+-------------------------+
| **CSE 521** |
| **PROJECT 2: USER PROGRAMS** |
| **DESIGN DOCUMENT** |
+-------------------------+

Group
=====

Damodar Praharsh Medisetty <damodarp@buffalo.edu>
Sai Priyatham Kurma <skurma@buffalo.edu>
Shirisha Reddy Bandari <shirisha@buffalo.edu>

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.

Ans) No new declaration of struct, struct member, typedef, enumeration, global or static variable.

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?

Ans) Argument parsing is implemented using the tokenize function, which breaks down the input file_name into individual tokens. The tokens are then stored in the argv array, and the number of arguments (argc) is incremented accordingly.

The tokenize function takes the file_name string as input and tokenizes it using strtok_r, splitting it into individual tokens based on spaces. Each token is then stored in the argv array, and the argc counter is incremented for each token.

To ensure that the elements of argv[] are in the correct order, tokens are stored in argv[] as they are encountered while tokenizing the file_name. This ensures that the order of arguments is preserved

To avoid overflowing the stack page, memory is dynamically allocated for argv[] using malloc and realloc. This ensures that enough memory is allocated to hold all the arguments and their respective pointers.

The stack layout is carefully managed in the setup_stack function to ensure that arguments are pushed onto the stack in the correct order without overflowing the stack page. Memory is allocated for the stack using palloc_get_page, and a minimal stack is created by mapping a zeroed page at the top of user virtual memory. Arguments are then pushed onto the stack in reverse order (from last argument to first argument) to ensure that they are in the correct order when accessed by the program. Additionally, the stack pointer is aligned to multiples of 4 to ensure proper alignment for memory access.

RATIONALE
==========

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

Ans) Pintos implements strtok_r() over strtok() due to its ability to handle multiple threads safely. Unlike strtok(), which uses a static pointer for parsing, strtok_r() explicitly passes parsing state, making it more predictable and preventing conflicts between threads.

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.

Ans) In Unix-like systems, the shell manages command parsing, allowing users to customize behavior through scripting. This flexibility grants users control and interactivity, facilitating tasks with features like command history and tab completion. Maintenance is simplified as updates to parsing can occur at the shell level, without kernel modifications. By separating parsing from the kernel, system resources are utilized more efficiently, as the kernel can focus on core tasks like process management. This approach offers advantages over Pintos' kernel-centric model, enhancing user experience and system maintainability.

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.

Ans)

a) Variable

int64_t sleep_time_ms = 1000;

Declared a variable 'sleep_time_ms' and initialized it to 1000 to make the process sleep for 1000 milliseconds

b) Typedef declaration:

typedef int pid_t

Declared for the skeleton implementation of the exec system call.

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

Ans) File descriptors are related directly to open files, if the value of the file descriptor is 1, then we write to the console. The file descriptors are unique within a single process.

ALGORITHMS
==========

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

Ans) Reading system call is Phase 3, so did not implement it.

In the 'syscall_handler' function when the system call number matches SYS_WRITE, it extracts the parameters (fd, buffer, and size) from the stack and calls the sys_write function with these parameters. It then returns the value of sys_write and stores it in the eax register of the interrupt frame (f). This return value will be retrieved by the user program after the system call is completed.

For writing we implemented the system call sys_write, which allows writing data from the kernel to a specified file descriptor.

sys_write Function: This function takes three parameters: fd (file descriptor), buffer (pointer to the data to be written), and size (the size of the data to be written).

Before proceeding with the write operation, the function checks whether the memory pointed to by the buffer pointer is valid by calling the get_val_uaddr function. It checks both the starting address and the ending address of the buffer to ensure that the entire buffer is accessible in user space. If the memory is not valid, it calls sys_exit with an error code (-1) to terminate the process.

If the buffer memory is valid and fd is equal to 1 (indicating a write operation to the console), the function calls putbuf to write the buffer contents to the console. It then returns the size of the data that was written.

If fd is not equal to 1 (indicating an invalid file descriptor), the function returns -1 to indicate an error.

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?

Ans) You can call that system call to ensure it is on the allocated page. Given that the file size is 4096 bytes, data must fit in the most 2 pages. Thus, we need to call 2 times when the data is not aligned to the page or 1 time if we can get the beginning of the page from the address to ensure that the data are on valid pages. Even for the 2 bytes, it is the same as above, that is it would take 2 calls if the data is not aligned to the page or 1 time if we can get the beginning of the page from the address.

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

Ans) Phase 3 Question. Not implemented

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.

Ans) In our code, we check if the memory address pointed to by uaddr is above PHYS_BASE, which is the physical memory boundary. If so, it returns -1, indicating an error condition. It also checks if uaddr is NULL. If so, it also returns -1. It checks if the memory address pointed to by uaddr is valid in the current process's page directory. If not, it returns -1. For example, for the write system call, before writing the file, we check for all the above conditions. Only when the conditions satisfied, the file was written.

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"?

Ans) Phase 3 Question. Not implemented

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?

Ans) Phase 3 Question. Not implemented.

RATIONALE
==========

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

Ans) We chose the first approach, where we validate the user address first, and then access the user memory from the kernel. This is because this approach is straightforward, efficient, and simple. This way of approach, resulted in a cleaner code, as this checking can be provided as a helper function and can be used in the system call. Moreover, it helped us to achieve efficiency, as page faults are faster than the bound checks and page validation. Once, this validation is done, we get the data without copying it into the kernel space. In addition, we can write the complete data using “putbuf” to copy the entire buffer and write it to standard out.

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

Ans) Phase 3 Question. Not implemented.

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

Ans) Phase 3 Question. Not implemented.

SURVEY QUESTIONS
==========

1) **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?**

Ans) The assignment overall was a great experience. We loved working with user programs. It was a great practical learning experience.

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

Ans) Yes working on system calls and user programs was gave us a good knowledge about how OS is implemented.

3) **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?**

Ans) All hints were provided to us through TA's project class and Montana's recitation videos. The pintos documentation was also very helpful while building and implementing the code.

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

Ans) No suggestions. The professor and the TA's did a wonderful job assisting students and their doubts.

5) **Any other comments?**

Ans) The project overall was a great learning experience.

Contribution:

All the team members contributed equally towards the working of the project

Damodar Praharsh Medisetty: Implemented Argument Passing, System Calls
Sai Priyatham Kurma: Implemented Argument Passing, System Calls
Shirisha Reddy Bandari: Implemented Argument Passing, System Calls