+-------------------------+
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+-------------------------+

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.

No changes made for argument passing.

---- 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?

We have begun by allocating and mapping a fresh, zeroed page at the top of user space (PHYS_BASE−PGSIZE) in setup_stack(). That page becomes our entire user stack.

First, we are copying the full command line (from process_execute()) into a temporary buffer and split it with strtok_r(), storing each token's pointer in a local argv[] array. Then, to lay out the strings in memory as C expects, we push each argument string onto the stack in reverse order, decrementing esp by the string's length +1, copying it there, and saving its final address.

Once all strings are on the stack, we word-align esp down to a multiple of four bytes, push a null sentinel (to terminate argv), and then push the saved addresses, again in reverse, so that the first argument's address ends up at argv[0]. Finally, we push the pointer to this argv[] array, the integer argc, and a dummy return address.

Because we are starting at PHYS_BASE and only subtract from esp within our one-page allocation, we can detect any underflow or allocation failure: if setup_stack() can't fit everything in one page, it returns false, load() aborts, and the process exits cleanly—no overflow, no leaks, and a correctly ordered argv[] for the user program.

---- RATIONALE ----

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

Pintos only provides strtok_r() because the classic strtok() relies on a hidden static state

and is not thread-safe, two properties an OS kernel really needs. By using strtok_r(buf, sep, &save_ptr), we are keeping all parsing context in our own variable, avoid any global or per-thread hidden buffers, and can safely tokenize multiple strings (or re-enter the parser from interrupt context) without overwriting. So, strtok_r() gives us the simplicity of strtok()'s API and the safety and ability to enter again required inside Pintos.

>> 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.

Simplicity and Safety in the Kernel:
By pushing all parsing into a user-level shell, the kernel stays small and avoids subtle bugs (and potential security holes) in argument-splitting logic. Any mistakes in quoting, escaping, or locale handling can be fixed in the shell without touching kernel code or risking a system-wide crash.

Rich, Customizable Command Syntax:
Parsers in userspace can offer powerful features like wildcards, pipes, redirection, aliases, job control, custom quoting rules without bloating the kernel. You can swap shells (bash, zsh, fish) to get totally different parsing behaviors, language extensions, or interactive conveniences, all without any kernel changes.

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.

```
/* Maximum number of open file descriptors per thread. */
#define FD_MAX 128
```

Inside threads/thread.h:

```
int exit_status;                /* Child's exit code for parent's process_wait. */
uint32_t *pagedir;              /* Base pointer to this process's page directory. */
struct file *executable_file;   /* File handle for the running executable (denies writes). */
struct hash supp_page_table;    /* Supplemental page table for virtual-memory
mappings. */
struct file *fd_table[FD_MAX];  /* Table of open file pointers indexed by file descriptor. */
int next_fd;                    /* Next available file descriptor number (starts at 2). */
struct thread *parent;          /* Pointer to this process's parent thread. */
struct list child_list;         /* List of this thread's child processes. */
struct list_elem child_elem;    /* Element linking this thread into its parent's child_list. */
struct semaphore wait_sema;     /* Parent downs this to wait for a child's exit. */
struct semaphore load_sema;     /* Child ups this to signal load() completion. */
bool load_success;              /* Child sets this to tell parent if load() succeeded. */
```

bool waited;   /* True if parent has already called wait() on this child. */
unsigned magic;   /* (Existing) Detects stack overflow. */

Inside threads/thread.c:
extern struct thread *get_thread_by_tid(tid_t tid);

Inside filesys/filesys.c:
extern struct lock filesys_lock;   /* Global lock serializing all file-system operations. */

>> 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, each process keeps its own small table an array of struct file * indexed by file descriptor numbers. When you call open(), the kernel grabs a free slot in that array (starting at 2, since 0 and 1 are stdin/stdout), stores the struct file * there, and returns that index as your file descriptor. All subsequent reads, writes, seeks, and closes use that index to look up the right struct file *.

Because each process has its own table, file descriptors are only guaranteed unique within a single process. Two different processes can both have descriptor 3 open—they just point into two different tables and might point to two different struct file *s, or even the same file opened twice.

---- ALGORITHMS ----

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

Pointer checks:

Every time we touch a user pointer (whether it's the syscall's esp, a buffer, or a C-string), we run it through validate_ptr() or validate_range().

These helpers confirm that the address isn't NULL, it lies below PHYS_BASE, and each page (or each character, for strings) is actually mapped in the process's page table. If any check fails, we instantly call sys_exit(-1), killing the misbehaving process before it can corrupt the kernel.

Reading (SYS_READ):

fd 0 (stdin): we loop calling input_getc() one byte at a time and stuff characters into the user buffer.

fd ≥ 2 (files): we look up cur->fd_table[fd]; if it's non-NULL we grab filesys_lock, call file_read() to pull in up to size bytes, then lock_release(). Otherwise we return -1.

Writing (SYS_WRITE):

fd 1 (stdout): we simply invoke putbuf(buffer, size), which pushes your bytes onto the console.

fd ≥ 2 (files): again we fetch the struct file*, lock filesys_lock, call file_write(), then unlock. If the descriptor is invalid we return -1.

>> 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?

Our validate_range(uaddr, size) walks from the page containing the first byte of the buffer up to the page containing the last byte, doing:

```
for (page = start & ~PGMASK; page <= end & ~PGMASK; page += PGSIZE)
  pagedir_get_page(cur->pagedir, page);
pagedir_get_page(cur->pagedir, end);
```

Here, the final check guarantees the very last byte is valid even if it lies in the same page as the loop's last iteration.

Full 4096 B copy:
Best case (page-aligned): loop touches exactly one page ⇒ 1 lookup, plus the extra "end" check on the same page -> 2 total
Worst case (straddles two pages): loop visits page A and B ⇒ 2 lookups, then final check on B -> 3 total

Tiny 2 B copy:

Within one page: same as above -> 2 lookups

Cross-page boundary: loop visits page A and B -> 2 lookups, plus final check on B -> 3 total

Optimized single-loop approach

```
start_page = start & ~PGMASK;

end_page = (start + size - 1) & ~PGMASK;

for (page = start_page; page <= end_page; page += PGSIZE)

        pagedir_get_page(cur->pagedir, page);
```

This logic do exactly one pagedir_get_page() per touched page:

4096 B ⇒ 1 lookup if aligned, 2 if wide apart

2 B ⇒ 1 lookup if contained, 2 if crossing

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

We are tracking each child in the parent via a small child_rec structure:

```
struct child_rec {
  tid_t tid;
  int exit_status;
  bool waited;
  struct semaphore dead;
  struct list_elem elem;
};
```

On the parent side, process_execute() calls thread_create() to start the child, then immediately allocates a child_rec (with its semaphore initialized to 0 and waited = false) and links it into the parent's child_list. Later, when the parent invokes process_wait (child_tid), it searches for that record, if it's missing or already marked waited, it returns -1 right away. Otherwise, it calls sema_down(&rec->dead), blocking until the child signals its exit. Once unblocked, the parent copies out rec->exit_status, sets rec->waited = true to prevent a second wait, removes and frees the record, and returns the saved status.

On the child side, process_exit() looks up its own child_rec in the parent's list just before tearing down its page tables. It writes its exit code into rec->exit_status and then calls sema_up(&rec->dead), instantly waking any parent blocked in process_wait(). After that, the child continues cleaning up its own resources and terminates.

By this way we can have a clean, race-free handoff of the exit code and by this each child can only be waited on once, and automatically reclaims the small per-child records , which avoids both lost wakeups and resource leaks.

>> 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.

Any time we touch user memory, whether fetching the syscall number, its arguments, or a whole data buffer, we risk a bad pointer that must immediately kill the user process. To keep our core logic uncluttered, we are front-loading all pointer checks into a handful of tiny validators:

- validate_ptr(uaddr) checks for NULL, user-space range, and a live page mapping.
- validate_range(uaddr, size) walks an address range page-by-page.
- validate_str(s) walks a NUL-terminated string.

Each validator calls sys_exit(-1) on failure, so the rest of the syscall body can simply assume its inputs are safe.

Once validation succeeds, we grab any needed locks (e.g. filesys_lock), do the real work, and release locks as soon as possible—never holding a lock while calling back into user code. For transient allocations (scratch pages, bounce buffers), we use a simple C “goto cleanup” pattern or paired allocate/free within the same block, which ensures that every exit path drops locks and free up pages.

Example (SYS_OPEN):

1. validate_str(path) kills the process if the path is bad.
2. lock_acquire(&filesys_lock)
3. file = filesys_open(path)
4. lock_release(&filesys_lock)
5. If the file is NULL, return -1; otherwise install it in fd_table and return the new descriptor.

By isolating pointer checks, lock management, and cleanup into helpers and cleanup labels, the body of each syscall stays laser-focused on what it’s doing, not on how to undo every possible error.

---- 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"?

When a thread calls exec (via process_execute), it immediately creates a child record in its own struct thread—including a load_sema (initialized to 0) and a load_success flag. After thread_create(...) returns the child’s TID, the parent does sema_down(&child->load_sema), sleeping until the child wakes it.

In the child’s start_process, right after load(...) finishes, the child sets thread_current()->load_success to true or false and then calls

sema_up(&thread_current()->load_sema). That unblocks the parent's sema_down. Back in the parent, we inspect child->load_success: if it's false, exec returns –1; otherwise, it returns the new TID. This simple semaphore handshake guarantees that exec never returns until the new executable has finished loading, and that success or failure is communicated cleanly back to the caller.

>> 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?

We associate each child with a small child_rec in the parent's thread: it contains the child's TID, an exit_status slot, a dead semaphore (initialized to 0), and a waited flag.

Waiting before or after child exit:

When P calls wait(C) before C exits, it finds the matching record and does sema_down(&rec->dead), blocking until C's process_exit() writes its exit code into rec->exit_status and does sema_up(&rec->dead). If P calls wait(C) after C has already exited, that earlier sema_up() left the semaphore count at 1, so sema_down() returns immediately. In both cases, P then reads rec->exit_status, sets rec->waited = true, removes and frees the record, and returns the status, which ensures exactly one handoff and no lost wake-ups.

Parent exits without waiting:

If P terminates without ever calling wait(), its own process_exit() walks its child_list. For each child_rec (whether the child is still running or already dead), it frees the record and, for surviving children, clears their parent pointer so they won't attempt to signal a gone parent. This avoids dangling semaphores or memory leaks.

Child exits after parent:

When C's process_exit() sees parent == NULL, it skips the semaphore handshake entirely and proceeds to clean up its own resources (page tables, thread struct, etc.).

Special cases:

- Calling wait() on a non-child or on a child already waited on returns –1 immediately.
- Exactly one exit status is delivered per child.
- All child_rec allocations and semaphores are freed in every ordering of parent/child exits.

By doing this we can have proper synchronization in all timing scenarios and can prevent race conditions and lost wake-ups. Also we can ensure no resource leaks regardless of whether parent or child whoever exits first.

---- RATIONALE ----

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

We chose to gate every kernel access to user memory through a small set of helper functions (validate_ptr, validate_range, validate_str) that call pagedir_get_page(). This guarantees that any bad pointer immediately triggers process termination, which prevents obscure null or unmapped dereferences in the kernel. By centralizing these checks, our syscall handlers stay focused on their core logic like open, read, write, etc. instead of getting stuck in repetitive error branches.

Reusing pagedir_get_page() means we lean on Pintos's existing page-directory interface, so there's no extra tracking or bookkeeping needed at this stage. When we later add a full VM subsystem, we can swap out those helpers for a more sophisticated faulting or pinning mechanism without touching every syscall. This design makes a balance between safety (no stray kernel derefs), clarity (clean syscall code), and future extensibility.

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

Advantages

Simplicity & speed: Constant-time lookup and teardown: fd_table[fd] gives you the struct file* immediately.

Process isolation: No global FD namespace—one process can't destroy another's descriptors.

Clear Unix-style API: Small integers match familiar POSIX semantics.

Disadvantages

Fixed limit: With FD_MAX=128, you can exhaust descriptors even if only a few are open at once.

No reuse: Our next_fd++ never reclaims closed slots, so a long-running process that opens/closes files can "leak" fds until it hits the cap.

Wasted space: Sparse tables waste memory when only a handful of fds are active.

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

In our design we left pid_t exactly equal to tid_t, so every process's user-visible PID is just its kernel thread ID. That choice kept things trivial, no extra lookup tables or translation logic, and zero runtime overhead when a syscall hands you a PID. Because Pintos treats each thread as a "process," there's no confusion between threads and processes under

the hood.

Had we separated them—say, by introducing a dedicated PID allocator and a PID->TID lookup table—we would gain a clean separation of “process identity” from “scheduling identity.” That would let us safely recycle thread IDs without worrying about dangling PIDs, hide internal kernel threads from user space, and manage PIDs (e.g. reuse or range-limit) independently. In our small-scale project, however, the simplicity of the identity mapping can outweight those potential benefits.