Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project 4

Group 11
Zeneli Jora, Machado Ferreira, Del Pozzo Daniele

Spring Semester, May 7, 2025

1 Questions

1. Please answer the following questions regarding your implementation of the `wait` system call.

   (a) How is your syscall initialized?

   (b) What existing function is used to wait for a process?

   (c) How is the parent made to wait for the child's completion?

   (d) How is the correct exit status of the child returned?

   (e) How is waiting avoided for already-terminated children?

   (f) How is an error returned for incorrectly terminated children?

   (g) How is waiting prevented for non-child processes?

   (h) How is double-waiting for the same child prevented?

2. Please answer the following questions regarding your implementation of the `exec` system call.

   (a) How is your syscall initialized?

   (b) What existing function is used to execute a new process?

   (c) In which function does the parent wait for the child to be fully created?

   (d) How and where does the child notify the parent of its creation?

   (e) How do you make sure the parent knows if the child fails ?

3. Please answer the following questions regarding your management of memory access.

   (a) How do you check if the address is a valid pointer?

   (b) How do you check if the address is in the kernel space?

   (c) How do you check if the address is in the user space?

2 Answers

1. Answers regarding the `wait` system call.

   (a) The syscall is triggered by the user who passes (pushes onto the stack) the process ID of the process they want to wait for. During kernel initialization, the syscall is set up by adding the `syscall_wait` function pointer to the syscall array, which contains all the relevant syscall handlers. When a syscall is made, the `syscall_handler` identifies the requested syscall and invokes the appropriate function. Inside `syscall_wait`, we begin by retrieving the process ID that was passed by the user.

   (b) The `process_wait` function is used to wait for a process.

   (c) The parent is made to wait for the child's completion by setting the `parent_waiting` flag in the child's struct to true, indicating that the parent is waiting. Then, the parent is blocked by `calling thread_block()`, which suspends its execution until the child completes.

   (d) The correct exit status of the child is returned by locating the child's struct in the parent's children list. Once the child struct is found, the child sets its `exit_status` variable. This ensures that, regardless of what happens to the child before or after the parent calls `wait`, the parent will always retrieve the correct exit status.

   (e) Waiting for already-terminated children is avoided by immediately returning their exit statuses instead of blocking. This is managed by checking if `child->has_exited` is true. If the child has already exited, the code skips the `thread_block()` call, directly returns the exit status, and removes and frees the child struct that the parent still holds. If the parent tries to wait again for an already exited child, the condition `if (child == NULL)` will be true because the child struct was removed from the list during the first call, and `-1` will be returned as requested.

   (f) An error is returned for incorrectly terminated children by checking for several conditions. According to the wait specification, if a child process did not call `exit()` but was terminated by the kernel (e.g., due to an exception), the `wait(pid)` call must return `-1`. This is handled by checking if `child == NULL`, which occurs if the child was freed in `process_execute` due to returning `-1` from `thread_create`. In this case, `-1` is returned immediately.

   If the child was not loaded successfully, the condition `if (!child->load_success)` ensures the child is removed from the parent's children list, and `-1` is returned.

   Another case of incorrect termination happens when a child process is loaded but later terminated by the kernel. In this case, the child process calls `thread_exit` and eventually `process_exit`, which sets the child's `has_exited` value to true. When `wait` is called, it checks `if (child->has_exited)`, and returns the child's exit status. However, if the exit status is not set correctly (e.g., it remains `TID_ERROR` by default), it returns that value, which is `-1`.

(g) Waiting is prevented for non-child processes by verifying if the caller is indeed the parent of the process it is trying to wait for. This check is performed with the statement `if (child == NULL)`. If this condition is true, it indicates that the caller is attempting to wait for a process that is not its child. In such cases, the function immediately returns `-1`, thus preventing any waiting for non-child processes.

(h) Double-waiting for the same child is prevented by removing the child's struct from the parent's children list after the first successful wait. Once the child is waited on, the second call to `wait` will encounter `if (child == NULL)`, as the child struct has been cleared from the list. This check immediately returns `-1`, preventing the parent from waiting on the same child again. Similarly, if the child has already terminated and was cleared from the list during the first wait, the second wait will also result in `child == NULL`, ensuring that no further waiting occurs for that child.

2. Answers regarding the `exec` system call.

(a) The syscall is initialized when the user pushes the pointer to the command line string onto the stack. In the kernel, the syscall is set up by adding the function pointer for `syscall_exec` to the call array during `syscall_init`, which holds the function handlers for all syscalls. When a syscall is made, `syscall_handler` reads the specific syscall and calls the corresponding function. In the case of `syscall_exec`, the process begins by reading the command line string pointer passed by the user.

(b) The existing function used to execute a new process is `process_execute`.

(c) The parent waits inside the `process_execute` function. As soon as the child has been created correctly, the parent takes the child's instance struct from the list of children and waits on its semaphore. This is done at the line `sema_down(&child_entry->exec_sync)`.

(d) The child notifies the parent of its creation in two places during the process execution:

1. Inside `start_process`: After verifying that the loading was successful, the child notifies the parent by calling `sema_up(&child_me->exec_sync)`. This signals the parent that the child has been correctly initialized and the program has been loaded successfully.

2. On an unsuccessful load: If the loading fails, the child thread calls `thread_exit`, which in turn calls `process_exit`. During this process, the parent is notified by waking up from its wait, and the child's struct instance is found in the parent's children list. The child's resources are deallocated, and the parent will return `-1` indicating the failure.

(e) To ensure the parent knows if the child fails to load, we do the following:

1. Update the `oad_successful` flag: The child struct, which is held by the parent, has a `load_success` flag. If the child fails to load correctly, this flag is updated to reflect the failure.

2. Return `-1` on failure: Instead of returning the child's process ID for a successful load, if the child fails to load, we return `-1`. This informs the parent (and the process calling the syscall) that the child did not load correctly.

3. Answers regarding the `management of memory access`.

(a) To check if the address is a valid pointer, we perform the following step:

Null pointer check: We ensure the address is not a null pointer by checking the condition `cmd_line == NULL`. If this condition is true, we return `-1` and do not proceed with the execution of the process (i.e., we do not call `process_execute`).

This ensures that we don't attempt to execute a process with an invalid address.

(b) To check if the address is in kernel space, we compare the address with `PHYS_BASE`. Specifically, we use the function `is_kernel_vaddr(const void *vaddr)`, which checks if the provided address is equal to or greater than `PHYS_BASE`.

In the case of `syscall_exec`, we call this function with the command line pointer `(cmd_line)` like this: `if(is_kernel_vaddr(cmd_line))`.

(c) To check if an address is in user space, we ensure that the address is part of the process's page. This is done by calling the `pagedir_get_page` function, which checks if the address belongs to the user space of the current process.

The function works by looking up the physical address corresponding to the virtual address. It internally calls `lookup_page`, which checks whether the address is part of the process's page directory. If the address is outside the page's boundaries, `lookup_page` triggers a special branch and returns `NULL`.

This ensures that the address passed by the user is within the valid memory space of the process. If the address is not part of the process's page, it returns `NULL`, indicating the address is invalid.

3 Files Changed

Modified Files:

1. /pintos-env/pintos/threads/thread.c
2. /pintos-env/pintos/threads/thread.h
3. /pintos-env/pintos/userprog/syscall.c
4. /pintos-env/pintos/userprog/process.c

4 Changes

1. /pintos-env/pintos/threads/thread.c
   `tid_t thread_create(...)` (`added`):
   - We updated this function to initialize and add the child's thread struct to the parent's list of children. This ensures that every child thread is properly tracked by its parent.
   `static void init_thread(...)` (`modified`):
   - We made changes to this function so that it initializes the list for each user program thread struct. We perform this initialization here instead of in `tid_t thread_create`, as `init_thread` is called during the initialization of any thread struct, ensuring that all threads have a properly initialized and usable list for child management.

2. /pintos-env/pintos/threads/thread.h
   `struct child` (`added`):
   We introduced the `child` struct to store information about each child process of a parent. This is crucial because we want the parent to still be able to retrieve useful data about the child after it exits, especially when invoking the `wait` syscall.

   The `struct child` has the following fields:

- `struct list_elem elem`: Used to store the child struct in the parent's children list.
- `tid_t tid`: Stores the child's thread ID (TID).
- `bool is_waiting` : Indicates if the parent is waiting for this specific child.
- `bool has_exited`: Tracks whether the child has exited.
- `struct semaphore exec_sync`: Used for synchronization between the child's loading and the parent process.
- `bool load_success`: Indicates if the child's loading was successful.
- `int exit_status`: Stores the child's exit status, which the parent can retrieve.

`struct thread` (`modified`):
We removed redundant variables that were previously held within `struct thread` and are now stored in `struct child`. Additionally, we added a `children` list (`struct list children`) to track all child processes of a thread. This modification ensures better organization and management of child processes.

3. /pintos-env/pintos/userprog/syscall.c
`void syscall_init` (`modified`):
The function was updated to call `syscall_exec` and `syscall_wait` for the respective `EXEC` and `WAIT` system calls.

`static void syscall_exec` (`added`):
This new function is designed to handle the `EXEC` system call. It first checks that the command line input passed on the stack is a valid pointer, ensures it is not a kernel memory address, and verifies that it belongs to the caller's page. After validation, it calls `process_execute`, which attempts to load the user process and returns either the process `ID (PID)` or `-1` if loading fails. The function then returns the result of `process_execute` by storing it in the `eax` register of the interrupt frame.

`static void syscall_wait` (`added`):
This function handles the `WAIT` system call. It retrieves the process `ID (PID)` passed by the caller, then calls `process_wait` with this `PID`. After `process_wait` completes, the function returns the result from the system call by placing it in the `eax` register of the interrupt frame.

4. /pintos-env/pintos/userprog/process.c
`#include "threads/synch.h"` (`added`):
We added this import to use synchronization functions related to semaphores.

`#include "kernel/list.h"` (`added`):
We added this import to utilize functions related to lists.

`#include "threads/malloc.h"` (`added`):
We added this import to use memory management functions, like `free`, for deallocating child structs.

`tid_t process_execute`` (`modified`):
We updated this function to wait for the child process to initialize. If the child is not found, it returns `-1`. Once the child wakes the parent, if the child failed to load, the parent frees its struct and returns `-1`; otherwise, it returns the child's `PID`.

`static void start_process` (`modified`):
This function was modified to update the child's struct with its loading status. If the child loads successfully, it wakes up the parent right after setting up the stack, before beginning execution.

`int process_wait` (`modified`):
We updated this function to handle all possible cases:

- If the child doesn't exist, it returns -1.
- If the child didn't load properly and is being waited upon (which should not happen), it returns -1.
- If the child already exited, it returns the exit status on the first wait and -1 on a second wait.
- If the child is already being waited for, it returns -1.
- If the caller can wait for the child, it blocks the caller using `thread_block`. When the child unblocks it, the function returns its exit status and frees the child struct.
- If the child is not yet exited and the caller is awakened, it returns -1 and allows another thread to wait for the child.

`void process_exit` (`modified`):
This function was updated to retrieve the calling child's struct held by the parent. If the struct exists, it updates the `has_exited` flag and `exit_status` for the parent. After clearing the memory page, it prints the exit string. If the child had an unsuccessful load, it wakes up the parent waiting for the child's initialization. For any other exit, it unblocks the parent if waiting, and frees the child structs.