+--------------------------+

| CS 153 |

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

+--------------------------+

---- GROUP ----

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

Ryan Rivas <rriva002@ucr.edu>

Jeremy Mah <jmah002@ucr.edu>

---- 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.

thread.h, struct thread:

struct semaphore sleep\_sema;

-A semaphore to put the thread to sleep.

Same implementation as project 1.

int return\_status;

-Stores the thread's return status.

bool waited;

-Stores whether or not the thread has already been waited upon.

---- 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?

The first argument follows the program name and continues for each

additional argument when theres a space between them. We use strtok\_r

in our line token = strtok\_r (file\_name, " ", &save\_ptr), which takes a

string, the space and a pointer to keep track of parsing the input. To

make sure that the elements of argv[] are in order, the addresses of the

arguments are pushed into a stack from end to beginning. If we need to

get the arguments back, we just pop from the stack since it has the right

order of arguments. We have several checks to ensure that there is no

overflow on the stack page. There is a check for argument length, we

aligned the stack pointer to a number divisible by four, and we checked

to make sure the stack doesn't overflow when we push in arguments.

---- RATIONALE ----

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

Pintos implements strtok\_r() over strtok() because strtok() has potential

race conditions and isn't thread-safe because it uses a static pointer.

strtok\_r() works in multi-threading and is thread-safe.

>> 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.

1. In unix systems the separation enables more complex commands like

pipelining. This isn't possible in Pintos unless the Pintos kernel is

heavily changed.

2. Parsing occurs in the shell instead of kernel space. This makes it a

lot safer because the shell could help check unsafe command lines, which

saves complexity for the kernel.

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.

#define MAX\_FD 128

-boundary for max file descriptor MAX\_FD = 128 (section 3.4.2 of the

Pintos documentation allows for an arbitrary limit of 128 open files).

static struct file \*files[MAX\_FD];

-creates an array of file pointers.

>> B2: Describe how file descriptors are associated with open files.

>> Are file descriptors unique within the entire OS or just within a

>> single process?

When a process opens a file, that file is placed in the array of open

files; the file descriptor is related to the index where that file was

placed. File descriptors are unique within a single process.

---- ALGORITHMS ----

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

>> kernel.

We first check if the file descriptor is valid. Next we check if the

input address is valid. If something is invalid, the syscall will exit.

We then finally return the value we get from either reading or writing

into eax.

>> 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?

1. least: 1 greatest: 8192 (calling pagedir\_get\_page())

2. least: 1 greatest: 2

3. An improvement can be to skip the validation phase, but this can be

dangerous.

>> B5: Briefly describe your implementation of the "wait" system call

>> and how it interacts with process termination.

The wait system call calls process\_wait, which gets the thread of the

child process. We then check if the process is either null, or waited by

parent, if it is, then we return -1. If it hasn't exited and is not

waited, then we set the waited flag to true. The thread is then added to

the list of waiters and the parent process is blocked. It is unblocked

when the child process exits.

>> 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.

The address are validated before reading data to make sure there is no

bad pointer. If we detect any problems for the process to execute

successfully, we would kill the process, before starting it. All

temporarily allocated resources are freed when the process exits, which

happens in sys\_exit. It is responsible for freeing the stack, closing

files, and cleaning up all children. This ensures that everything will be

freed upon exiting.

An example would be when this error happens, the process would go through

thread\_exit then process\_exit. The terminated process's resources will be

all freed, then the executing file of the process will be freed as well.

Finally we free the page directory at the end of exit. This freeing

prevents memory leakage.

---- 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 code ensures it by using semaphores to wait on the load to finish.

Success/failure status is passed back to the thread when the parent calls

exec() to read the information.

>> 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?

The parent process holds a semaphore until the child exits. After the

child exits, the wait struct will still exist, but the semaphore will

have a value of 1, so when the parent calls sys\_wait, it will return

immediately. All resources are feed in each case because the parent

releases all its resources when a process is exited.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

We chose this method because it simply worked with what the documentation

required us to do. Other ideas did not seem ideal at the point of

choosing implementation methods.

>> B10: What advantages or disadvantages can you see to your design

>> for file descriptors?

An advantage is that every thread has its own set of fds. Our design

makes it easy to find files opened by a process by using the fd. A

disadvantage is that in order to get a list of all open files, it

would take a very long time because of the implementation.

>> 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 this. We used tid\_t.