CS 130: Project 2 - User Programs Design Document

Group Information

Please provide the details for your group below, including your group's name and the names and email addresses of all members.

- **Group Name**: *lorin&lucy*
- **Member 1**: Jintong Luo `<luojt2022@shanghaitech.edu.cn>`
- **Member 2**: Lufei Li `<lilf2024@shanghaitech.edu.cn>`

Preliminaries

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

No.

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.

We didn't declare new struct or change struct member or static variable 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?

***Argument Parsing**: I used the `strtok_r()` function to split the command-line string, using spaces as delimiters.*

***Argument Order**: The extracted arguments are stored in the `argv[]` array in the order they are extracted. By looping through `strtok_r()`, the addresses of each argument are stored in `argv[]`, ensuring that their order matches the user's input.*

***Avoiding Stack Overflow**: In `load` function, call `setup_stack()` to make `esp` (stack pointer) to point to the `PHAS_BASE` (top of stack). We align the stack pointer `esp` to a 4-byte boundary to ensure correct memory allocation boundaries.*

Rationale

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

`strtok_r()` is the thread-safe version of `strtok()`. In Pintos, the kernel splits commands into the command line (executable name) and arguments. To ensure that each thread can correctly track the position of argument parsing in a multithreaded environment, we need to store the address of the arguments in a place that can be accessed later. This way, each thread calling `strtok_r()` has its own independent pointer (`save_ptr`), which allows it to remember its position without interfering with other threads.

> **A4:** In Pintos, the kernel separates commands into an executable name and arguments, while Unix-like systems have the shell perform this separation. Identify at least two advantages of the Unix approach.

Higher Flexibility: Users could select the shell that best meets *their needs without relying on the fixed functionality provided by the kernel. Shells offer powerful scripting capabilities, enabling users to write complex command sequences using scripting languages without modifying kernel code.*

Better Security: Since the parsing of command-line arguments is done by the user-space shell rather than directly by the kernel, this reduces the risk of exposing the kernel to potential security vulnerabilities.

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.

In `thread.h`:

```
struct child_thread
{
   tid_t tid;               /* tid of this child thread */
   bool is_dead;            /* whether the child thread has been dead */
   struct list_elem elem;       /* list elem to be added in allchild_list
*/
   struct semaphore sema_wait;      /* semaphore to control waiting */
   int exit_status;         /* the status the child thread exit with */
};

struct file_entry
{
   int fd;              /* file description ID */
   struct file* file;           /* the corresponding file */
   struct list_elem file_elem;      /* list elem to be added in filelist */
};

struct thread
{
   struct thread* parent;       /* the parent of this thread */
   struct semaphore sema_load;      /* control the logic of the child
process and complete
                        the parent's wait for the child */
   struct child_thread* child;      /* the child of this thread */
   struct list allchild_list;       /* Store all the children */
```

```
    struct list filelist;        /* list of all files */
    struct file* curr_file;      /* current processing file */
    int curr_file_fd;            /* current file descriptor */
    int exit_status;         /* exit status */
    bool success;            /* check whether the child thread execute
                         successfully */
};
```

In `syscall.h`:

```
/* Ensure that only one process is executing file system at a time. */
static struct lock file_lock;
```

> **B2:** Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

File descriptors are unique just within a single process. Each process tracks a list of its file descriptors (stored in struct `thread` as a list of struct `file_entry`), as well as the current file descriptor number. Our struct `file_entry` associates the file descriptor numbers with the corresponding file.

Algorithms

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

*In reading and writing, we first check the user pointer. An invalid user pointer should terminate the process. In **read**:*

- Check whether the `buffer` pointer is valid.
- Check the value of `fd`:
  - If `fd = 0(STDIN_FILENO)`, then call `input_get()` and return the return value.
  - If `fd = 1(STDOUT_FILENO)`, it is invalid, we should terminate the process.
  - Else, Use `get_file_entry(*user_ptr)` to find the file object associated with the file descriptor. If the file object exists, return the number of bytes read (stored in `f->eax`). Otherwise, return `-1` to indicate failure.

*In **write**:*

- Check whether the buffer pointer is valid.
- Check the value of `fd`:
  - If `fd = 0(STDIN_FILENO)`, it is invalid, we should terminate the process.
  - If `fd = 1(STDOUT_FILENO)`, then call `putbuf()` and return the return value.
  - Else, Use `get_file_entry(*user_ptr)` to find the file object associated with the file descriptor. If the file object exists, return the number of bytes written (stored in `f->eax`). Otherwise, return `-1` to indicate failure.

In addition, we should call `acquire_process_lock()` to ensure that only one process at a time is executing this file. We need to call `release_process_lock()` then as well.

> **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?

When a system call copies a full page (4,096 bytes) of data, the minimum number of page table inspections is 1, and the maximum is 2. This depends on whether the data spans one page or two pages. For a system call that copies only 2 bytes of data, the minimum number of page table inspections is also 1, and the maximum is 2. This is because the data might span one page or two pages. The number of page table inspections can be reduced by optimizing the page table structure or using a caching mechanism. In an ideal scenario, with a high TLB cache hit rate, the number of page table inspections can be significantly reduced, potentially even approaching zero.

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

- *Call `process_wait (tid_t child_tid UNUSED)`.*
- *`process_wait (tid_t child_tid UNUSED)` first iterates through the current process's child list `allchild_list` to find a child process with a tid matching the given `child_tid`.*
- If a matching child process is found, it checks whether the child process has already been waited for. If it has, the function returns `-1` to indicate that waiting cannot be repeated. If the child process has not been waited for, the `is_waited` flag is set to true, and `sema_down` is called to block the parent process until the child terminates. The child process notifies the parent using a semaphore (`sema_up`) when it exits.
- *`process_wait` also retrieves the exit status from the child process's structure and returns it. After retrieving the exit status, the child process entry is removed from the parent's child list.*

> **B6:** Accessing user program memory at a user-specified address may fail due to a bad pointer value, requiring termination of the process. Describe your strategy for managing error-handling without obscuring core functionality and ensuring that all allocated resources (locks, buffers, etc.) are freed. Give an example.

Before accessing user memory, check the validity of the pointer using functions like `check_pointer` or `check_buffer`. If the pointer is invalid, terminate the process directly using the function `terminate()` to avoid unpredictable behavior from subsequent operations.

***Example**: When dealing with bad-ptr calling `sys_read()`, such as `NULL` pointer or address greater than `PHYS_BASE` are detected in `check_pointer()`, we would call `terminate()`.*

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"?

In my implementation, the `sys_exec()` system call relies on synchronization using a semaphore to ensure that the parent thread waits until the new executable has finished loading. Specifically:

- *In the `process_execute()` function, the parent thread calls `sema_down()` to block itself until the child thread completes the loading process.*

- *The child thread, which is responsible for loading the new executable, calls `sema_up()` to unblock the parent thread once the loading is done.*

The success/failure status of the loading process is communicated back to the calling thread using the `thread_current()->success` flag. In the `start_process()` function, the child thread sets the `thread_current()->parent->success` flag based on whether the loading is successful. After being unblocked, the parent thread checks this flag. If the loading was successful, it returns the tid of the new thread. Otherwise, it returns `-1`.

> **B8:** Consider a 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?
> - P calls `wait(C)` after C exits?
> - P terminates without waiting, before C exits?
> - P terminates after C exits?
> - Are there any special cases?

- ***P calls `wait(C)` before C exits**: In the `process_wait()` function, P blocks itself using `sema_down()` until C calls `sema_up()` to wake it up. After P retrieves the exit status, it removes the child process entry from the child list, thereby releasing the associated resources.*
- ***P calls `wait(C)` after C exits**: If C has already exited, P calling `sys_wait()` will directly retrieve the exit status of C without blocking. P removes the child process entry from the child list after obtaining the exit status, ensuring that resources are released.*
- ***P terminates without waiting, before C exits**: When P terminates, it calls `process_exit()`, which releases all resources, including file descriptors and the child process list. C will continue to run until it completes and exits. When C exits, it attempts to notify P, but since P has already terminated, the exit status of C is ignored.*
- ***P terminates after C exits**: There is nothing. When P terminates, it releases all resources, including the child process list. Since C has already exited and its resources have been cleaned up, there is no resource leakage.*

Rationale

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

I chose to implement access to user memory using the `check_pointer` and `check_buffer` functions to pre-check the validity of pointers. The main reasons for this approach are as follows:

- ***Safety**: Before accessing user memory, the `check_pointer` function is used to verify that the pointer is valid. If the pointer is invalid, the process is terminated immediately to prevent unpredictable behavior. This approach effectively prevents illegal pointer access, ensuring the stability and security of the system.*

- ***Efficiency**: By pre-checking the validity of pointers, we can avoid complex checks during each memory access. Once a pointer is verified, subsequent memory operations can proceed safely, improving the execution efficiency of the code. For contiguous memory accesses, batch checking can be performed to reduce redundant validation overhead.*

- ***Resource Management**: When checking the validity of pointers, if an invalid pointer is detected, the `terminate` function can be called to terminate the process and release all allocated resources, ensuring no resource leaks occur.*

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

Advantages:

- *File descriptors are dynamically allocated, allowing for flexible handling of an arbitrary number of open files without a fixed-size array limitation.*
- *Each file descriptor is encapsulated in a `file_entry` structure, which includes both the file pointer and the file descriptor number. This makes it easy to manage and access file-related information.*
- *The design allows for a straightforward API for file operations. Functions like `sys_open()`, `sys_close()`, and `sys_read()` can easily locate and manipulate file descriptors through the list.*

Disadvantages:

- *Accessing a specific file descriptor requires traversing the list, which can be inefficient for a large number of open files. This might introduce performance overhead, especially in systems with many concurrent file operations.*
- *Managing a dynamic list of file descriptors adds complexity to the code. Operations like adding, removing, and searching for file entries require careful handling to avoid bugs.*

> **B11:** The default `tid_t` to `pid_t` mapping is the identity mapping. If you changed it, what advantages does your approach offer?

- *A dedicated `pid_t` can be used to manage process-level attributes (e.g., process state, exit status) independently of thread-level attributes. This can simplify process management logic.*
- *You can handle special cases more easily.*
- *A separate `pid_t` can help in managing parent-child relationships more effectively.*

Survey Questions

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

No.