# INF333 2023-2024 Spring Semester

***Sefiller***
*Onur Alp Gündüz 21401958@ogr.gsu.edu.tr*
*Kerim Ayberk Çıtak 20401870@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.

> No preliminary comments or notes for the TAs.

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

- Stack Overflow: How to solve Pintos Project2 userprogram intrcontext error
- Stack Overflow: Pintos userprog - all tests fail, is kernel vaddr
- YouTube: Pintos Tutorials by Prof. Pramod Subramanyan
- Stanford University: Pintos Project 3

## 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.

> `No new structs or variables are declared in the project.`
> `Currently, existing ones are used.`

### 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?

> Argument parsing is implemented by extracting arguments from the stack frame of the system call handler. The elements of `argv[]` are arranged by parsing the user-supplied command string and tokenizing it using `strtok_r()`. To avoid overflowing the stack page, a buffer is used to temporarily store each argument, and dynamic memory allocation is employed if needed.

### 2.3 Rationale

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

> Pintos implements strtok_r() instead of strtok() due to thread safety concerns. strtok() relies on a static variable to track parsing location, making it vulnerable to race conditions in a multithreaded environment. strtok_r() resolves this by allowing each thread to maintain its own parsing state, ensuring safe string tokenization.

**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.

> Two advantages of the Unix approach are improved flexibility in command execution and easier integration with shell scripting.

## 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

```
thread.h
struct child
  {
    tid_t tid;
    bool isrun;
    struct list_elem child_elem;
    struct semaphore sema;
    int store_exit;
  };

  struct thread_file
  {
    int fd;
    struct file* file;
    struct list_elem file_elem;
  };
   struct list childs;
    struct child * thread_child;
    int st_exit;
    struct semaphore sema;
    bool success;
    struct thread* parent;

    /* Structure for Task3 */
    struct list files;
    int file_fd;
    struct file * file_owned;
```

**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 has a one-to-one mapping to each file opened through syscall. The file descriptor is unique within the entire OS. We decided to maintain a list(struct list files) inside kernel, since every file access will go through kernel.

**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 a higher-priority thread awaits, the current thread relinquishes the lock, unblocks higher-priority waiting threads, and adjusts its priority if necessary based on the highest-priority waiting thread. If a higher-priority thread is present, a context switch occurs to let it run. This sequence ensures efficient priority management and responsiveness in the system.

## 3.2 Algorithms

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

The sys_write and sys_read functions are used to read and write user data from the kernel. The sys_write function is responsible for writing data from the kernel to either a file or the console. It first retrieves the user-space arguments from the interrupt frame and then performs pointer validation for security. If the target of writing is the console (STDOUT_FILENO), it writes the data to the console using putbuf. Otherwise, in the case of writing to a file, it retrieves the corresponding file descriptor from the thread's file list, acquires the file system lock, and writes the data to the file using file_write. If the file descriptor is invalid, it returns 0. The sys_read function follows similar steps to read data from the user. It finds the corresponding file descriptor and reads the data from the file using file_read. If an invalid file descriptor or pointer is detected, it returns -1. These functions ensure secure and controlled access to user-space data, perform necessary validation checks, and utilize the appropriate file system for reading and writing operations.

**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?

For a system call transferring a full page (4,096 bytes), the least and greatest possible page table inspections are one. The same holds for a call transferring only 2 bytes. The current implementation efficiently checks the entire page's validity in one pagedir_get_page() call. Further optimization could include techniques like demand paging for larger transfers, potentially enhancing efficiency and performance.

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

We ensure the validity of the given thread ID by iterating through the list of `child` structs associated with the current thread. If the thread ID is not found, it implies that either no thread with that ID exists or that the corresponding `child` struct has already been removed, possibly due to a prior call to `wait`. In such cases, the function returns -1. However, if the child with the specified ID is found, the `child` struct includes a boolean flag, `isrun`, indicating whether the child's thread has successfully executed. While the

child remains alive, the parent waits for its termination using a semaphore specifically designed for this purpose.

The "wait" system call serves the purpose of waiting for a child process to terminate. Upon termination, the child's exit status becomes available to the parent process, which can retrieve it using the "wait" system call. This mechanism allows for synchronous coordination between parent and child processes, enabling the parent to proceed only after the child has completed its execution.

**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.

To ensure robust error handling in system calls without cluttering the code, we adopt a straightforward strategy. First, we thoroughly validate user-specified addresses and memory accesses to prevent bad pointer values. For instance, in our "write" system call, we validate user pointers before accessing memory, ensuring they are within valid user address space.
Secondly, upon detecting a bad pointer or memory access failure, we promptly trigger an error-handling routine, such as process termination. This ensures that errors are addressed without complicating the primary code logic.
For resource cleanup upon error detection, we utilize a resource management approach like RAII. For instance, in the "write" system call, if a bad pointer is detected, we release any temporary resources like locks or buffers before terminating the process.
By encapsulating error-handling logic into reusable functions and macros, we maintain code readability and consistency across the codebase. This approach strikes a balance between robust error handling and maintaining code clarity.
For example, consider the scenario where a bad pointer is detected while accessing user memory in the "write" system call. We immediately invoke the error-handling routine, release any allocated resources, and terminate the process, ensuring proper cleanup and preventing further execution with invalid data.

## 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"?

> In our "exec" system call implementation, we ensure synchronous execution by calling the process_execute() function, which initiates the loading of the new executable. Before proceeding with the execution, the check_ptr2() function is invoked to validate the user-supplied pointer, ensuring it points to a valid user address and page in memory. This validation prevents erroneous memory access and helps maintain system integrity. If any issue is detected during this validation process, such as an invalid address or inaccessible page, the exit_special() function is called to terminate the process safely. Thus, by validating the pointer before execution and handling potential errors appropriately, our code ensures the reliability and security of the "exec" system call.

**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?

> To ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits, we utilize semaphores associated with the child process. When C exits before P calls wait(C), it signals the semaphore indicating its completion. Then, when P calls wait(C), it waits until the child process finishes execution. After C exits, regardless of P calling wait(C) or not, all resources associated with C are cleaned up, including releasing locks, freeing memory, and closing open files.
> If P terminates without waiting for C to exit, resources associated with C are still cleaned up. This cleanup can occur during P's exit process or be handled by the operating system upon detecting P's termination.
> Special cases may arise if there are multiple child processes or if P terminates abruptly without waiting for any child processes. In such scenarios, proper cleanup and synchronization mechanisms are necessary to handle these cases gracefully.

## 3.4 Rationale

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

In our implementation, we handle accessing user memory from the kernel through a function called check_ptr2(). This function serves as a gatekeeper, ensuring that any attempt to access user memory is safe and valid. First, it checks if the provided virtual address falls within the user space boundary. If it doesn't, the process is immediately terminated to prevent any unauthorized access.

Next, check_ptr2() verifies whether the page containing the given address is valid and mapped to physical memory. This step ensures that the memory access won't lead to segmentation faults or other memory-related issues. Finally, the function checks the content of the page by attempting to read four consecutive bytes. If any of these checks fail, indicating a potential memory access violation, the process is terminated.

We've opted for this approach because it offers comprehensive validation of user memory access. By checking both the address boundary and the page mapping and content, we ensure the integrity and security of system calls involving user memory. Additionally, we use the get_user function to safely read a byte from the user address, further enhancing the safety of our implementation.

Overall, our approach prioritizes robustness and security, minimizing the risk of security vulnerabilities and system crashes associated with accessing user memory from the kernel.

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

Advantages:

Uniform Structure: Our design for file descriptors allows for a unified approach, regardless of whether the descriptors originate from pipes or file openings. This uniformity simplifies the handling of file-related operations, enhancing code clarity and maintainability.

Scalability: Each thread maintains its list of file descriptors, offering a scalable solution with no inherent limit on the number of open file descriptors. This design allows processes to efficiently manage a large number of open files without hitting predefined limits, except for memory constraints.

Disadvantages:

Duplicate Structs: One drawback of our implementation is the presence of redundant file descriptor structures for standard input (stdin) and standard output (stdout) in each thread. This redundancy can lead to increased memory consumption and potentially complicates certain operations.

Linear Access Time: Accessing a file descriptor involves iterating through the entire list of file descriptors associated with the current thread. As a result, the access time is proportional to the number of file descriptors (O(n)), leading to potential performance issues, especially when dealing with a large number of open files. This could be mitigated by storing file descriptors in an array, which would allow for constant-time access (O(1)).

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

One advantage is simplicity. By not introducing additional mappings between thread IDs (tid_t) and process IDs (pid_t), the codebase remains more straightforward and easier to understand. This simplicity can reduce the complexity of the system and make it easier to maintain and debug.
Another advantage is avoiding redundancy. In systems where threads and processes are tightly coupled, such as many modern operating systems, having separate identifiers for threads and processes might not provide significant benefits. Using only thread IDs (tid_t) to identify both threads and processes simplifies the system's structure and eliminates the need for maintaining multiple mappings.
Additionally, not implementing a separate mapping from tid_t to pid_t can save memory and computational resources. Each additional mapping consumes memory and requires additional processing overhead for maintenance and lookup. By avoiding unnecessary mappings, system resources can be conserved and performance can be improved.
Overall, the decision not to implement a mapping from tid_t to pid_t can contribute to a leaner, more efficient system design, particularly in cases where the distinction between threads and processes is less critical or where simplicity and resource efficiency are prioritized.

## 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?

The assignment was quite challenging and took a lot of time to complete.

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

> We have definitely learned to appreciate the complexity and uniqueness of problems associated with OS code.

**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?

> Providing additional examples or case studies related to system call implementation could be helpful for students to better understand the concepts.

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

> Providing more frequent feedback and guidance throughout the assignment could help students stay on track and address any misunderstandings or difficulties early on.

**Q5:** Any other comments?

> Split this project into 3 projects. 1: argument passing. 2: wait, write, read, open. 3: fork. distribute the other calls among 2 and 3 in some logical way.