```
  +-----------------------+
|   Operating Systems   |

| PROJECT 2: USER PROGS |

|    DESIGN DOCUMENT    |
  +-----------------------+
```

Team White

Stefani Moore <stefani.moore@ucdenver.edu>

Shawn Johnson <shawn.a.johnson@ucdenver.edu>

Lena Banks <lena.banks@ucdenver.edu>

Sara Kim <sara.kim@ucdenver.edu>

Tests Passed

80 of 80 tests passed.

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.

typedef struct user_program /*Used to hold the file_name, argument count (argc), and extracted command line arguments (argv)*/

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 process_exectute(), a copy of the command line string is made so that calling strtok_r does not corrupt the original command line string. It is then tokenized to extract the file_name inorder to call

thread_create, which will lead to start_process(). Within the start_process function the entire command line is parsed and it's arguments, the argument count, and the file name are saved in the user_program struct. The arguments are extracted in order and placed in consecutive spots of the args[] array. The stack is then initialized with each argument in the correct order ensured by our args[] array, and all other information that the process needs to be there. A simulated return from an interrupt then forces the thread to start.

RATIONALE

A3: Why does Pintos implement strtok_r() but not strtok()?

In Pintos strtok_r is implemented instead of strtok because it is more threadsafe. Strtok_r is reentrant, to avoid the case where another thread gains control and also calls strtok, which would change the savepointer. When the original thread regains control, it would pick up where the other thread's strtok left off. With strtok_r, we provide the saveptr, so we the problem is avoided.

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 Unix approaches shortens the time inside kernel and offers more robust checking by checking whether the executable is there before passing it to 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.

struct child_parent /*Child process struct to indicate child's status*/

struct user_syscall /*Holds parsed arguments and syscall_index extracted when syscall handler is reached*/

struct process_file /*For process files with file descriptor, file that is open pointer, and shared elem*/

struct lock file_lock /*Global file lock*/

> In the thread struct

struct file* cur_file; /* Pointer to the currently executing file */

struct list files; /* List of open files */

```
struct thread *parent;         /* Parent thread */
struct semaphore child_sema;    /* Child lock used to wait the child */
struct list child_processes;    /* List of child processes */
bool success;                  /* Used to track status of the process */
int fid_count;                 /* File descriptor count */
int waiting_on_thread;         /* Thread tid we are waiting on */
int exit_code;                 /* Exit code */
```

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

The file descriptors are unique to each individual file that is opened by calling the open system call. These files get added to the current process's list of open files and assigned this unique file descriptor. The uniqueness, occurs within a single process only and when a file gets closed its file descriptor is freed and can be reused by the system later for another file. If the file is open and gets reopened the file will get a new file descriptor each time this occurs.

ALGORITHMS

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

In the syscall handler, when the read function is called a file lock is acquired and the current file is found with the process_get_file that searches through the open file list looking for its file descriptor. If process_get_file returns NULL meaning there either was no file or the file could not be read then the lock is released and -1 is returned. If a file is found then the file_read function is called with the current file passed to it followed by the file lock being released and an unsigned int size being returned from the read function.

The read fuction is almost identical to the write function, but instead of calling file_read, file_write is called.

Both the read and write functions are accessed through system calls and expect a file descriptor, pointer to a buffer, and a parameter specifying the size of the buffer.

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 of possible calls to pagedir_get_page(), or inspections, is 1 and the greatest number is 4,096. The least number of possible calls is the case where a segment size for ELF in memory is greater than or equal to the segment size in files. The greatest number results when each ELF is only one byte in size and 1 inspection is required for each byte. For 2 bytes, the least number is 1 and the greatest number is 2 following the same argument as mentioned above.

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

When process wait is called, it loops through the list of the process' children list to check if any of the structs have a matching tid. If no match is found then -1 is returned. If a matching tid is found, the child_parent struct contains a boolean variable has_exited which is indicate whether or not the child has finished processes. If the child is still alive, we will wait for it to die, using a lock and condition through cond_wait. When the process is done it is removed from the list and the exit_status of the process is returned.

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.

Before any system calls are actually 'called' the arguments are extracted from the stack and checked that they are valid. This checking is done by the user_memory_ok function. After this preliminary check the system call handler continues or exit(-1) is called if not. If the arguments are invalid then no locks or buffers are allocated. When a page fault occurs, the process exits and all files are closed. Since all of the checking is done prior to the functions being called this avoids the primary function of our code from being obscured.

For an example, a write system call will first get the arguments checked and extracted through the syscall_handler. The user_syscall struct is populated with an array of the arguments, a syscall_index, and an argument count. We then identify that the syscall is a write system call and call the write function where the file descriptor, buffer, and buffer size are passed to the write function. At this point all arguments have been previously validated and therefore no more validation is needed and we are free to execute the write function if there is something to be written to the file.

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"?

Every thread struct has a semaphore called child_sema that when process_execute finishes and returns a new process's ID to a call to exec the parent thread finds the newly created process in the list of its children based on its thread identifier. Then it downs the new process's semaphore which ensures proper synchronization with the code responsible for loading a process.

If a process loads correctly, or if it fails when trying to load, its semaphore is 'upped' so the parent process knows that loading finished. Before the semaphore is upped a boolean flag is set to indicate success or failure of loading which is used to let the parent know the result of the loading.

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?

Proper synchronization is ensured by using a semaphore with every thread called child_sema. When P calls wait(C) before C has exited, the child_sema is downed and initialized to 0. When C finally exits, then the semaphore is upped.

RATIONALE

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

User memory is checked before access, which made the most sense to us that it be checked from the very beginning.

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

Advantages would be that the same structure can always be used to store the necessary file information and since each thread has its own list of files there is no limit (besides memory) on the number of open files/descriptors. The disadvantages are that there can exist many duplicate structs for stdin and stdout and that accessing a file descriptor requires iterating through the list, which is done in linear (O(n)) time.

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

Not applicable.

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?