Advanced Operating Systems
Project 2: User Programs
Design Document

The questions in this design document should reflect the design of the code you wrote for the project. **Your grade will reflect both the quality of your answer in this document and the quality of the design implementation in your code.** You may receive partial credit for answering questions for parts of the project that you did not get to implement, but you must indicate in your answer that there is no corresponding implementation, or you will not receive any credit.

For each question, **you should include both the name of the file(s), function name(s), and the line numbers where the relevant code may be found** – both the code that answers the question directly and any function that you refer to in your answer.

These design documents will be completed and submitted as a group. Please use this document as a guide for design and discuss the questions and their potential answers prior to beginning implementation. **All text in *italics* is intended to be replaced with your answer**. When you have completed your design document, submit it to the Canvas assignment Project 1 (Threads): Design.

Team Information Fill out the following information for each member of your team.

Name: *Yunseok Kang*
EID: *yk9784*
Email: *yunseok.kang@utexas.edu*
Unique Number: *51275*

Name: *Dohyun Kwon*
EID: *dk29493*
Email: *gundo0109@utexas.edu*
Unique Number: *51275*

Name: *Stefanus Adrian*
EID: *sja2673*
Email: *stefanus.adrian@utexas.edu*
Unique Number: *51275*

Preliminaries Please provide any preliminary comments and cite consulted sources.

If you have any preliminary comments on your submission or notes for the TAs, please give them here.

Your answer 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.

Your answer here.

ARGUMENT PASSING (10 points)

The 10 total points are divided equally among the following sub-questions.

A1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary to implement argument passing. Identify the purpose of each in 25 words or less.

1. Global variable in process.c:

- `PARSE_BUFFER_SIZE 100; (line 25)` - serves as a predefined constant to specify the maximum size of the buffer used for parsing command-line arguments. It defines the size of the arrays used to store copies of the command line (fn_copy and fn_copy2) and to hold pointers to the individual arguments (argv). This constant ensures that the arrays are sufficiently large to handle the expected range of input while also setting a limit to prevent excessive memory allocation.

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? Is it enough to limit the size of the incoming commandline? Why or why not?

Argument parsing implementation:

1. **Copying and Tokenizing the File Name:** The process_execute function makes a copy of the file_name (which contains the program name and its arguments) and tokenizes this copy to separate the program name from its arguments. The tokenization is done using strtok_r to handle spaces between arguments.
2. **Opening the File:** Before creating a new thread for the process, the function checks if the file can be opened using filesys_open. If the file cannot be opened, the function returns an error.
3. **Passing Arguments to start_process:** The original file name (including arguments) is passed to the start_process function via thread_create.
4. **Parsing Arguments in load:** In the load function, the file name and arguments are again tokenized using strtok_r. The arguments are stored in an array argv, and the number of arguments is counted in argc.
5. **Pushing Arguments to the Stack:** The implementation sets up the user program's stack in the load function. This involves pushing the arguments (argv), the number of arguments (argc), and a fake return address onto the stack. The arguments are added to the stack in reverse order so that they can be correctly accessed by the user program according to the C calling convention.
6. **Aligning the Stack:** The stack is aligned to ensure it meets the x86 architecture's requirement that the stack pointer (esp) be aligned to a 4-byte boundary.
7. **Setting the Entry Point:** The eip register is set to the entry point of the user program, ensuring that execution begins at the correct location within the program.

Order of argv[ ] Elements: The elements of argv[ ] are pushed onto the stack in reverse order (from the last argument to the first). This reversal is necessary because, in the C calling convention,

argv[0] should be the first element to be pushed onto the stack, but the stack grows downwards. By pushing arguments in reverse, when they are accessed in forward order during program execution, they appear in the correct order (argv[0], argv[1], ..., argv[argc-1])

Avoiding Stack Overflow: This management includes calculating the exact amount of space needed for all the arguments, the padding for alignment, and the additional pointers (such as the argv array pointers and the fake return address). By keeping track of the amount of data pushed onto the stack (cnt_byte) and ensuring that this does not exceed the page size (typically 4KB), the implementation can avoid stack overflow.

Limiting the Command Line Size: Limiting the size of the incoming command line is an essential but not entirely sufficient measure to prevent stack overflow. It ensures that an excessively long command line does not lead to immediate overflow. However, this alone is not enough because the total space required on the stack is not just dependent on the command line size but also on the number and size of the arguments and the required padding for alignment. Therefore, while limiting command line size is a necessary check, the implementation must also consider the total memory footprint of the parsed arguments and auxiliary data structures on the stack to ensure safety against overflows.

Related functions and variables in process.c:

- load (line 285)

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

strtok_r() is a re-entrant and thread-safe version of strtok(), distinguished by its use of an additional argument provided by the caller to save the context of tokenization, such as the position of the next token. This design allows strtok_r() to be safely used in concurrent environments and nested loops, unlike strtok(), which maintains internal state and is not thread-safe. In Pintos, where the kernel is responsible for parsing commands into executable names and arguments, the use of strtok_r() is particularly beneficial. It necessitates storing the address of the arguments' context in a user-defined location, enabling subsequent retrieval and processing, which aligns with Pintos's design of command handling.

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. By handling command parsing at the shell level, the time spent within the kernel is minimized, ensuring that kernel operations remain efficient and focused on core system tasks.
2. The shell can perform thorough validations, such as verifying the existence of executables before they are handed off to the kernel and checking if the argument count exceeds permissible limits, thereby preventing potential kernel errors.
3. The ability to parse commands allows the shell to perform sophisticated pre-processing, functioning as an interpreter in addition to being a user interface. This capability enables the execution of complex command sequences and operations, like running multiple commands simultaneously (e.g., cd; mkdir tmp; touch test;) or using pipelines.

SYSTEM CALLS (10 points)

The 10 total points are divided equally among the following sub-questions.

B1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration. Identify the purpose of each in 25 words or less.

Added member variables of struct thread in thread.h:

- uint32_t *pagedir; (line 102) - Manages the thread's virtual memory mappings.
- int exit_status; (line 105) - Stores the exit code when the thread terminates.
- struct file *fd[128]; (line 108) - Array holding references to open files.
- struct thread* p; (line 109) - Pointer to the parent thread.
- int load_failed_flag; (line 110) - Indicates if thread loading failed.
- struct semaphore load_completed_sema; (line 111) - Synchronizes the completion of thread loading.
- struct semaphore termination_signal_sema; (line 114) - Signals parent thread upon termination.
- struct semaphore cleanup_wait_sema; (line 115) - Ensures resources are cleaned up after termination.
- struct list child_threads_list; (line 116) - List of child threads.
- struct list_elem child_list_elem; (line 117) - Enables inclusion in the parent's child list.

Global variable in syscall.c:

- struct lock syscall_lock; (line 18) - Ensures mutual exclusion in system call execution to prevent data races and inconsistencies during concurrent access by multiple threads.

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

In Pintos, file descriptors are associated with open files within each process, and their uniqueness is maintained only within a single process. File descriptors are not unique across the entire OS, which helps in maintaining isolation and independence between processes.

B3. Describe your code for accessing user data in the kernel.

After validating and ensuring the safety of the operation, the kernel performs the actual data transfer through the addresses the user program conveyed.

- **Checking User Pointers**: The function *exit_if_invalid_ptr* is a safety check implemented before accessing user data. When handling the read system call, for example, the kernel needs to ensure that the buffer pointer provided by the user is valid.
- **Acquiring System Call Lock**: To manage access to shared resources like files, we uses a lock, ensuring that file operations are synchronized among threads.

B4. Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. Based on your code, 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?

- **Least Number of Inspections**: In Pintos, since user memory and kernel memory are distinct and switching between them requires changing the page directory, the least number of inspections theoretically could be 1 if the entire page is already verified to be valid and accessible. This scenario assumes a best-case where the page doesn't span multiple virtual pages or doesn't cross the user-kernel boundary (PHYS_BASE).
- **Greatest Number of Inspections**: In the context of Pintos and its approach to handling user pointers and memory layout, the theoretical maximum number of page table inspections for copying a full page of data (4,096 bytes) would be up to 4,096 if—and only if—every byte of data were to reside on a different page. This scenario, while possible in theory, is highly unlikely due to the contiguous allocation of memory pages in most operating systems. Therefore, a more realistic assessment of the greatest number of inspections required for copying a full page would be 2. This accounts for cases where the block of data spans across the boundary of two contiguous pages. Hence, the data being contiguous in memory means that, practically, the system would need to inspect the page table at most twice to ensure the entire page's data is valid and accessible for the operation.
- **For Copying 2 Bytes of Data:** Similarly, the least number of inspections is 1 if the 2 bytes are within a single page and that page is already validated or doesn't require repeated checks. The maximum would be 2 if each byte resides in a different page, necessitating a check for each byte.
- **Room for improvement**: Minimizing the number of inspections of the page table could benefit from hardware mechanisms that check page access rights more efficiently than software. On the software side, batching validation checks for user pointers can reduce the frequency of calls, enhancing performance.

B5. 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 sentences, describe the strategy or strategies you adopted for managing these issues. Give an example.

- **Pre-validation of User Pointers**: Implement a robust validation function (like *exit_if_invalid_ptr*) that checks if a user pointer is valid before any operation is attempted. This function should be called at the beginning of system calls or before any critical operations that access user memory.
- **Explicit Resource Release**: Explicit Resource Release: It is crucial to release resources, such as locks and files, before concluding a system call. A practical approach to guarantee this is to implement a dedicated function responsible for releasing resources. This function should be invoked at the end of each system call. Such a systematic method ensures that all resources are properly freed.

- **Example**: When a user program initiates a system call, the process involves three critical steps: 1) Perform sanity checks on the provided arguments, including validating addresses, 2) Execute the intended system call, and 3) Release any resources that were used. Adhering to this straightforward yet effective procedure ensures thorough error checking and the proper release of resources.

B6. Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

In Pintos, the "wait" system call synchronizes a parent process waiting for a child's termination, allowing the parent to retrieve the child's exit status efficiently. When a child is created, its PID is stored for the parent's reference, and a semaphore initialized to 0 manages synchronization. The parent waits on this semaphore, which the child signals upon termination, indicating it's safe for the parent to proceed and clean up.

B7. Consider parent process P with child process C. Explain how your code ensures synchronization and avoids race conditions in each of the following cases:

a) when P calls wait(C) before C exits?

b) when P calls wait(C) after C exits?

c) when P terminates without waiting before C exits?

d) when P terminates without waiting after C exits?

e) Are there any special cases?

To effectively manage these scenarios, Pintos leverages semaphores for synchronization and adopts strategies for avoiding race conditions.

a) The parent process, P, blocks on C's semaphore in the `wait` call. Since C has not exited yet, its semaphore value is 0, causing P to wait.

b) If C has already exited and signaled its semaphore before P calls `wait`, the semaphore count is greater than 0, allowing P to proceed without blocking. This ensures that C's exit status is preserved until P performs `wait`, thus ensuring correct retrieval of the exit status and preventing race conditions.

c) If P terminates before C without calling `wait`, P might either ignore C's exit status or handle it through another mechanism like a zombie process handler in the kernel. The OS must ensure that C's resources are eventually freed. Adopting C to another process like `init` could be a possible solution.

d) Similar to (c), if P terminates after C has exited but without waiting on C, the system needs to handle C's exit status and resource cleanup without P's direct involvement.

Special Cases

a) **Multiple Children**: If P has multiple child processes and terminates, the system must handle each child's synchronization and cleanup correctly, whether or not P waits on them.

b) **Nested Children**: For nested child processes (e.g., C spawns its own child, C'), the synchronization and cleanup mechanisms must recursively ensure all descendant processes are appropriately managed.

B8. 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"?

To ensure the *exec* system call waits until the new executable has loaded and to communicate the load's success or failure back to the calling thread in Pintos:

- **Synchronization**: A semaphore in the calling thread's structure is initialized to 0 before loading the executable. After initiating the load, the calling thread blocks by calling *sema_down* on this semaphore.
- **Load Status Communication**: The load success or failure is stored in a shared variable. Once the loading process completes, this variable is updated with the load's outcome, and *sema_up* is called on the semaphore to unblock the calling thread.
- **Return Value**: The calling thread checks the shared variable to determine the load's success or failure. It returns the new process's PID if successful or -1 if failed.

B9. What advantages or disadvantages can you see to your design for file descriptors?

- **Advantage**: Our file descriptor design goal is simple and clear. The straightforward mapping of file descriptors to file objects within a process's thread structure simplifies the management of open files. This clear association aids in readability and maintainability of the code. Besides, by maintaining a separate file descriptor table for each process, the design ensures isolation between processes. This prevents accidental or malicious interference among processes in terms of file operations.
- **Disadvantage**: As the number of open files increases, the linear search through the file descriptor table to find an unused entry or to lookup a file descriptor on each operation can become a bottleneck, affecting performance.

B10. The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

- **Flexibility in Process Identification**: By decoupling thread IDs (tid_t) from process IDs (pid_t), the system gains flexibility in managing processes and threads. This can be particularly advantageous in systems that support multi-threading within processes, allowing for a clearer distinction between process-level operations and thread-level operations.
- **Improved Cleanup and Termination**: Custom mappings can simplify the cleanup process when a process terminates, especially in systems where processes may have multiple threads. This can help ensure that all resources are properly reclaimed.

SURVEY QUESTIONS (optional) Answering these questions is optional, but it will help us improve the course in future semesters. 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 semester.

In your opinion, was this assignment, or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?

Your answer here.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Your answer here.

Is there some particular fact or hint we should give students in future semesters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

Your answer here.

Do you have any suggestions for the TAs to more effectively assist students, either for future semesters or the remaining projects?

Your answer here.

Any other comments?

Your answer here.