Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project 5

Group 11
Machado Ferreira Simone, Zeneli Jora, Del Pozzo Daniele

Spring Semester, May 15, 2025

1 Questions

Consider file `syscall.c`

1. What are the fields of your hash structure? Which field is used as an id when the lookup of an entry is performed?
2. Here is the list of needed syscall codes at the end of your implementation:
   a) `SYS_HALT` /* Halt the operating system. */
   b) `SYS_EXIT` /* Terminate this process. */
   c) `SYS_EXEC` /* Start another process. */
   d) `SYS_WAIT` /* Wait for a child process to die. */
   e) `SYS_CREATE` /* Create a file. */
   f) `SYS_REMOVE` /* Delete a file. */
   g) `SYS_OPEN` /* Open a file. */
   h) `SYS_FILESIZE` /* Obtain a file's size. */
   i) `SYS_READ` /* Read from a file. */
   j) `SYS_WRITE` /* Write to a file. */
   k) `SYS_SEEK` /* Change position in a file. */
   l) `SYS_TELL` /* Report current position in a file. */
   m) `SYS_CLOSE` /* Close a file. */

   For each one of them, specify the syscall number, how many parameters are on the stack and their meaning, and the meaning of the return value stored in the `eax` register (if present).

3. Some of these syscalls take a pointer as parameter. Is this pointer always valid? What should you check about it?
4. Only one thread at a time can manipulate the filesystem. How do you ensure this?
5. File descriptors from 0 to 2 (included) are reserved for stdin, stdout and stderr. How do you generate a new file descriptor? In which syscall?
6. The current implementation of the `write` syscall only writes to standard output (file descriptor: `STDOUT_FILENO`). How do you allow the `write` syscall to write on both stdout and a file?

2 Answers

File `syscall.c`

1. Our hash structure is composed by the four following fields:
   - `struct hash_elem` – Used by the hash related functions to link an item to the hash table
   - `tid_t owner` – ID of the thread responsible for calling the process
   - `int fd` – Integer containing the file descriptor number
   - `struct file *file` – Pointer to the file structure associated with the file descriptor

   The fields used as an id when the lookup of an entry is performed are the `fd` integer and the `owner` thread ID, this can be seen both in the hash function `unsigned hash_file_item` and the function `bool compare_item`. In the `unsigned hash_file_item` function the `fd` and the `owner` are combined with a XOR operator to generate the hash value, while in the `bool compare_item` function they are used to sort items by `owner`, or if that isn't possible by `fd`.

2. a) `SYS_HALT` – Syscall number: 0 – No stack parameters, doesn't return any value through `eax`. It terminates PintOS.
   b) `SYS_EXIT` – Syscall number: 1 – takes an integer `status` on the stack, which represents the exit status of the terminating process ,doesn't return any values through `eax`.
   c) `SYS_EXEC` – Syscall number: 2 – takes a char pointer `cmd_line`, used to run program and pass it command line arguments values. Returns process id as an integer of the process through `eax` or -1 if the process encountered an error.
   d) `SYS_WAIT` – Syscall number: 3 – takes an integer `pid` as a parameter, used to identify a child process and if found, waits for it to complete. Returns exit status as integer of the child waited on through `eax` if the wait was successful or it was the first time calling wait on that child. Returns -1 in case the wait call was redundant, the pid wasn't found, the child still hasn't exited, or the waiting process was interrupted before the child could exit.
   e) `SYS_CREATE` – Syscall number: 4 – takes a char pointer `file`, which is the given name of the file to be created and an integer `initial_size`, which as the name suggests is the initial size of the new file. Returns a 1 through `eax` if the file was created or 0 if any issue was encountered.
   f) `SYS_REMOVE` – Syscall number: 5 – takes a char pointer `file`, which is the name of the file to be deleted. Returns a 1 through `eax` if the file was deleted successfully or 0 if any issue was encountered.

g) `SYS_OPEN` – Syscall number: 6 – takes a char pointer `file`, which is the name of the file to be opened by the user. Returns the file descriptor associated with the file through `eax` upon successful opening, -1 if any issue was encountered.

h) `SYS_FILESIZE` – Syscall number: 7 – takes an integer `fd`, which is the file descriptor of the file the user is requesting the size of. Returns the size of the file as integer through `eax`.

i) `SYS_READ` – Syscall number: 8 – takes an integer `fd`, which is the file descriptor of the file the user is requesting to read or 0 if the user wishes to read from stdin, a void pointer `buffer`, used as buffer to write characters taken from the file (to be read), an unsigned integer `size`, which is the size of the buffer, used to bound the amount of memory allocated for the read operation, that being the amount of bytes that can be written in the buffer. Returns the number of bytes that were read through `eax` or -1 if any issue is encountered.

j) `SYS_WRITE` – Syscall number: 9 – takes an integer `fd`, which is the file descriptor of the file the user wishes to write to or 1 if the user wishes to write to stdout, a void pointer `buffer` which is an array of characters to be written into the file, an unsigned integer `size`, which as the name suggests is the number of bytes that the user wishes to write, which must be less than or equal than the size of the `buffer`. Returns the number of bytes written to the file as integer through `eax`.

k) `SYS_SEEK` – Syscall number: 10 – takes an integer `fd`, which is the file descriptor of the file upon which the user wishes to offset the file position for the next read/write operation, an unsigned integer `position` which is the byte offset to be applied. Returns no value through `eax`.

l) `SYS_TELL` – Syscall number: 11 – takes an integer `fd`, which is the file descriptor of the file the user wishes to know the file position of. Returns an unsigned integer through `eax` which is the offset from the start of the file. This offset is expressed in bytes and represents the position of the next read/write operation that will be applied to the file.

m) `SYS_CLOSE` – Syscall number: 12 – takes an integer `fd`, which is the file descriptor of the file the user wishes to close. Returns no value through `eax`

3. The pointer isn't always valid, it needs to pass the following checks, that are bundled inside the `check_user_address` function for convenience, to ensure its validity before it can be used in the kernel space:

   a) it's not a null pointer, checked by doing `ptr != NULL`

   b) it's not a kernel space address, checked by using the `is_user_vaddr (ptr)` function, returns true if it's a virtual address, thus passes the check.

   c) it's owned by the current process, this is true if it resides in its memory page, checked by using the `pagedir_get_page(thread_current ()->pagedir, ptr)` function. This function a kernel virtual address corresponding to the physical address or a NULL pointer if the pointer is unmapped, thus not available.

   The `check_user_address` returns true if it passes all checks, thus the pointer can be safely used in kernel space.

4. Ensuring that only one thread can access the file system at a time is done with the use of a lock `struct lock file_lock`.
   Anytime atomicity needs to be ensured when manipulating the filsystem, the lock is acquired by a thread using the `lock_acquire(&file_lock)` and released after all of the needed operations are completed with the `lock_release(&file_release)` function.

5. Generating new file descriptors is done through the `SYS_OPEN` syscall, implemented in the syscall_open function.
   Going step by step through the function:

   a) the lock is acquired

   b) The owner's thread id is stored.

   c) an index i is set to FD_START, which is defined as 3, since values 0, 1 and 2 are already used by the stdin, stdout and stderr. Intuitively FD_START is the first available fd that can be given to any file.

   d) iterator which is a `file_item` is initialized.

   e) iterator is used to iterate over the hash table until an empty "spot" is found that could house the `owner` and `fd` tuple by checking whether the currently checked entry composed by `owner` and `i` is found in the hash table, if it isn't then the entry is suitable for a new fd.

   f) As soon as a suitable spot is found, the new fd is added as an entry in the hash table with the newly found `owner` and `i` tuple.

   g) The `fd` is then returned to the caller to be used for further file specific actions

   h) the lock is released

   Following the flow of the function it can be seen, the lock is acquired before interacting with the file system and the hash table, thus atomicity is ensured, and since the lowest free file descriptor is used, it ensures that the hash table isn't cluttered with useless unused information. Once a file is closed, the corresponding fd is freed up for further use by other files.

6. A check is added to the `syscall_write` function, that is applied to the fd that the user passed the function through the stack. If the fd corresponds to STDOUT_FILENO, then the write operation is applied to the standard output by sending the data in `buffer` to the console using `putbuf(buffer, length)`. Else if fd > 2, then the file associated with the fd is fetched and the data in `buffer` is written in it using `file_write(fi->file, buffer, length)`.