CS2043

LAB 2: USER PROGRAMS

DESIGN DOCUMENT

220309D

Karunaratne T.K.M.C

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> 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.

In **process.c**, there are changes to the function declarations for handling argument passing. The function **static bool setup_stack(void **esp, char **saveptr, const char *filename)** was updated to manage the command-line arguments more effectively by preparing the stack for the new process. The **bool load(const char *file_name, void (**eip)(void), void **esp, char **saveptr)** function was also modified to load the executable and initialize the process's instruction pointer and stack, ensuring that command-line arguments are correctly set up. These changes improve how arguments are parsed and passed to the new process.

---- 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?

Argument parsing is implemented in the setup_stack function. The algorithm to handle the parsing of arguments is as follows:

1. **Create Local Variables**: Initialize two local variables, **char **cont** and **char **argv**, with a default size of 2.
2. **Parse Command Line**: Use **strtok_r** to parse the command line starting from a saved pointer. Maintain a pointer to each argument in **char **cont** during the parsing process. If the number of arguments exceeds the current allocation, double the size of both **cont** and **argv**.
3. **Copy in Reverse Order**: Once the command line is fully parsed, copy **cont** to **argv** in reverse order. This reverse order is necessary because arguments must be pushed onto the stack from last to first. During this copying process, each character string is also pushed onto the stack.
4. **Align Stack Pointer**: After the copying process, ensure that the **argv** contains all arguments in reverse order. Then, align the stack pointer to a multiple of 4 bytes for proper word alignment.
5. **Push Pointers onto Stack**: Push the array of pointers from **argv** onto the stack.
6. **Push Additional Parameters**: Finally, push **argv, argc**, and a fake return address onto the stack.

---- RATIONALE ----

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

Pintos uses strtok_r() instead of strtok() because strtok() is not safe when used by multiple threads at the same time.

strtok() remembers where it left off in a static (shared) memory location. If two threads use strtok() at once, they'll overwrite each other's progress, causing mix-ups. strtok_r() is safer because it uses a separate pointer for each call, so each thread can keep track of its own progress without interfering with others.

>> 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.

1. Users can easily change or combine commands without needing to modify the actual programs
2. By keeping command parsing separate from the executable, each program can focus on doing one thing well. This makes the programs easier to write, understand, and maintain

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.

Added the following to struct thread:

struct list file_list;

- This list keeps track of all files that the thread (or process) currently has open. It allows the OS to manage open files and close them when necessary

int fd;

- This is the file descriptor, a unique identifier for each file opened by the process. File descriptors are typically incremented as each new file is opened.

struct list child_list;

- This list keeps track of the child processes spawned by the current thread or process. It enables the OS to manage child processes

tid_t parent;

- This holds the thread ID (or process ID) of the parent process. It helps establish a parent-child relationship, allowing a process to be aware of its parent, which is useful for inter-process communication or synchronization.

struct child_process* cp;

- This is a pointer to the child process structure associated with the current thread. It allows efficient access to information about the child process

struct file* executable;

- This pointer holds the file structure of the executable file for the process. It is used to deny write operations to the executable file while the process is running, ensuring the file remains unmodified during execution.

struct list lock_list;

- This list keeps track of all locks that the thread currently holds. This can be used to release all locks if the thread exits unexpectedly and helps with lock management to prevent deadlock or resource contention.

Added the following struct in syscall.h:

```
struct child_process {

  int pid;

  int load_status;

  int wait;

  int exit;

  int status;

  struct semaphore load_sema;

  struct semaphore exit_sema;

  struct list_elem elem;

};
```

- The struct child_process is meant to be a child process which will hold important information such as the pid, load status and semaphores.

```
struct process_file {

    struct file *file;

    int fd;

    struct list_elem elem;
```

};

- The process_file holds the file you're currently working with. It contains the file descriptor and the content of the file.

struct lock file_system_lock;

- Use for locking critical section that involves modifying files.

>> B2: Describe how file descriptors are associated with open files.

>> Are file descriptors unique within the entire OS or just within a

>> single process?

- Each process has a list of its open files (tracked by file_list in struct thread), and each file opened by a process is given a unique file descriptor within that process. This file_list contains entries of struct process_file, which holds both the file pointer and the associated FD. When a process opens a new file, a new struct process_file is created, and an unused FD is assigned The OS only guarantees that file descriptors are unique within each process. This approach allows different processes to use the same FD numbers for different files without interference, as each process manages its own file descriptor space independently.

- In Pintos, file descriptors are associated with open files in a way that they are unique only within each individual process and not across the entire OS.

---- ALGORITHMS ----

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

>> kernel.

1. <u>User program calls the system call</u>

   A user program starts input/output operations by calling **syscall_write** or **syscall_read**. These functions take parameters like the file descriptor (**fd**), a pointer to the data buffer, and the size of the data. For example, **syscall_write(fd, buffer, size)** lets the kernel know what to write, while **syscall_read(fd, buffer, size)** indicates what to read.

2. <u>Switch to kerenel mode</u>

When a system call is made, the operating system switches from user mode to kernel mode. This switch allows the kernel to perform actions that the user program cannot. The transition happens through an interrupt or trap, which saves the user program's state and loads the kernel state, enabling the execution of the system call.

3. Validation

In kernel mode, the system checks if the parameters are safe and valid. For **syscall_write** and **syscall_read**, the kernel first verifies that the file descriptor (**fd**) is within range (e.g., **fd < 0 || fd >= MAX_FD**). It also checks if the buffer address is valid using **is_user_vaddr(buffer)**. These checks prevent invalid memory access and ensure the system runs safely.

4. Execution of helper function

After validation, the kernel runs helper functions to carry out the operations. For writing, it calls **write_to_fd(fd, buffer, size)**, which moves data from the user buffer to the file. For reading, **read_from_fd(fd, buffer, size)** is used to transfer data from the file back to the user buffer. These functions handle the details of reading and writing without burdening the user program.

5. Return to user mode

Once the operation is done, the kernel prepares to return control to the user program. It sends back the result, like how many bytes were read or written

>> 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?

Least Number of Inspections:

When copying a full page (4,096 bytes), if the system call copies the data in one go, the least number of inspections would be **1 inspection**. This would happen if the entire page is mapped in a single call to pagedir_get_page() to verify the address once before the copy operation.

Greatest Number of Inspections:

The greatest number of inspections could occur if the copy is done byte-by-byte or in small chunks, causing pagedir_get_page() to be called for each byte (or each chunk) being copied. In this case, for 4,096 bytes, there could be up to **4,096 inspections** (one for each byte) if it checks the page table for every single byte copied.

If system call only copy 2 bytes of dats :-

Least Number of Inspections:

- For copying only 2 bytes, the least number of inspections would typically be **1 inspectio**n if the data is copied in a single operation after confirming the validity of the address.

Greatest Number of Inspections:

- The greatest number of inspections in this case would be **2 inspections** if each byte is inspected separately before copying them.

Room for improvement in these numbers

- ❖ Yes there is a room for improvement in these numbers

  The least inspections could remain at **1**, while the greatest could be improved from **4,096 inspections** to just **1** or **2.**

>> B5: Briefly describe your implementation of the "wait" system call

>> and how it interacts with process termination.

In my implementation of the "wait" system call, found in `syscall.c`, the function `syscall_wait` is invoked when a user program requests to wait for a child process to terminate. This function checks the validity of the provided process ID and verifies that the process exists.

Once validated, it calls the `process_wait` function from `process.c`, which puts the calling process into a waiting state until the specified child process finishes execution. The `process_wait` function manages the termination of the child process by ensuring that it correctly cleans up resources, such as memory and file descriptors, and retrieves the exit status. This interaction ensures that the parent process can obtain the exit status of the child while preventing resource leaks, facilitating proper process termination and synchronization in the system.

>> 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.

In my implementation of system calls in **syscall.c**, I focused on handling errors that might occur when accessing user memory. Before using any memory address provided by the user, I check if the address is valid using functions like **is_user_vaddr** and **validate_user_pointer**. These checks make sure that the address is safe to access and prevents the program from crashing if a bad pointer is used.

For each system call, such as **syscall_write** or **syscall_read**, I first verify that the inputs, like the file descriptor and memory address, are correct. If any of these checks fail, I return an error code immediately, so the system call doesn't try to perform an action that could cause problems. This keeps the main code clean and focused.

I also created a system for managing errors. If something goes wrong, I make sure to release any resources I was using, such as locks or temporary buffers. For example, if there's an error while reading data, I unlock any resources before reporting the error. This helps avoid memory leaks and keeps the system running smoothly.

To make error handling consistent, I return specific error codes for different types of problems. This way, when a user program encounters an error, it can understand what went wrong without getting lost in complicated error-handling code. By having clear error codes, developers can easily figure out how to fix issues.

---- 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"?

The exec system call in my implementation ensures that it does not return until the new executable has completed loading

The exec function calls a loading function that attempts to load the specified executable file into memory. This function will perform checks to ensure that the executable is valid and that it can be loaded successfully.

After the loading process, the loading function will return a success or failure status. If the loading fails, it returns -1, indicating an error.

In **syscall.c**, after calling the loading function, the exec implementation checks the return value. If the load was successful, the thread continues execution; if it failed, the exec function will return -1 to the calling thread.

The implementation ensures that the calling thread waits for the load operation to complete. This is typically achieved by having the loading function run in the context of the calling thread, ensuring that the exec system call does not return until after the loading process is complete.

Like this, the exec call provides the correct status back to the thread that invoked it, allowing for proper error handling in the event that the executable cannot be loaded.

>> 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?

Before C Exits:

To avoid problems when the parent process (P) calls wait(C), we can use a mechanism like a semaphore or condition variable. This will make P wait until the child process (C) finishes executing. Additionally, we can use a status flag in C to let P know when it is safe to proceed. This way, P will only continue after C has completed, preventing any potential issues.

After C Exits:

Once C has finished executing, it should notify P, which can be done through a signal or by setting an exit flag. This allows P to wake up from its waiting state and move on to clean up any resources used by C, such as memory or other data structures. By ensuring that P receives the notification, we can safely manage the resources associated with C without any conflicts.

When P Terminates Without Waiting....

before C Exits:

If P ends before C, C will become a "zombie" process, which means it still exists in the system but is no longer associated with its parent. The operating system keeps track of C's exit status until P can read it. To handle this situation, the system can reassign C to the init process (PID 1), which will eventually call wait to clean up C’s resources properly.

After C Exits:

If P exits after C has finished, the init process takes over the responsibility of cleaning up C. It will manage C’s exit status and free any resources that C was using. This way, the operating system ensures that all processes are cleaned up correctly, even if the original parent process is no longer around.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

For access to user memory from the kernel, we chose to use function

decomposition for error catching. This reduces the difficulty and is

more simple than to implement page fault memory handling.

Whenever a pointer is invalid, it will be caught by the page fault interrupt

handler. In the interrupt handler the syscall exit(-1) is called.

>> B10: What advantages or disadvantages can you see to your design

>> for file descriptors?

Since every file descriptor is unique for each process, it eliminates

the need to account for race conditions.

>> B11: The default tid_t to pid_t mapping is the identity mapping.

>> If you changed it, what advantages are there to your approach?

We used the default tid_t to pid_t mapping.

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?

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

>> 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?

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

>> Any other comments?