+-------------------------+
| CSE 421/521 |
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+-------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Hariharan Sriram <hsriram@buffalo.edu>
Srihari Ponakala <sriharip@buffalo.edu>
Kritiman Manikonda <krithima@buffalo.edu>

---- PRELIMINARIES ----

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

>> Describe briefly which parts of the assignment were implemented by
>> each member of your team. If some team members contributed significantly
>> more or less than others (e.g. 2x), indicate that here.

Hariharan Sriram: 1x
Srihari Ponakala: 1x
Kritiman Manikonda: 1x

>> 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.

Answer:
(void **esp, const) setup_stack char **save_ptr, char *file_name):
Allocates and initializes a process's user stack, including alignment and parsing of arguments.

char **argv: Before being pushed to the stack, parsed command-line arguments are stored in the dynamically allocated array char **argv.

int argc: The number of command-line arguments that a process has parsed is stored in the integer int argc.

char *save_ptr: When argument parsing, the auxiliary pointer char *save_ptr is used to tokenize reentrant strings with strtok_r.

---- 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?
Answer:

With the help of the thread-safe `strtok_r()` function, the command line is parsed into individual arguments, dividing the input into tokens. The file's name is the first token, and the remaining tokens are arguments.
During execution, the right order in `argv[]` is ensured by pushing these arguments onto the user stack in reverse order. These arguments are stored in a manually generated `argv[]` array, and they are additionally pushed onto the stack together with the argument count (`argc`) and a null return address to mimic function call behavior.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?
Answer:
Strtok_r() is thread safe and strtok() is not thread safe. strtok_r() maintains a unique context, allowing many threads to tokenize different strings separately.

>> 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.

Answer:
The major advantages of using Unix approach for parsing:
1) The shell generally handles the command line parsing, which simplifies the usage of the kernel. This allows the kernel to only focus on execution and management while the shell takes care of splitting commands, parsing arguments etc.
2) The kernel maintains its simplicity by pushing parsing to the shell. Without affecting the functionality of the kernel, users can change or modify the shell.
3) It is user friendly syntax, complex syntaxes can be understood by the user while coding.

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."
Answer:
The following are the declarations in the project:
struct process_file:
Maps file descriptors to files and keeps track of open files for every process. This makes sure that each process manages its file descriptors independently.

struct child_process:
keeps track of data about child processes, such as their PID, load and exit statuses, and semaphores that allow parent and child processes to synchronize.

const int CLOSE_ALL:
The operation to close every open file descriptor for a process is represented by the const int CLOSE_ALL function.
const int ERROR:
In situations where a system call fails, the general error code const int ERROR is utilized.
const int NOT_LOADED, LOAD_SUCCESS, LOAD_FAIL:
During a process' loading, the variables const int NOT_LOADED, LOAD_SUCCESS, and LOAD_FAIL represent different states.

bool thread_alive:

When a process is ending, the bool thread_alive is used to synchronize whether a thread is active.

struct ipc_buffer:
A shared buffer used for communication between processes that holds: data[
IPC_BUFFER_SIZE]: the message buffer.
sema: The semaphore is used for synchronization.

>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?
Answer:
The integers that are assigned to open each file are called file descriptors.
In order to keep track of open files, every process has its own file_list. It is a list of struct process_file. The struct process_file joins the file descriptors to their associated struct file.These are generally stored in a process table. Every entry is a file descriptor that is connected to a file.
A process-specific counter t->fd, initialized at 2 (reserving 0 and 1 for STDIN and STDOUT), is used to allocate file descriptors in a sequential manner. In order to keep file descriptors different within the process and avoid interprocessor issues, each process independently keeps them.
To retrieve the file that corresponds to the specified fd, the file_list is iterated whenever a system call, such as a read or write, is made. For releasing related resources, the CLOSE_ALL constant can be used to close a single file or all of them at once.
Each file descriptor is unique within a process.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel.
Answer:
Reading process:
The user validates the exact memory location of the file. To make sure that the user buffer points to legitimate user memory, use validate_buffer.
convert_user_vaddr is used to translate the user's virtual address to a kernel address. It then copies the data from the user to the kernel buffer and reads the file. The data is then sent back to the user buffer.
Writing process:
The user buffer is validated to check the memory location.

The data is copied from the user buffer to the kernel and write operation takes place.
To make sure the user buffer's memory range is legitimate, validate it.
Data should be copied into a kernel buffer from the user buffer.
Write the data from the kernel buffer to the file by performing the write operation.
Error Handling:
When an invalid memory access occurs during these operations, the process is immediately terminated via terminate_process.

>> 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?
Answer:
Given a full page 4096 bytes, 4KB of data is to be copied. The least number of inspections will be 1 if the entire memory is in a single page. The greatest number of inspections will be 1024 times as if the 4KB is split across 1024 pages, then each page will be called which makes it 1024 times.
For the system that only copies 2 bytes of data. The Least number of inspections will be 1 and maximum will be 2 because if the 2 bytes is split across 2 pages.

Yes, there is room for improvement by validating the entire memory in parts instead of validating for each byte. This way, for a full page the number of inspections will be a maximum of 1 and for 2 byte copy it will remain as 1 or 2 as itself is an optimal solution.

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

When a child process receives a wait from the parent process, the parent process is blocked until the child process finishes executing thanks to a semaphore condition. The `struct child_process`'s `exit_sema` semaphore is used to complete it. When the child process completes its execution, it calls `sema_up` to let the parent process know it's ready to continue. By deleting the child process from its `child_list`, the parent process cleans up and retrieves the child's exit status. This method guarantees accurate tracking of several offspring processes, resource deallocation, and appropriate synchronization. Even if the child process ends abruptly, cleanup is handled smoothly by the parent.

>> 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.

Answer:

The best possibility to check and avoid the errors while running the code are:
To verify user memory, file descriptors, and inputs without interfering with system call logic, use additional methods like validate_buffer, validate_string, and is_valid_pointer.
handle_syscall_error is a centralized error handling tool that ensures stability by gracefully ending problematic processes.
To make sure resources are cleaned up, free open files and child process data by utilizing functions like process_close_file and remove_child_process.
To prevent resource use and unexpected behavior, terminate_process is used.

Eg: For the above given example if there is an error in wait() system call, there is a possibility that the user buffer is not pointing to the right address or the file descriptor does not exist.
The above strategies will be followed to resolve the error.
This answer will be updated once the project is completed.

---- 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"?

Answer:

The best ways to verify and prevent errors during code execution are to employ helper functions like validate_buffer, validate_string, and is_valid_pointer to verify inputs, file descriptors, and user memory without interfering with the main system call logic.
The handle_syscall_error function provides centralized error handling by ensuring that

problematic processes are gracefully terminated, preserving system stability.
Using processes like process_close_file and remove_child_process, which release open files and child process data and stop resource leaks, proper resource cleanup is guaranteed.
In order to prevent needless resource usage and undefinable behavior, the terminate_process function is also used to instantly end problematic processes.
The parent process in the exec system call waits for the child process to finish loading. After finishing, the child process signals the parent process by setting a flag to show success or failure. In order to verify correct synchronization and error handling during execution, the parent process then looks at this flag to see if the exec operation succeeded or failed.

>> 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?
Answer:
Scenarios:
1) P calls wait (C) before C exits:
Might use a semaphore to ensure P is blocked until C is completed.
2) P calls wait (C) after C exits:
Return C's exit status of weather success/failure.
3) P terminates without waiting, before C exits:
When C completes its execution and exits, it must clean its resources.
4) **P terminates without waiting, after C exits:**
The C's resources are emptied during the termination as the P process won't manage them.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?
Answer:
This architecture makes it easier to comprehend and maintain by separating memory validation from core logic. Helper methods like `is_valid_pointer` minimize the possibility of errors resulting from invalid memory access and centralize error handling. Debugging is made easier and improved security is guaranteed with this modular approach. Our version also has Inter-Process Communication (IPC) features, which allow processes to

effectively send and receive messages by utilizing a shared buffer with semaphores for synchronization. Additional features like `remove_child_process` and `process_close_file` for dynamic resource cleanup guarantee system stability and stop resource leaks. This makes our method more reliable.

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?
Answer:
Advantages:
As every process has its own database of file descriptors, our method provides the significant advantages of simplicity and isolation. By doing this, security is guaranteed and process conflicts can be avoided. Processes can open, use, and close files as needed without using up all of the resources available due to the implementation's ability to enable dynamic allocation and deallocation.
Disadvantages:
One possible disadvantage is that if too many files are opened at once or if files are not closed properly, programs could be running out of file descriptors. In addition, it takes more work to maintain stability when maintaining file descriptors across process boundaries (for example, for inherited or IPC file descriptors). In spite of these difficulties, our design is efficient for the majority of use cases because it finds a compromise between utility and simplicity.

>> B11: The default tid_t to pid_t mapping is the identity mapping.
>> If you changed it, what advantages are there to your approach?
Answer:
By connecting threads to their processes directly, the default `tid_t` to `pid_t` identity mapping simplifies process and thread management and makes debugging and tracking simple. It is practical and effective to use the identity mapping since altering it could allow for more flexible designs, such as shared process IDs for threads, but it also adds complexity and hazards.

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?
The project was moderate. It took us enough time as we had to implement different methods like ipc and the code is more robust as it handles errors.

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?
On working on system call, it helped us gain more knowledge on how files are called.

>> 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?
The project was fine, but it would be great if we had this also in phases.

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

>> Any other comments?