CS 162
Project 2: User Programs
Design Document

Fill in the following information:

GSI:
Group Number: 102
Andrew Blatner <ablatner@berkeley.edu>
Bryan Tran <btacju@gmail.com>
Noah Ruderman <noahc66260@gmail.com>
Kevin Lam <kel018@berkeley.edu>

Preliminary Questions
==========

1) Please write down any miscellaneous comments or notes for your GSI.

2) Currently, process_wait tries to decrement a semaphore called temporary. Explain why this implementation of process_wait wouldn't work if the exec system call were implemented.

The counterpart to process_wait() is process_exit(), which calls sema_up(). If we were to use fork() and exec(), the child process wouldn't share memory with the parent process and so the call to sema_up(&temporary) would fail to modify the temporary variable in the parent process.

3) For checkpoint 1, you are implementing a dummy version of write that prints to stdout. Describe, in two sentences or less, how a user could break the operating system exploiting this dummy version.

The implementation does not check whether or not the pointer passed is valid or not. A user could break the operating system by passing userspace pointers into write() that do not point to valid accessible memory, causing crashes.

4) Briefly describe how the syscall interface works. How does the operating system obtain the syscall arguments passed from userspace?

First the caller pushes the system call arguments and then the system call number on to the stack. Then, when the system call handler is invoked through `int $0x30`, the system call

number is right at the stack pointer and the arguments are located at subsequent increasing addresses. This way, all the argument validation and handling is done from within the kernel.

5) In filesys.c, you'll see various filesystem functions that work on struct file objects. In all modern operating systems, these filesystems operations are abstracted away from userspace code; why would it be a bad idea to expose these operations to users?

Filesystem operations can be harmful when misused. For example, if certain filesystem functions were called, they could delete important files or even completely clear out entire disks. It's for user security, as filesystem operations could potentially be used maliciously.

6) Run pintos --gdb -v -k --qemu --filesys-size=2 -p tests/userprog/iloveos -a iloveos -- -q -f run iloveos, and use GDB to print the address of the priority field of idle thread. Do this again; does the address of the priority field change? We expect you to be able to use GDB comfortably throughout this project; if you're struggling with this, we strongly encourage you to come to office hours.

The address of the priority field of idle thread is 0xc010401c. The address doesn't change every time the program is run.

Process Syscalls
==========

7) Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration for implementing halt, exec, exit, and wait. Briefly identify the purpose of each.

```
static struct list process_list; // List of PCB's

/*
 Contains information and synchronization primitives needed to
 implement system calls. */
struct pcb {
     int pid; // This process's ID
     int ppid; // Parent process ID

     struct list child_list; // Contains list of child processes
     struct list_elem parent_list; // list_elem for parent's list

     /* Held by process for its lifetime. When parent is able to
acquire it, it knows the child has exited or been terminated.
     struct lock wait_lock;
     int exit_status; // Stores exit status for use by wait
```

```
     struct lock load_lock; // Held for duration of process load
     int load_status; // Used by exec to indicate success or
failure.
     struct list_elem pcb_elem;
}
```

8) Briefly describe your planned implementation of the "wait" system call and how it interacts with process termination.

When a process calls “wait”, it iterates through the process list and searches for a pcb with the child process id. If the id is not found or the parent id is not the callers id, it returns an error. Otherwise, it calls wait_lock.acquire(). When the child terminates, the exit status is saved to exit_staus and wait_lock is released. Then, if the parent calls wait before the child finishes, the parent blocks until the child finishes and releases the lock. If the child finishes before the parent calls wait, the lock is left free and the exit status is saved.

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

To ensure that, you have to be able to check if loading the new executable fails. This can be done by using two variables, one to keep track of the load status and a semaphore to make sure the parent returns after the child. The load status variable can be updated in start_process() in userprog, which is where processes are started and loaded. There we can see whether the load succeeded or passed, so we can keep track of the load status. To make sure the parent process does not return before the new executable is done loading, we initialize our semaphore to 0. In our parent process, we call sema_down(), and in our child process after load() is called, we call sema_up() in order to signal to the parent process that we are done loading, so that it can continue running.

10) 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 ensure proper synchronization using the wait_lock semaphore. C acquires wait_lock right when it is created, and it does not release it until just before process_exit() returns. If P calls wait(C) before C exits, it will not be able to acquire wait_lock, so it blocks until C completely exits. If P calls wait(C) after C exits, it will be able to acquire wait_lock immediately. In either

case, the call to wait(C) is what frees the resources related to C. If P terminates before C, it can set C's list_elem parent_list to NULL. Then, when exiting, C can check if parent_list is NULL, and if it is, free its own data.

wait: Attempts to acquire child's wait_lock. When it does, frees child's data.
process_exit: Sets all children's parent_list members to NULL. Checks if own parent_list is NULL. If so, free's own data. If not, saves status and exits.

11) How are you planning to implement reading/writing user memory from the kernel and why did you decide to implement it in this way?

Reading/writing user memory can be done by first checking the validity of the user virtual address provided by converting it to a kernel virtual address. Then we need to check whether this address exists between the bounds PHYS_BASE and maximum address of 0x08048000. If it exists, we can dereference it and use it in a syscall. The arguments also need to be converted to kernel virtual addresses and checked for validity. We chose this method initially because it is easier to implement which reduces the chances of more bugs in our code.

Argument Passing
==========

12) Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration for argument passing. Briefly identify the purpose of each.

#define MAXARGV = 151; \\ to impose the max number of arguments a program can take

13) Briefly describe your planned implementation for 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?

After parsing the elements into argv[], to keep them in the right order we first push the address of each element onto the stack in reverse order. Then, push a null pointer, to keep the front of the stack. This also makes argv[argc] null, which is in line with C standard. While pushing the addresses of the elements onto the stack, we count how many so we can try to keep it under a page limit of 4kb. You can store 4096 characters on a 4kb page, so we limit the number of elements of the stack to be 150. We then push the pointer to argv[0], argc, and a return value.

File Operation Syscalls
==========

14) Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration for the file operation syscalls. Identify the purpose of each in 25 words or less.

```
static struct list all_open_files;

static struct list avail_fd; // list of available file descriptors to
be initialized to {2, 3, … , 127}

/* Contains per file information /*
struct file { // struct for use in per-process file list
     int fd; // file descriptor
     int refs; // Count of processes with file opened

     /* Set true by remove function
        Afterwards, it's not deallocated until refs == 0*/
     bool removed;

     /* Used for membership in all_files of avail_fd.
        No file can be in all_open_files and avail_fd at once. */
     struct list_elem elem;
}

/* Contains per process information about the file /*
struct file_ptr {
     struct file *f; // Pointer to file struct in all_files list
     int pointer; // read/write location within file
     struct list_elem pcb_elem; // Used for membership in open_files
}

struct pcb {
     …
     struct list open_files; // list of files (struct file_ptr)
     …
}
```

15) Describe how file descriptors are associated with open files in your design. Are file descriptors unique within the entire OS or just within a single process?

To keep track of file descriptors, we will keep track of them individually within each process, so they will be unique within a single process. Each process will have a list to keep track of all the files and file descriptors. Whenever a file is opened, a file pointer is added to the list, and the file struct also contains the file descriptor. Whenever we want to access the file, we can search through the list looking at all the file descriptors to get the right file. File descriptors will be unique within each process since each process has its own list of files it uses. We would reserve the first two places in the list to be NULL, to represent STDIN_FILENO and STIDOUT_FILENO.

Whenever we add a file pointer to the list, we have to assign it a file descriptor, so we keep an ordered list of file pointers, from 2 to 127. We just pop off the first element of the list to use as the file descriptor every time we add a new file pointer to the list. If the list is empty, this means we have reached our 128 open file limit and can't open more files.

16) What advantages or disadvantages can you see to your design for file descriptors?

One advantage with this design is allowing multiple processes to open and reference a file at the same time. A benefit of the `removed` flag is that the file does not need to keep track of the processes that have opened it. A disadvantage is that every time we want to access a file we have to search through the whole list. On the other hand, our list is dynamic, so we only use as much space in each process as we need. We also would have to keep track of all the possible file descriptors to use in a list. Using arrays would save time when we want to access the files, but it would waste space, as you would have to make a static array of size 128 in every single process. An array would also make it easy to keep track of file descriptors, as you just take the index as the file descriptor.

17) 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?

If a system call causes a full page of data to be copied from user space to kernel, the least amount of inspections possible of the page table would be one, since if all the data lies on one page, we will only check one page for it. The greatest possible number of inspections would be 2, since the full page of data could be across two pages.

For copying 2 bytes of data, the answer would be the same, since both bytes could lie on the same page. One byte could also be on one page, and the other byte on the next page, so it might require 2 calls, although the chance of that would be much lower than in the case of the full page of data.