+--------------------------+  
 | CSE 451 |  
 | PROJECT 2: USER PROGRAMS |  
 | DESIGN DOCUMENT |  
 +--------------------------+  
  
---- GROUP ----  
  
>> Fill in the names and email addresses of your group members.  
  
Hampton Terry hterry@uw.edu

Kevin Tang kevin.tang2@gmail.com

Matthew Dorsett madman2@cs.washington.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.

In syscall.c:

#define NUM\_SYSCALLS 32

-A global constant for the number of system calls our system supports. Currently 32 based off -of the number of system calls appear need to be implemented in future pintos assignments.

static void syscall\_handler (struct intr\_frame \*);

typedef void \*(\*handler) (void \*arg1, void \*arg2, void \*arg3);

static handler syscall\_vec[NUM\_SYSCALLS];

-A vector to store system calls numbered by their system call numbers as indices

---- 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 the file\_name\_ parameter of the start process function which where we implemented argument passing, we are given a string of arguments separated by spaces.

The first step we take is to tokenize the string by spaces which turns all spaces into null terminated characters. While the string is tokenize, we also save off the addresses for each argument in the string as we examine them.

Next, we pass the first token into the load function which loads our program into memory from disk.

At that point we begin adding arguments onto our stack, starting with the entire string that we tokenized in the beginning. We then word align the stack, and begin pushing the addresses of each word in our string in reverse order onto the stack. This is followed by a pointer to our char \* array of arguments, followed by the number of arguments, then the return value.

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

Strtok\_r ensures that the save pointer of strtok is stored on the stack in the context for a particular thread whereas with strtok, there is nothing that guarantees this. This is important because when when multiple threads try calling strtok, there might be a single global saveptr which would cause race conditions between threads using strtok.  
  
>> 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.

The first advantage is that this allows for there to be less time and operations spent in the kernel. If something goes wrong with argument parsing, this would only affect the shell user program whereas in pintos’ case where the parsing is done in the shell, if something were to go wrong it has potential to be disastrous because it is in kernel mode.

Unix also implements more complex features in the shell such as piping and redirection in which operations are already done on the string. Pintos could get these working as well with likely either parsing out special symbols for then passing the entire argument strings to pintos or passing the entire string to pintos and letting handle the piping and redirection in the kernel.

With the first option, these special symbols and arguments are already being parsed in some way so it would probably be much easier to just pass the parsed arguments to the kernel.

With the second option, this would add a great deal of unnecessary complexity to the kernel for operations that should be able to be done in user mode.

So, likely the unix argument passing has an advantage over pintos in this respect.  
  
 SYSTEM CALLS  
 ============  
  
---- FROM USERSPACE TO THE KERNEL ----  
  
>> B0: When syscall\_handler() is called, where is the system call number  
>> stored? Where are the arguments to the system call, if any, stored?   
>> Give the backtrace of the stack during syscall\_handler() (be sure  
>> to resolve addresses both in user space and in kernel space!) and  
>> give your answer in relation to this backtrace. Explain why the  
>> syscall number and arguments appear at this place in the stack.

The system call number is stored on the top of the stack frame of the interrupt handler, followed by the arguments. Here is a snapshot of the stack of the user calling function when a user program attempts to use the write() system call:

%esp Sys Call # Param #1 Param #2 Param #3

0xbfffff0c: 0x0000000c 0x00000001 0x0804be40 0x0000000d

And here is a backtrace of a call to write()

#0 syscall\_handler (f=0xc010bfb0) at ../../userprog/syscall.c:65

#1 0xc0021928 in intr\_handler (frame=0xc010bfb0) at ../../threads/interrupt.c:373

#2 0xc0021b2f in intr\_entry () at ../../threads/intr-stubs.S:37

#3 0xc010bfb0 in ?? ()

#4 0x0804a444 in write (fd=514, buffer=0xbfffff0c, size=35) at ../../lib/user/syscall.c:138

At #4, the user makes a call to write(), which we know is in user-space because the address < PHYS\_BASE. Then it traps to the kernel, entering kernel-space at address 0xc010bfb0 which is greater than PHYS\_BASE. A struct intr\_frame is created on the kernel stack, but it stores the state of the user calling function’s stack, allowing it to access its stack pointer and thus A) the system call number, and B) the parameters. The system call number and the parameters are stored at this location because this facilitates communication between user-space and kernel-space; the user-level function only has to push its arguments onto the stack like it would for a normal function call.  
  
---- 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.

Thread.h:

struct semaphore wait\_sema;

Synchronizes the parent waiting for the child to finish

struct semaphore wait\_on\_parent;

Synchronizes the child waiting on the parent to finish

struct semaphore load\_sema;

Synchronizes process\_execute and start\_process  
  
struct list child\_list;

A list of all children for a thread

struct list\_elem child\_list\_elem;

An element of the thread that is to be added to the child list of any parent thread

struct thread \*parent;

A pointer to the parent thread

struct file \*program;

The file that is associated with the program. Used to deny writes to executables

struct file \*fds[128];

The array that stores all the files for a particular process where the file descriptors are the indexes associated with the files in this array.

Syscall.c:

struct pipe\_buffer{

char buffer[PIPE\_BUFFER\_SIZE]; //pipe buffer

int start; //pointer to next char to be read from buffer

int end; //pointer to index to write next char to buffer

int fd\_read; //FD for read end of pipe

int fd\_write; //FD for write end of pipe

int size; //number of chars stored in buffer

};

Stores a buffer and the file descriptors that mark its read and write ends, along with pointers to the current read and write locations.

struct pipe\_buffer \*pipe\_buffer\_array[MAX\_PIPES];

A global array of pipes to allow for interprocess communication.

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

We implemented file descriptors so that each process will have its own set. We have a file descriptor array in thread which stores the files associated with the file descriptors which are the indices of the array.

Every time a file is opened, we find the first available file descriptor in our array and set that struct file \*array to the call to filesys\_open. This way when given a file descriptor we can jump to that particular index of the array. If we find a file located in that index, we know this is the file we are looking for. Otherwise, if we find NULL, we have an unused file descriptor.

For closing files, se just call file\_close on the file located at that particular fd index, and then set that index to NULL.

File descriptors for pipes work differently because of the requirement that pipes should be allowed to do interprocess communication. The array of file descriptors for pipes is stored in a global struct. In which all threads will use to hand allocation of pipe allocation.

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

Our read and write syscalls handle two general cases:

-The FD references a regular file stored on disk

-The FD references an end of a pipe.

In our implementation, file descriptors 3 to MAX\_FD - 1 are used to reference open files. File descriptors 0, 1 2 are reserved for stdin, stdout, and stderr. Each thread has an array of pointers to open files. When the read or write syscall is called, the thread’s array of file descriptors is checked. If that index is null, no file exists. Otherwise, the index will point to the open file. At this point, we use the file\_read or file\_write function to process the system call.

File descriptors MAX\_FD to MAX\_FD + MAX\_PIPES - 1 are used to represent one end of a pipe. Because pipes need to be able to communicate between individual processes, we defined an array of MAX\_PIPES struct pipe\_buffer in syscall.h. Each struct pipe\_buffer is referenced by two file descriptors, one for the read end, and one for the write end. Each struct pipe\_buffer contains a circular buffer, with pointers to the next index to read and write to.

The last case handled by the read and write system calls handles invalid input. If the pointer to the buffer is an invalid address, exit(1) is called. If the FD is invalid, the function returns -1 to indicate error.

Read and write syscalls also acquire a lock before they make any changes to the file saved on disk. This is to avoid any synchronization issues of multiple threads trying to read or write to the same file.

>> 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?

If a system call requires a full page of data, two calls to pagedir\_get\_page would be needed. The first would always be necessary for the getting the page of the start of the region of data

but if the data spanned two pages, then a second call would be necessary. Since we’re limited to just a page of data only two checks would ever be necessary.

In the case that only copies two bytes, the data will likely only span one page but it is entirely possible that it doesn’t, in which case, we would again need to check two pages.

An improvement could be using a different method where the page tables are not checked at all and a program is allowed to page fault as soon as it accesses a region of memory that is not assigned. In this case if all goes well, no inspections would be necessary. If the program page faults, then the fault is handled accordingly at which time the pages are inspected.

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

The implementations of wait and exit are tightly coupled together because there is a great amount of interaction between the parent and child to coordinate when each is allowed to exit and finish terminating its process.

The wait system call works by calling process\_wait which, once it determines it is a valid call to the function (the pid passed in exists, and is a child of the current thread), will down a semaphore which causes the current thread to wait until the child process finishes. Immediately after, the parent thread will up a different semaphore to allow the child to finish exiting because the child thread is already waited for by the parent.

Process\_exit, which is called through the exit syscall, has corresponding interactions with the semaphores. When a thread reaches exit, it will up a semaphore to tell its parent that it has exited, then immediately down a semaphore in order to wait until either (1) the parent calls wait, or (2) the parent itself exits. This is done so that a parent has the opportunity to call wait on a child thread even if the child thread has already finished running. This means that in exit, the current thread must up the “wait\_on\_parent” semaphore for all of its children.

>> 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 main thing done to not obscure the system calls in error code is to provide functions in which we check for valid pointers and file descriptors and anything else they pass us that maybe be bad arguments. This allows for simple function calls that can be used across all the system calls to do these checks and obscure away all the checks necessary such as checking for NULL, resolving the address of the page, etc. The examples of this are our verify\_ptr and verify\_fd functions.

The other thing we do is to always do these checks and exit as early as possible when necessary. There is a pattern of doing the checks all at the beginning and the termination all at the end or in if statements that include returns. It is generally not too difficult to distinguish where the significant portions of the system calls are and where the error handling is.  
  
---- 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"?

The behavior of exec is that it replaces the current process with the new exec’d process. This is done mostly by the call to “load” inside start\_process\_exec. The call to this function returns an error code if it fails. By checking this error condition, we can tell if the call to exec worked or not. If it didn’t work we can simply return -1 and propagate the error code up. If all goes well however, the call to exec will not return at all because it will be replaced by a different process.

There is not much in terms of synchronization for the exec system call, however, we have a separate process\_execute function, which does a forking exec. This one, in particular, does require some synchronization because the newly created thread immediately calls start\_process, but we can only know if it succeeded or failed after a call to “load”. The parent thread calling process\_execute must then wait for the child to finish loading by downing the “load\_sema” semaphore, and when the child either fails or succeeds, it ups the semaphore telling the parent it can continue.

>> 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?

For parent P and child C, there are many ways in which wait can run. If P calls wait(C)

before C exits:

P will block inside wait until C reaches exit, allowing P to continue. C then waits inside exit until P finishes wait. Both threads can continue after this point.

after C exits:

C exits first, so it ups a semaphore then waits inside exit until P does its call to wait. P can go through wait without waiting because C has already upped the semaphore. P then allows C to unblock and both threads can finish.

Or if P terminates without waiting:

before C exits:

When P exits, it removes itself as the parent for all of its children. When C reaches exit, it no longer has a parent, and can finish exit without waiting.

after C exits:

C will wait inside exit until woken up by the parent. If wait is not called, C will not be woken up until P finishes its call to exit where it will up the semaphore for all its children, allowing the children to finish process termination.

In the case of freeing up resources, process\_terminate handles all of this. Once a thread has finished it’s synchronization steps with wait, it will go and free up resources such as closing file descriptors and allowing writes back to the executables.

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

We implemented this by checking for valid addresses and arguments and handling them appropriately. Some errors such as passing an invalid file descriptor call for the system call to return an error code whereas others such as passing in an invalid address call for the program to exit. How we handled invalid user addresses is by using pagedir\_get\_page function which tries to resolve the address passed in. If this function returns NULL signifying that the virtual address can not be resolved, we know that it is an invalid pointer and handle it appropriately.

We chose to handle this feature in this way because of the simplicity of checking if the address was inbounds. This meant, however, that we would have to check the addresses in multiple locations, such as for read and write, we would have to check in two locations, both the beginning and the supposed end of the buffer.

The other option was that we let the the page fault and handle it accordingly there, which seems more efficient but it seems a little bit easier to get wrong and more difficult to do. We went with the simpler and safer but slower solution.

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

With our current design of file descriptors, we have an array of file pointers stored in each thread. It always requires that we always allocate that amount of space but the advantage we gain is that it is very easy to look up a file descriptor when given one. All we have to do is jump right to that index and see if there is a file located in that region.

A disadvantage to our system when pipes, and eventually other kinds of file descriptors are included that are not limited to interacting with just files. We decided to block of regions of file descriptors for various purposes, like files and pipes. This still allows for easy translation of file descriptors however we aren’t allowing our system to dynamically allocate descriptors for different purposes because we are presetting a limit to each region of file descriptors.  
  
>> 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 did not change the tid\_t to pid\_t mapping. This is because in pintos there is one thread per process system, so it would seem unnecessary to do anything but the identity mapping.  
  
 SURVEY QUESTIONS  
 ================  
  
Answering these questions is optional, but it will help us improve the  
course in future quarters. Feel free to tell us anything you  
want--these questions are just to spur your thoughts. You may also  
choose to respond anonymously in the course evaluations at the end of  
the quarter.  
  
>> In your opinion, was this assignment, or any one of the three problems  
>> in it, too easy or too hard? Did it take too long or too little time?  
  
>> Did you find that working on a particular part of the assignment gave  
>> you greater insight into some aspect of OS design?  
  
>> Is there some particular fact or hint we should give students in  
>> future quarters to help them solve the problems? Conversely, did you  
>> find any of our guidance to be misleading?  
  
>> Do you have any suggestions for the TAs to more effectively assist  
>> students, either for future quarters or the remaining projects?  
  
>> Any other comments?