INF333 2024-2025 Spring Semester

Pintocchio
Ali Burak SARAÇ <21401932@ogr.gsu.edu.tr>
Sude Melis PİLAZ <22401992@ogr.gsu.edu.tr>

Homework II
Design Document

Please provide answers inline in a `quote` environment.

1 Preliminaries

Q1: If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.

> We created a dedicated process_block structure separated from thread structure. This approach helped us manage parent-child relationships more effectively and simplified synchronization between processes.

Q2: Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, and lecture notes.

> Dr. Ramesh Yarraballi's YouTube Channel
> Prof. Thierry Sans's YouTube Channel

2 Argument Passing

2.1 Data Structures

Q3: 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.

In process.c:

- `#define CMD_ARGS_MAX 32`
  Maximum number of command-line arguments allowed for a process.

2.2 Algorithms

Q4: 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?

Our argument parsing implementation is based on a two-phase approach. In `process_execute()`, we first extract only the program name (`argv[0]`) using `strtok_r()` so that the thread is created with the correct name. The full command line is then passed to `start_process()`.

In `load()`, we parse the complete command line using a helper function (`get_args()`) which tokenizes the command string into an array of arguments (`argv`) and returns the argument count (`argc`).

For stack arrangement, `setup_stack()` constructs the stack from high to low addresses as follows:

- Each argument string is pushed onto the stack in reverse order (`argc-1` to 0).
- The stack pointer is aligned to a 4-byte boundary using zero padding.
- A null pointer sentinel (`argv[argc]`) is pushed.
- Pointers to each argument string (`argv[0]` to `argv[argc-1]`) are pushed in reverse order.
- The address of `argv` itself is pushed.
- `argc` is pushed.
- Finally, a fake return address (0) is pushed.

To avoid stack overflow, we define `CMD_ARGS_MAX` (32) to limit the number of arguments.

2.3 Rationale

Q5: Why does Pintos implement `strtok_r()` but not `strtok()`?

Pintos uses `strtok_r()` instead of `strtok()` because `strtok_r()` is safe to use with many threads at the same time, but `strtok()` is not.

The reason is that `strtok()` saves its progress in a static variable, so if two threads use it at once, they interfere with each other. `strtok_r()` fixes this by letting each thread keep its own progress using an extra pointer.

This is important in Pintos because different threads might need to split up command lines at the same time. For example, we use `strtok_r()` in both `process_execute()` and `get_args()`, and these can run in different threads.

Q6: 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.

The Unix approach, where the shell separates the executable name and arguments before passing them to the kernel, has several advantages:

1. **Simpler kernel design:** By handling command parsing in the shell, the kernel remains less complex and more focused on its core responsibilities. This reduces the likelihood of bugs and makes the kernel easier to maintain.

2. **Greater flexibility for users:** The shell can provide advanced features such as wildcard expansion, variable substitution, and custom parsing rules. This allows users to have a more powerful and customizable command-line experience without requiring changes to the kernel.

3 System Calls

3.1 Data Structures

Q7: 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

- **In process.h:**
  - `struct file_descriptor`
    Manages file descriptors for open files:
    - `int file_id;` - Unique identifier for the file descriptor
    - `struct file *file;` - Pointer to the actual file structure

        - `struct list_elem elem;` - List element for open_files list in each thread
    - `struct process_block`
      To help a parent process track and synchronize with a child process — including whether the child has loaded successfully, exited, and what its exit status was.
        - `tid_t tid;` - Child's thread ID
        - `int exit_status;` - Exit status of the child
        - `bool is_exited;` - Whether the child has exited
        - `bool waited;` - Whether parent has waited on this child
        - `struct semaphore exit_sema;` - Synchronizes parent with child's exit
        - `struct semaphore load_sema;` - Synchronizes parent to child's loading
        - `bool load_status;` - Success/failure of child's loading
        - `struct list_elem elem;` - List element for child processes
- **In syscall.c:**
    - `static struct lock file_lock;`
      Global lock for synchronizing file system operations across multiple threads.
- **In thread.h:**
    - New members in `struct thread` under `#ifdef USERPROG`:
        - `struct list children;` - List of child processes
        - `struct process_block *cinfo;` - Process information structure
        - `int exit_code;` - Process exit status
        - `struct list open_files;` - List of open file descriptors
        - `int next_fd;` - Next available file descriptor number (starts from 2, reserving 0 and 1 for stdin and stdout)
        - `struct file *executable;` - Currently executing file

Q8: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

In our implementation, file descriptors are linked to open files using the `file_descriptor` structure, which holds both the file descriptor ID and a pointer to the file object. Each thread keeps its own list of open files (`open_files`) within its thread structure.

File descriptors are unique only within a single process, not across the entire operating system. Each thread has its own file descriptor numbering, starting from 0, 1 (for standard input, output), and increasing from there. The `next_fd` field in the thread structure keeps track of the next available descriptor number for that thread.

When a file is opened, the following steps occur:

3.1. A new `file_descriptor` structure is created.

3.2. The next available descriptor number from the thread's `next_fd` counter is assigned.

3.3. The pointer to the opened file is stored.

3.4. The structure is added to the thread's `open_files` list.

3.5. The thread's `next_fd` counter is incremented.

This per-thread approach allows different processes to use the same descriptor numbers for different files, ensuring isolation between threads and simplifying descriptor management.

Q9: Describe the sequence of events when `lock_release()` is called on a lock that a higher-priority thread is waiting for.

When `lock_release()` is called on a lock that a higher-priority thread is waiting for, the following sequence of events occurs:

1. The lock's `holder` field is set to `NULL`, indicating the lock is now available.

2. The system checks the lock's semaphore waiters list to see if any threads are waiting.

3. If waiters exist, the highest-priority waiting thread is unblocked by calling `sema_up()` on the lock's semaphore.

4. The unblocked higher-priority thread is added to the ready queue.

5. Since a higher-priority thread is now ready, the scheduler is invoked to determine if a context switch is needed.

6. The higher-priority thread acquires the CPU, runs `sema_down()` to completion, and sets itself as the new lock holder.

7. The original thread that released the lock may be preempted immediately if the newly unblocked thread has higher priority, implementing priority scheduling.

3.2 Algorithms

Q10: Describe your code for reading and writing user data from the kernel.

Our code for reading and writing user data from the kernel is primarily implemented in the `sys_io` function, which handles both read and write operations with a unified approach:

1. **Parameter handling**: The function takes a file descriptor (`fd`), a buffer pointer, a size, and a boolean flag indicating whether this is a write operation.

2. **Special case for console output**: If the file descriptor is 1 (stdout) and the operation is write, we directly call `putbuf()` to output to the console after acquiring the file system lock.

3. **File descriptor lookup**: For other file operations, we: - Get the current thread - Acquire the file system lock to ensure thread safety - Iterate through the thread's open files list to find the matching file descriptor - Release the lock after finding the file (or not finding it)

4. **Error handling**: If no matching file descriptor is found, we return -1 to indicate failure.

5. **Performing I/O**: After finding the file: - We reacquire the file system lock - Based on the `is_write` flag, we either: - Call `file_write()` to write data from user space to the file - Call `file_read()` to read data from the file into user space - Release the lock after the operation

6. **Memory safety**: Before accessing user memory, the `syscall_handler` validates all user-provided addresses using `check_user_address()`, which verifies both that the address is in user space and that it's properly mapped in the page directory.

This implementation ensures thread-safe file operations while properly handling the transition between kernel and user memory spaces.

Q11: 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?

When a system call copies a full page of data from user space into the kernel, the least number of inspections of the page table is 1, which occurs if the entire page is already in memory and mapped correctly, check_user_address() would only need to check the page table once.

The greatest number of inspections would be 4,096, where each byte of the page is inspected individually to check if it is present in memory.

This would be the case if the page table is not optimized and each byte is checked separately.

For a system call that only copies 2 bytes of data, the least number of inspections would be 1, if the page is already in memory and mapped correctly. The greatest number of inspections would be 2, if each byte is checked individually.

There is room for improvement in these numbers by optimizing the page table to allow for larger chunks of data to be checked at once, rather than inspecting 4 bytes at a time. This could be done by using a more efficient data structure for the page table, or by implementing a caching mechanism to store the results of previous inspections.

Q12: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

Our implementation of the "wait" system call is centered around the `process_wait()` function, which interacts with process termination through a synchronization mechanism:

1. When a parent process calls `wait(child_tid)`, it searches its list of children for the specified child process ID.

2. If the child is found and hasn't been waited on before, the parent marks it as "waited" to prevent multiple waits on the same child.

3. If the child hasn't exited yet, the parent blocks by calling `sema_down()` on the child's exit semaphore (`exit_sema`), which was initialized when the child was created.

4. When the child process terminates in `process_exit()`, it: - Sets its exit status in the `process_block` structure - Marks itself as exited by setting `is_exited` to true - Calls `sema_up()` on its exit semaphore to unblock the waiting parent

5. After the parent is unblocked (either immediately if the child already exited, or after the child exits), it: - Retrieves the child's exit status - Removes the child's `process_block` from its children list - Frees the memory allocated for the child's process information - Returns the exit status to the caller

This implementation ensures proper synchronization between parent and child processes, allowing the parent to retrieve the child's exit status while preventing resource leaks.

Q13: 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.

Our approach to handling user memory access errors focuses on early validation and clean termination: We implemented a `check_user_address()` function that validates any user-provided address before it's accessed. This function verifies both that the address is in user space and that it's properly mapped in the page directory. When invalid memory is detected, we immediately terminate the process with exit code -1. We validate all user addresses at the beginning of system calls before acquiring locks or allocating resources. For example, in the `SYS_WRITE` handler, we check the buffer address before attempting any file operations.
The `process_exit()` function handles cleanup of all resources (open files, memory, etc.) when a process terminates, ensuring no leaks occur even after abnormal termination. For example, in our `sys_io()` function, we validate the buffer address before acquiring any locks. If validation fails, the process is terminated immediately. If validation passes but the file descriptor is invalid, we return -1 without touching any file resources. The file lock is only acquired when we're certain we have a valid file to operate on, and it's released immediately after the operation completes, ensuring it's never left in a locked state.

3.3 Synchronization

Q14: 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"?

> Our implementation ensures that the "exec" system call doesn't return before the new executable has completed loading through a synchronization mechanism using semaphores:
>
> 1. When a parent process calls `exec`, it invokes `process_execute()`, which creates a new thread for the child process.
>
> 2. During child thread creation, we initialize a `load_sema` semaphore in the child's `process_block` structure with a value of 0.
>
> 3. The parent thread then blocks by calling `sema_down()` on this semaphore, ensuring it won't continue until the child signals it.
>
> 4. In the child's thread function (`start_process`), it attempts to load the executable. After loading succeeds or fails, it: - Sets the `load_status` field in its `process_block` to indicate success or failure - Calls `sema_up()` on the `load_sema` to unblock the parent
>
> 5. When the parent wakes up, it checks the `load_status` field: - If loading succeeded, it returns the child's thread ID - If loading failed, it returns -1
>
> This approach ensures that the parent process waits for the child to complete loading before returning from the exec system call, while also providing a mechanism to pass the load success/failure status back to the parent thread.

Q15: 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?

> Our implementation ensures proper synchronization between parent and child processes through a combination of semaphores and status tracking:
>
> **When P calls wait(C) before C exits:** P searches its children list for C's process block If found, P checks if C has already been waited on (preventing multiple waits) If C hasn't exited yet, P blocks on C's exit semaphore (`exit_sema`) When C eventually exits, it signals this semaphore, unblocking P P then retrieves C's exit status and frees C's process block
>
> **When P calls wait(C) after C exits:** P finds C's process block in its children list Since C has already set `is_exited` to true, P doesn't need to block P directly retrieves C's saved exit status P removes C's process block from its children list and frees it

When P terminates without waiting, before C exits: P's process resources are freed, but C continues execution independently C's process block remains in P's children list (which is freed when P exits) When C eventually exits, it still updates its exit status and signals its exit semaphore Since P is no longer waiting, the signal has no effect, but no resources are leaked

When P terminates without waiting, after C exits: C has already updated its exit status and marked itself as exited P's termination frees all its resources, including C's process block No synchronization is needed since C has already completed

Special cases: If P exits while C is running, C becomes an orphan but continues execution P maintains separate process blocks for each child, allowing independent waiting If P attempts to wait on a non-existent child, it returns -1 immediately

3.4 Rationale

Q16: Why did you choose to implement access to user memory from the kernel in the way that you did?

We chose to implement access to user memory from the kernel using a validation-first approach with the `check_user_address()` function.

First of all, our implementation directly validates addresses before accessing them, making the code flow easy to understand and maintain. The validation function checks that the address is in user space (`is_user_vaddr()`) and that it's properly mapped in the page directory (`pagedir_get_page()`). By validating addresses at the beginning of system calls, we can catch invalid memory accesses before performing any operations. When an invalid address is detected, we immediately terminate the process with a -1 exit code. This approach avoids complex error propagation and ensures that kernel integrity is maintained. While our implementation checks addresses in 4-byte chunks, which is sufficient for most system call arguments, we recognize that for large buffers this approach could be optimized. However, we prioritized correctness and simplicity in our implementation.

This approach balances simplicity and correctness, though at the cost of some performance optimization opportunities.

Q17: What advantages or disadvantages can you see to your design for file descriptors?

Our file descriptor design uses a list of file descriptor structures unique for each thread, containing a unique ID and a pointer to the actual file. This approach offers several advantages and disadvantages:

Advantages: Each process maintains its own file descriptor table, preventing one process from accessing another's files directly, enhancing security. The `next_fd` counter in each thread provides a straightforward way to assign unique IDs without complex management. The list-based approach allows for relatively quick lookups when a process has a moderate number of open files. When a process terminates, its file descriptor list makes it easy to identify and close all files it had open. The design easily accommodates special file descriptors (like stdin/stdout) alongside regular files.

Disadvantages: Finding a file descriptor requires iterating through the list, which becomes inefficient for processes with many open files. A hash table or array-based approach might be faster. Our implementation uses a global file system lock, which can become a bottleneck when multiple processes perform file operations concurrently. Each file descriptor requires a separate allocation, which could lead to memory fragmentation with many small allocations. We don't implement per-process limits on the number of open files, which could lead to resource exhaustion. If a process doesn't explicitly close its files, they remain open until process termination, potentially wasting system resources.

Q18: The default `tid_t` to `pid_t` mapping is the identity mapping. If you changed it, what advantages are there to your approach?

We kept the default identity mapping between `tid_t` and `pid_t`, so each thread ID is used directly as the process ID. If we had changed this mapping, this could result in allowing multiple threads to share the same process ID, implementing multi-threaded user processes. However, for our project, the identity mapping has been kept simple and effective.

4 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.

Q1: 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?

> We sincerely believe that the assignment was very challenging. The problems were complex and required a deep understanding of the Pintos operating system. The assignment took a lot of time to complete, and many problems we encountered were because of our lack of experience with Pintos. The main problem for this project was the lack of guidance and examples. Not only the assignment but also the Pintos documentation was not very helpful. We believe that the assignment would be more manageable if there were more examples and guidance on how to approach the problems.

Q2: Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Q3: 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?

Q4: Do you have any suggestions for us to more effectively assist students, either for future semesters or the remaining projects?

Please provide more examples and guidance on how to approach the problems. Sparing some time to talk, debate on the project in the classroom could be an effective way to help students understand the concepts better, making the lessons more interactive. Having short conversations about the common problems, key aspects of the project before or after the lessons could prove to be very beneficial. This is a popular method in many universities and it is proven to be very effective.

Sparing some time to talk, debate on the project in the classroom could be an effective way to help students understand the concepts better, making the lessons more interactive. Having short conversations about the common problems, key aspects of the project before or after the lessons could prove to be very beneficial. This is a popular method in many universities and it is proven to be very effective.

Q5: Any other comments?

Thank you for taking an interest in our feedback. We believe that the course could be steered in a more beneficial direction if the suggestions we provided above were taken into consideration. If the difficulty of the assignments could be balanced, we believe we can learn many things from this course.