Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Qizhi Chen 2100012959@stu.pku.edu.cn

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

copilot , chatgpt

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.

```
struct file_descriptor{
    int fd;
    struct file *file;
    struct list_elem elem;
};

struct thread
  {
    /* Owned by thread.c. */
    int exit_code;                      /**< Exit code of the thread. */
    struct semaphore sema_exec;         /**< Semaphore for waiting on child. */
    bool load_success;                  /**< Load success of the thread. */
    struct list file_list;              /**< List of file descriptors. */
    struct file* exec_file;
    int max_fd;                         /**< Max file descriptor. */
    struct list_elem elem;              /**< List element. */
    struct thread *parent;              /**< Parent thread. */
  };
```

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?

First alloc a new page to restore the coming string.

- For the father process, pass the argument string to child process and wait until the child thread loading successfully or unsuccessfully.

- the child thread use "load" function to execute elf file.
- Then,the child thread parse the command string, push the argument into the stack.
- the order argument passing just as the pintos document
- ```
  uintptr_t align = (uintptr_t)if_.esp % 4;
  if_.esp -= align;
  memset(if_.esp, 0, align);
  // here we push the address of the argv[...][...] to the stack
  const size_t ptr_size = sizeof(void *);
  // word align the stack
  if_.esp -= ptr_size;
  memset(if_.esp, 0, ptr_size);
  // push the address of argv[...] to the stack
  int i;
  for(i = argc - 1; i >= 0; i--){
    if_.esp -= 4;
    memcpy(if_.esp, &argv[i], 4);
  }
  // push the address of argv
  char* argv0 = if_.esp;
  if_.esp -= 4;
  memcpy(if_.esp, &argv0, 4);
  // push the number of arguments
  if_.esp -= 4;
  memcpy(if_.esp, &argc, 4);
  //fake return address
  if_.esp -= 4;
  memset(if_.esp, 0, 4);
  ```
- To deal with the situation that write the elf file, we use `file_write_deny` to forbid this method.

- `strlcpy(argvcopy, filename, PGSIZE);`
- we cut down the argument command line as the largest length is "PGSIZE = (1 << 12)".
- When stack overflow, the thread structure has a "magic" variance which is some magic constant number at first. By checking whether it's changed, we can quickly understand whether stack overflow.

RATIONALE

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

Because strtok () uses global data and is unsafe to use in threaded programs such as kernel

> 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. Flexibility: The Unix shell approach allows for greater flexibility in terms of the commands that can be executed, as it allows for the use of shell scripts and other programming constructs to combine and manipulate commands.
2. Modularity: The Unix shell approach promotes modularity by separating the responsibilities of the kernel and the shell. This separation of concerns makes it easier to modify and extend the functionality of the shell without affecting the kernel, and vice versa. Additionally, the use of shell scripts allows for easy sharing and reuse of code across different users and systems.
  ```

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.

```
struct file_descriptor{
    int fd;
    struct file *file;
    struct list_elem elem;
};
struct child_entry{
    tid_t tid;              // Child's tid
    struct thread *t;       // Child's thread
    int exit_code;          // Child's exit code
    bool is_alive;          // Child is alive or not
    bool is_waiting_on;     // Child is waiting on or not
    struct semaphore sema; // Semaphore to let parent wait on child
    struct list_elem elem;
};

struct thread
  {
//.....
    int exit_code;                      /**< Exit code of the thread. */
    struct semaphore sema_exec;         /**< Semaphore for waiting on child. */
    bool load_success;                  /**< Load success of the thread. */
    struct list file_list;              /**< List of file descriptors. */
    struct file* exec_file;
    int max_fd;                         /**< Max file descriptor. */
    struct list_elem elem;              /**< List element. */
    struct thread *parent;              /**< Parent thread. */
      /**< List of children. */
    struct child_entry *as_child;         /**< Child entry. */
  };
```

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

```
struct file_descriptor{
    int fd;
    struct file *file;
    struct list_elem elem;
};
```

one thread owns some file descriptor, which has a pointer to a open file node.

File descriptors are unique within a single process, meaning that no two file descriptors within the same process can have the same value. However, file descriptors are not necessarily unique across the entire operating system.

ALGORITHMS

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

user use "syscall" and with a exception handler, and get into `syscall_read` or `syscall_write` function.

In the function, kernel first check whether the syscall argument are feasible or not.

then if fd=0 or fd =1 , just use `input_getc` or `putbuf` .

otherwise, traverse the thread's file descriptor to find the corresponding file,

then use `file_read` or `file_write` , assisted with `lock`

> 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?

a full page:

- least: If the entire page is contiguous and mapped to a single page table entry, then only one inspection of the page table will be required to copy the entire page.
- greatest: If the page is mapped across multiple page tables, then the kernel should call `pagedir_get_page` twice.

same as 2 bytes. depending on whether the data is aligned to the page boundary. 1 is needed is aligned ans 2 if the first byte is in the end of the page block while the second byte is in the second block.

One possible approach is for me to align all data with four bytes, so accessing two bytes of memory usually requires me to query a page table.

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

When a user process calls "wait" with the TID of a child process as an argument, the kernel searches for a corresponding child_entry struct in the parent's child_list. If found, the kernel sets the child_entry's "is_waiting_on" flag to true to indicate that the parent is waiting for the child to terminate.

If the child process is still alive, the kernel blocks the parent process by calling sema_down() on the child_entry's "sema" semaphore. The child process will eventually call sema_up() on this semaphore when it terminates.

When the child process terminates, it sets its "exit_status" field to its exit code and wakes up any waiting parent processes by calling sema_up() on its child_entry's "sema" semaphore. The parent process that was waiting on the child's termination is unblocked and can then retrieve the child's exit code by calling "wait" again with the child's TID as an argument.

> 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.

The strategy adopted in the provided code snippet is to immediately terminate the process upon encountering a bad pointer value. This is done by checking if the faulting address is a valid user address, and if not, printing an error message, setting the return value to -1, and calling thread_exit() to terminate the process.

To avoid obscuring the primary function of code in a morass of error-handling, the code snippet follows a simple and straightforward approach of checking for errors as soon as they can occur. For instance, when checking for a bad pointer value in the "page_fault" function, the check is performed as soon as possible after obtaining the faulting address. Similarly, in the "syscall_handler" function, each system call is validated by performing checks on the arguments as soon as they are accessed.

To ensure that all temporarily allocated resources are freed upon encountering an error, the code snippet uses a combination of stack unwinding and semaphores. For example, in the "process_execute" function, a semaphore is used to synchronize the parent and child threads, and the semaphore is released only after the child process has successfully loaded its executable. If an error occurs during the process of loading the executable, the semaphore is immediately released, allowing the parent thread to continue executing and eventually terminate the child process.

The process releases its memory in thread_exit. If it has no parent, it frees its own memory; otherwise, it lets its parent process release its memory.

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"?

To ensure that the "exec" system call doesn't return before the new executable has completed loading, the code uses a semaphore to synchronize the parent and child threads. When the child process finishes loading the new executable, it signals the parent thread by releasing the semaphore. The parent thread waits for the semaphore to be signaled before returning from the "exec" system call.

The load success/failure status is passed back to the thread that calls "exec" through the use of a boolean flag in the thread struct called "load_success". If loading the new executable fails, the flag is set to false and the exit code of the current thread is set to -1. The parent thread then waits for the child thread to signal the semaphore before returning from the "exec" system call. When the child thread signals the semaphore, the parent thread checks the "load_success" flag to

determine whether loading the new executable was successful or not. If it was unsuccessful, the "exec" system call returns -1.

> 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?

1. By a simple but useful method , disable the interrupt when using process_wait and use sema_down to switch current thread.
2. if one process have no father(his father had died before waiting him), just free all needed resource. Otherwise, when a process is died , checking all his child threads , if any of them has already died but has not been wait, free all his child's resources.

RATIONALE

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

Implementation of accessing user memory from the kernel using the `get_user` function is simple and complete. The kernel directly calls this function on the given user address, and if an error occurs, it can be caught in the `page_fault` handler. This approach allows for efficient and reliable access to user memory from the kernel, and also enables proper handling of errors that may occur during the access.

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

advantage:

- The use of a lock ensures that multiple threads cannot access the same file descriptor simultaneously, which prevents race conditions and ensures thread safety.
- The file descriptor table is part of the thread struct, so each thread has its own file descriptor table, which allows for thread isolation.

disadvantage:

- Using the list as an index to search for the file descriptor cost a lot of time which also means there's a limitation to the maximum number of the file descriptor

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

- processes could have multiple threads.
- Having a separate thread ID and process ID can provide more flexibility. For example, it could allow for more fine-grained control over resource allocation, scheduling, and thread management.
- By having a separate identifier for threads, it becomes easier to separate concerns between processes and threads. This can make it easier to reason about the behavior of different parts of the system.