits.
struct file_attr
{
int fd;
struct file *f;
struct list_elem elem;
};
we use it to map between the file and its number as our system calls
take number and built in functions take files .
In struct thread :

{
1- struct list children :(of struct child) to store its children
2- struct list opened_files : (of struct file_attr)to store the
files which the thread opened and used.
3- struct semaphore sema_wait_load :
we use it to make the parent wait on the child until it is loaded to
add it in its children list.
4- struct semaphore sema_wait_exit :
we use it to make the parent wait on the child until it is exited and
return its status .
5-int next_file_id: its the number that thread give to its files .
And it's starting from 2 as 0 and 1 are assigned to STDIN and STDOUT.
}
>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?
Files descriptors are associated with open files using struct consist
of both of them.
file descriptors are unique within single process as the process set
number to files according to its next_file_id.
---- ALGORITHMS ----
>> B3: Describe your code for reading and writing user data from the
>> kernel.
1-First in syscall_hander() we validate the pointers , find the
number of system call then pop the system call parameter from stack.

2-Then call our system calls.
If WRITE system call :
if fd == STDOUT --> write to the kernel using putbuf(). Implemented on kernel\console.c
then map and get the file and call file_write() implemented in file.c
.
If READ system call :
if fd == STDIN → red from kernel using input_getc() .
then map and get the file and call file_read() implemented in file.c.
>> 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?
The least number is 1:
if all bytes stored in one page so there will be on call to the function.
the greatest number is 4096:
if each page is one byte so there will be 4096 calls
if 2 bytes of data :
the least number = 1 if the page contains more than one bit.
the greatest number = 2 if page contains exactly one page .
I think there is no room to improve .
>> B5: Briefly describe your implementation of the "wait" system call
>> and how it interacts with process termination.
Wait implementation
---
first we search for this given parameter(tid) in the list of children
if (!found) return -1 ;
else
if (! exited)
//wait until exit
sema_down(this thread → sema_wait)
return its exit status
interaction with process termination
---
in process_exit();
sema_up (thread-> parent-> sema_wait)
>> 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.

1-We valdite pointers using this function :
f (x == NULL)
return false
if (!is_user_vaddr(x)) // check if x <PHYS_BASE
return false
p = pagedir_get_page (thread_current()->pagedir, x)
if (p == NULL)//check if address mapped to the page directory
return false
return true
2- we call this function to valid esp pointer or any paramter of type
char *
3-to ensure that all temporarily resources are freed :
we call exit(-1) which call thread_exit --> process_exit()
-->close_all() and this function free all allocated structs
Example :
open syscall :
1- syscall_handler (struct intr_frame *f)
{
void* p =f->esp ;
if (!validate (p))
exit(-1)
switch (*(int *) p) {
Case SYS_OPEN: // it has one parameter .
{
void * a = f->esp +4 ;
if (!validate (a))
exit(-1);
if (!validate(*(char **)a))
exit(-1);
file = *(char **)a ;
f ->eax = open(file);
break ;
}
}
---- 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"?
1-By using semphore (sema_wait_load) to wait until executable loaded.
-- sema_down in process_execute() on the child after it is created
-- sema_up after calling load in start_process().
2- by use bool child_load which is set in load and tested in
process_execute()
if (!child_load)
return -1
else
return status its
>> 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?
1- when P calls wait(C) before C exits :
we use semphore (sema_wait_exit).
We use sema_down in process_wait() and sema_up in process_exit()
2- when P calls wait(C) after C exits :
we check in process_wait() if the child exited “this is a bool which
is set in exit ” if true so we don't wait and return it's status
if (!c->exited){
sema_init(&thread_current()->sema_wait_exit , 0);
sema_down(&thread_current()->sema_wait_exit);
}
3-when P terminates without waiting before C exits
c becomes orphan thread and the status of parent will be dying by
default.
4- when P terminates without waiting after C exits
no thing happen.
–---we always free resources in process_exit by calling close_all()
while (!list_empty(&thread_current()->opened_files)){
struct file_attr* fileAttr =
list_entry(list_pop_front(&thread_current()->opened_files), struct
file_attr, elem);
file_close(fileAttr->f);
free(fileAttr);
}
while (!list_empty(&thread_current()->children)){
struct child*c =
list_entry (list_pop_front(&thread_current()->children), struct
child, tid_elem);
list_remove(&c->tid_elem);
free(c);
}
---- RATIONALE ----
>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?
Because I need to check if the pointer in userspace and
“is_user_vaddr” does this. And I need to heck if address mapped to the
page directory and also pagedir_get_page does this.
We choose this method because it's more easy to us as it uses built
in functions
>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?
1-advantages :
the struct is an easy way to map .
2-disadvantages :
it may waste the memory .
>> 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 logical.