Operating Systems Design
PintOS Part2 : User Programs

CONTENTS

Notice

- All implementations on the slides are merely illustrative examples, and alternative approaches are also feasible.
- Additionally, feel free to incorporate any necessary functions or features required for the implementation.
- Part2 project description is based on the part1 answer code posted to eclass.

Extra: export the container

- Export container as tar file
  - $ docker export -o [file name].tar [container name]

- Testing your file
  - $ docker import [file name].tar [image name]:[tag name]
  - $ docker run –it –name [container name] [image name]:[tag name] /bin/bash

0

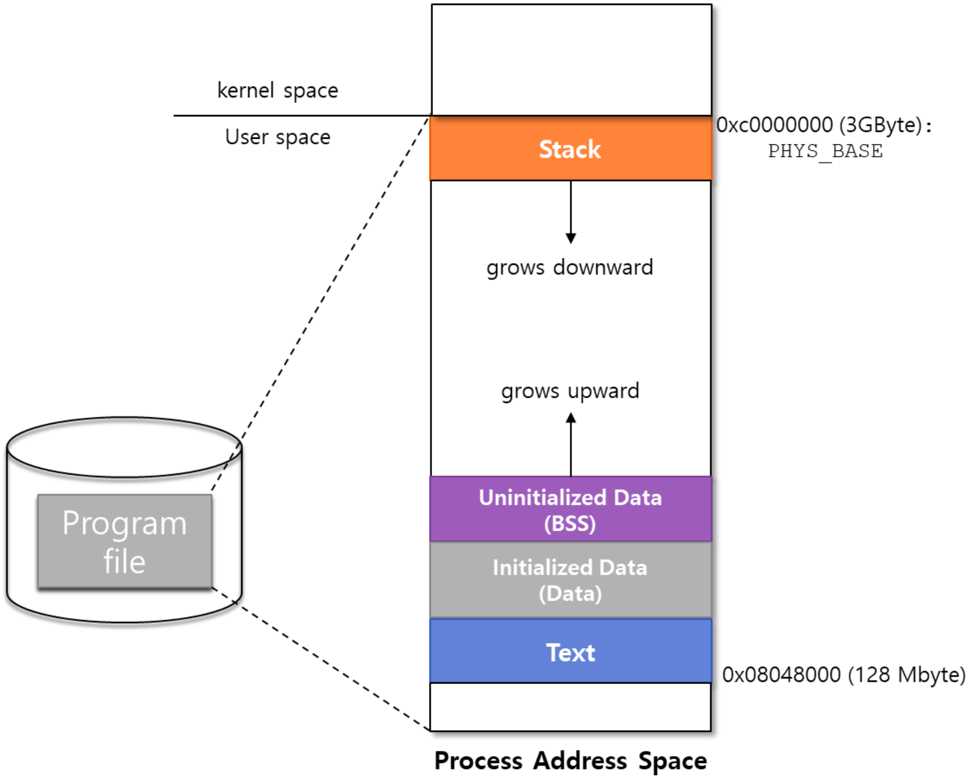
Background

Background

- **To run a program,**

1. Read the executable file from the disk
   - Reads the executable file of the program to be run from disk. This uses features of the file system to ensure that the file is correctly loaded from the disk into memory.
2. Allocate memory for the program to run
   - At this stage, Virtual memory space for code, data, stack, etc., defined in the executable file, is set up.
3. Pass the parameters to the program
   - Before entering the user mode, the operating system places the parameters (arguments) that the program needs on the user stack.
   - User stack is an area of memory for operations such as calling functions, storing local variables, and passing parameters while the program is running.
4. Context switch to the user program
   - The operating system switches context from kernel mode to user mode.
   - The operating system waits until the program is terminated. When the program finishes executing, the operating system reclaims the resources it allocated and handles the program's termination status.

- **Pintos VM layout**
  - Pintos virtual memory (VM) is divided into two main parts: user space and kernel space.
  - User space is where a program's code and data are stored.
  - Starting at the address just below PHYS_BASE, the user stack is located, and this space defines the limit to which the stack can grow.



Background

- **Running a program in PintOS**

Calling "process_execute"

```
static void run_task(char ** argv)
{
...
process_wait(process_excute(argv));
...
}
```

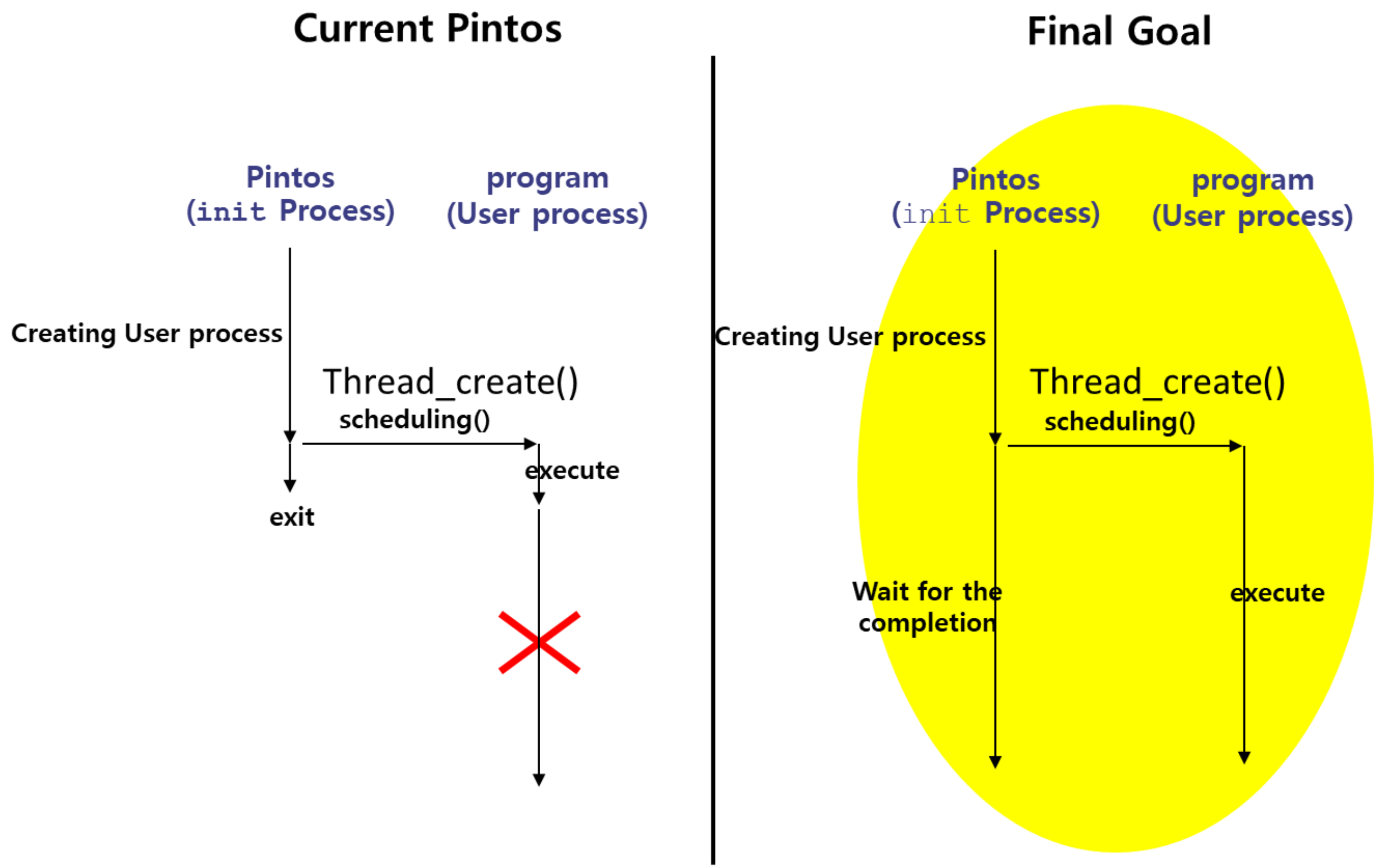
Create thread and start running a program

```
tid_t process_execute (const char
                         *file_name)
{
 ...
 tid = thread_create (..,start_process,.);
 ...
 return tid;
}
```

```
int process_wait (tid_t child_tid UNUSED)
{
    return -1;
}
```

The OS quits without waiting for the process to finish!!!

- **Executing a program**



1 Argument Parsing

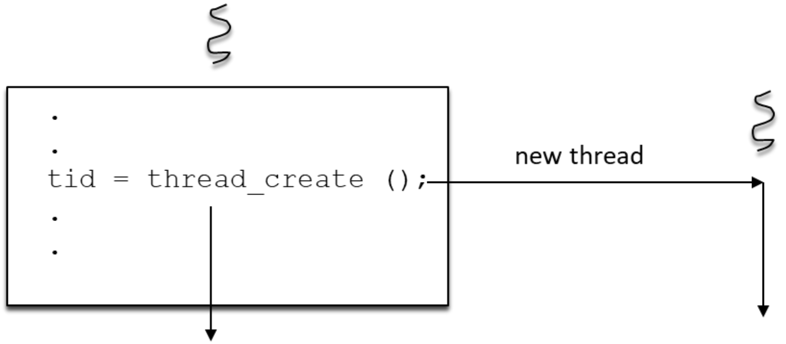
Argument Parsing

- **Current pintos**
  - Currently, pintos doesn't distinguish between programs and arguments
    - Ex) If you enter “bin/ls –l foo bar“ on the command line, Pintos recognizes "bin/ls -l foo bar" as a single program name.
  - The function process_execute() loads and executes process into memory so that the user can execute the command entered.
  - Although the program is stored as a string in file_name, process_execute() currently does not provide argument passing for the new process.

- **Main goal**
  - You need to develop a feature to parse the command line string, store it on the stack, and pass arguments to the program.
  - Instead of simply passing the program file name as an argument in process_execute(), you should modify it to parse words each time a space is encountered.
  - In this case, the first word is the program name, and the second and third words are the first and second arguments, respectively.

- **process_execute(const char *file_name) :**

  1. **File name parsing:**
     - Copy the `file_name` and separate the string based on the first space, extracting only the name of the program to be executed using strtok_r() function.
     - This operation determines the actual executable file name to be passed when creating a thread.
     - When parsing the string from `file_name` using the strtok_r() function with a space (" ") as the delimiter, the first token returns the string before the space.
     - ex) If the command "bin/ls -l foo bar" is entered, "bin/ls" will be returned.
  2. **Create a new thread**
     - Create a new thread using the thread_create(`file_name, PRI_DEFAULT, start_process, fn_copy`) function with the parsed file name. If thread creation fails, release the allocated pages.



Argument Parsing

- **thread_create (const char *name, int priority,thread_func *function, void *aux)**
  - It passes the name of the file it wants to execute.
  - It creates a thread with the executing function of `start_process`.
  - Allocating 4KB single page for kernel space. it initialize the thread structure.
  - Allocate the kernel stack. Kernel stack contains the address of the function need to be executed.
  - The parameters received by this function are stored in the kernel stack.

  - **Parameters**
    - name : the variable has a value of the first token of parsed string, pend it as the name of the new process
      - Ex) if you enter the command '/bin/ls –l foo bar' , then name = '/bin/ls'
    - function : function name (start_process)
    - aux : The arguments received when the function is executed.
      - Ex) if you enter the command '/bin/ls –l foo bar' , then aux = '/bin/ls –l foo bar'

Argument Parsing

- **Static void start_process (void *file_name_)**
  - This is the function used to start the user process, which takes the name of the program to run and arguments via the `file_name_` argument.
  - We'll use the command "/bin/ls -l foo bar" as an example to illustrate what happens in each step.

    1. Parse the arguments
       - Using the strtok_r() function, parse the `file_name_` given, splitting the arguments based on spaces (" "), and store them in the `argv` array. `argc` represents the number of parsed arguments.
       - In this case, argv[0] stores "/bin/ls", argv[1] stores "-l", argv[2] stores "foo", argv[3] stores "bar", and argc is 4.
    2. Initialize the interrupt frame (`struct intr_frame if_`)
    3. Load the executable file
       - Call the load() function to load the first token of `file_name` (i.e., "bin/ls").
       - This function loads the executable file into memory and initializes `if_.eip` (instruction pointer) and `if_.esp` (stack pointer).
       - If loaded successfully, use the argument_stack(`argv, argc, &if_.esp`) function to push the arguments stored in `argv` onto the stack. This sets up the stack so that when the user program starts, it can start with the appropriate arguments.

Argument Parsing

- **Static void start_process (void *file_name_)**

  4. Pass the load status
     - If the load fails, set `load_status` to -1. If it succeeds, set it to 1.
     - Notify the parent thread of the load status after acquiring the `lock_child` lock. Call `cond_signal` to signal the parent thread if it is waiting.
  5. Handle on load failure
     - If the load fails, free the used page. These are the pages we allocated for `file_name` and `argv`, and call thread_exit() to exit the current thread.
  6. Execute the user process
     - Use the `asm volatile` assembly block to set the stack pointer `%esp` to the address of the interrupt frame, and call `intr_exit` to execute the user process.
     - This is the process of switching context to the user program.

Argument Parsing

- **Static void start_process (void *file_name_)**

```
static void start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;
  ...
  success = load (file_name, &if_.eip, &if_.esp);
  if (!success)
    thread_exit ();
  /* Start the user process */
  asm volatile ("movl %0, %%esp; jmp
intr_exit" : : "g" (&if_) : "memory");
}
```

```
bool load (const char *file_name, void
(**eip) (void), void **esp)
{
  ...
  struct file *file = NULL;
  ...
  file = filesys_open (file_name);
  ...
  /* Set up stack. */
  if (!setup_stack (esp))
  ...
  success = true;
  return success;
}
```

```
void thread_exit (void)
{
  ...
  process_exit ();
  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();
}
```

Argument Parsing

- **void argument_stack(const char* argv[], int argc, void **esp)**
  - Use this function to put arguments on the stack in the correct format when the user program starts.
  - `argc` holds the number of arguments, and the `argv` array holds the values of the arguments. `esp` represents the pointer of the stack pointer
  - Here, argv[0] contains "/bin/ls", argv[1] contains "-l", argv[2] contains "foo", argv[3] contains "bar", and argc is 4.

1. Copy the arguments to the stack
   - Copy each argument to the stack use memcpy(), and store the starting addresses of the argument strings in the `argv_addr` array.
   - Because the stack grows from higher to lower address, each argument is copied onto the stack from the end.
2. Word alignment
   - Align `esp` to a multiple of 4, which is a 32-bit word boundary.
   - This is because Pintos is a 32-bit architecture: it uses 32-bit addresses, and the basic unit of data, the "word," is 4 bytes in size.
   - By performing word alignment like this, the CPU can fetch the data in a single memory access.

Argument Parsing

- **void argument_stack(const char* argv[], int argc, void **esp)**
  3. Add a NULL pointer
     - Decrease the stack pointer `esp` by 4 bytes, then store 0 in that space to add a NULL pointer indicating the end of the argument list.
  4. Set the argument addresses on the stack
     - Push the address of each `argv` argument stored in the `argv_addr` array onto the stack in reverse order. This creates an array of `argv` on the stack.
  5. Push the address of the `argv` onto the stack
     - Decrease `esp` by 4 bytes, then store a pointer to the first element of `argv` (i.e., the address of the first argument) on the stack.
  6. Push the `argc` onto the stack
     - Decrease `esp` by 4 bytes, then push the value of `argc` (the number of arguments) onto the stack.
  7. Push the return address onto the stack
     - Decrease `esp` by 4 bytes, then store 0. Typically, this space holds the return address of a function, but here, it stores 0 as it marks the starting point of the program.
  - After these steps, the `esp` will be the stack pointer that your program will use when it starts, and the stack will contain the program's arguments and all the information needed to start the program.

- **User stack layout in function call**

```
%bin/ls -l foo bar
```

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar₩0'	char[4]
0xbffffff8	argv[2][...]	'foo₩0'	char[4]
0xbffffff5	argv[1][...]	'-l₩0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls₩0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0 (fake address)	void (*) ()

grows ↓

stack top → 0xbfffffcc

- char[4], char[4], char[3], char[8]: Argument(string): 19 B
- uint8_t: 1Byte padding
- char * (×5): Argument's address
- char **, int: main(int argc , char **argv)
- void (*) () → fake address(0)

Argument Parsing

- **You need to add/modify ..**

  userprog/process.*

  - Modify tid_t process_execute(`const char *file_name`) (userprog/process.c)
  - Modify start_process(`void *file_name_`) (userprog/process.c)
  - Add static void argument_stack(`const char* argv[], int argc, void **esp`) (userprog/process.c)

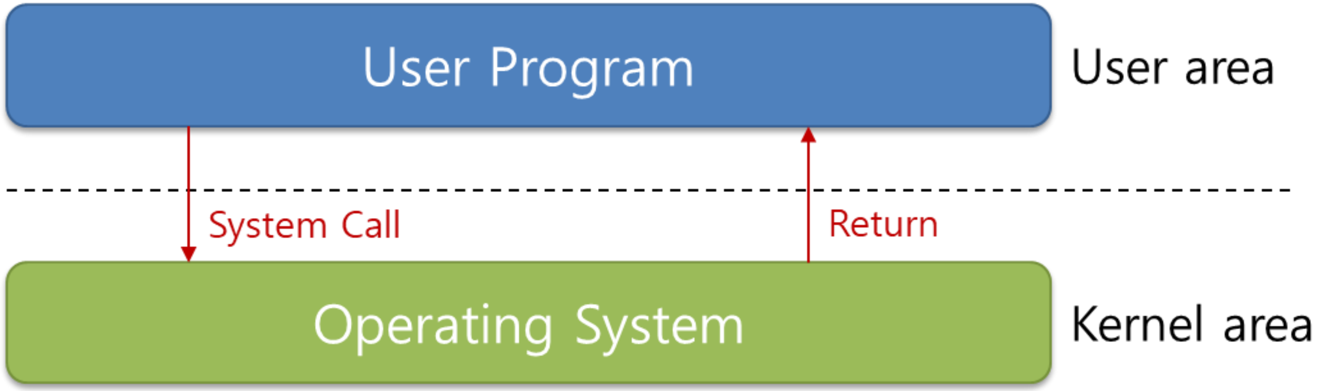
2

System call

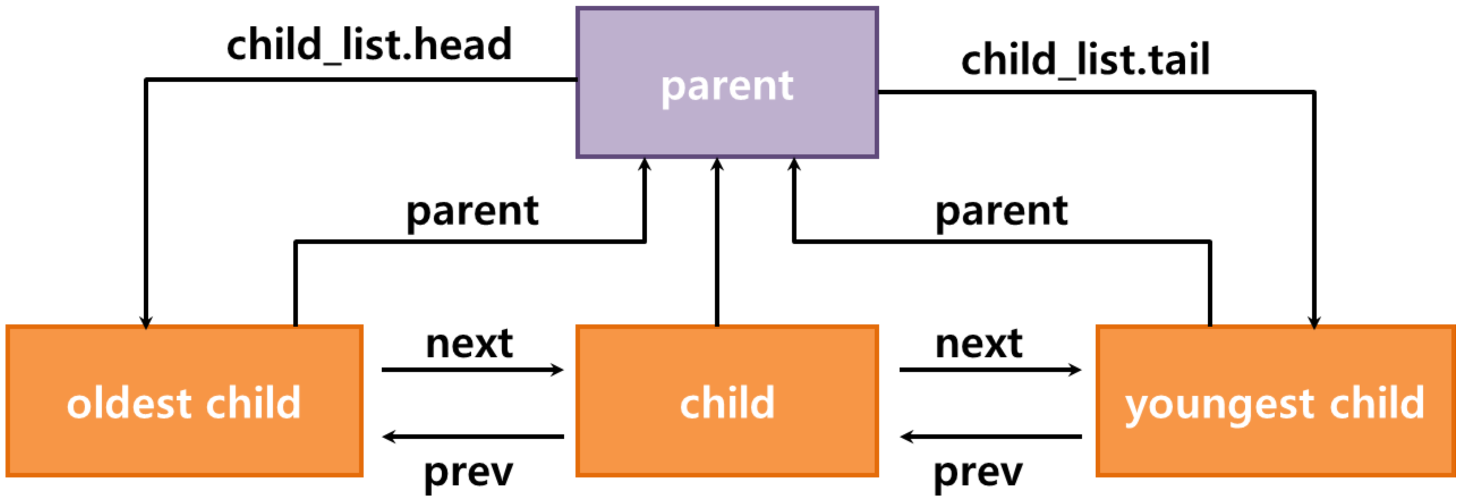
System call

- **Main goal**
  - Current Pintos doesn't have implemented system call handlers, so system calls are not processed.
  - You should ..
    - Implement system call handler and system calls
    - Add system calls to provide services to users in system call handler
      - Process related system call: halt, exit, exec, wait

- **System call**
  - Programming interface for services provided by the operating system
  - Allow user mode programs to use kernel features
  - System calls run on kernel mode and return to user mode
  - Key point of system call is that priority of execution mode is raised to the special mode as hardware interrupts are generated to call system call



- **Process hierarchy**
  - Augment the existing process with the process hierarchy.
  - When implementing system calls, it's important to consider the relationship between parent and child processes.

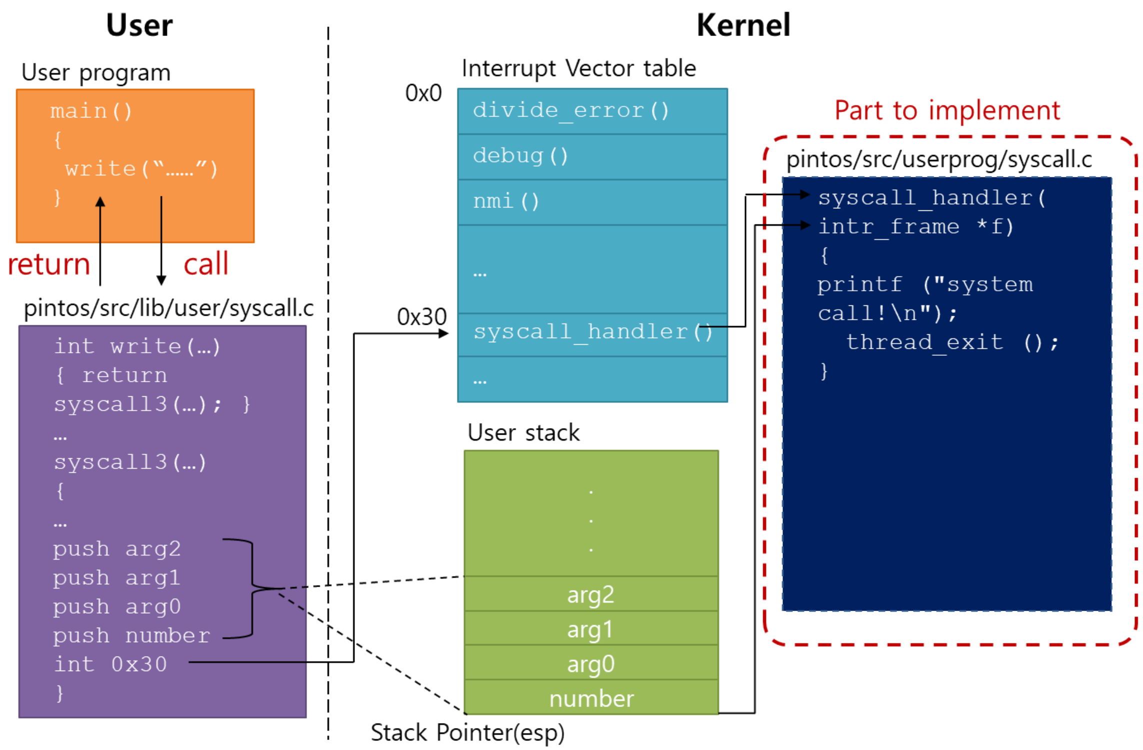


- **Process hierarchy**
  - To represent process hierarchy, the thread structure needs to be modified. (thread.h)

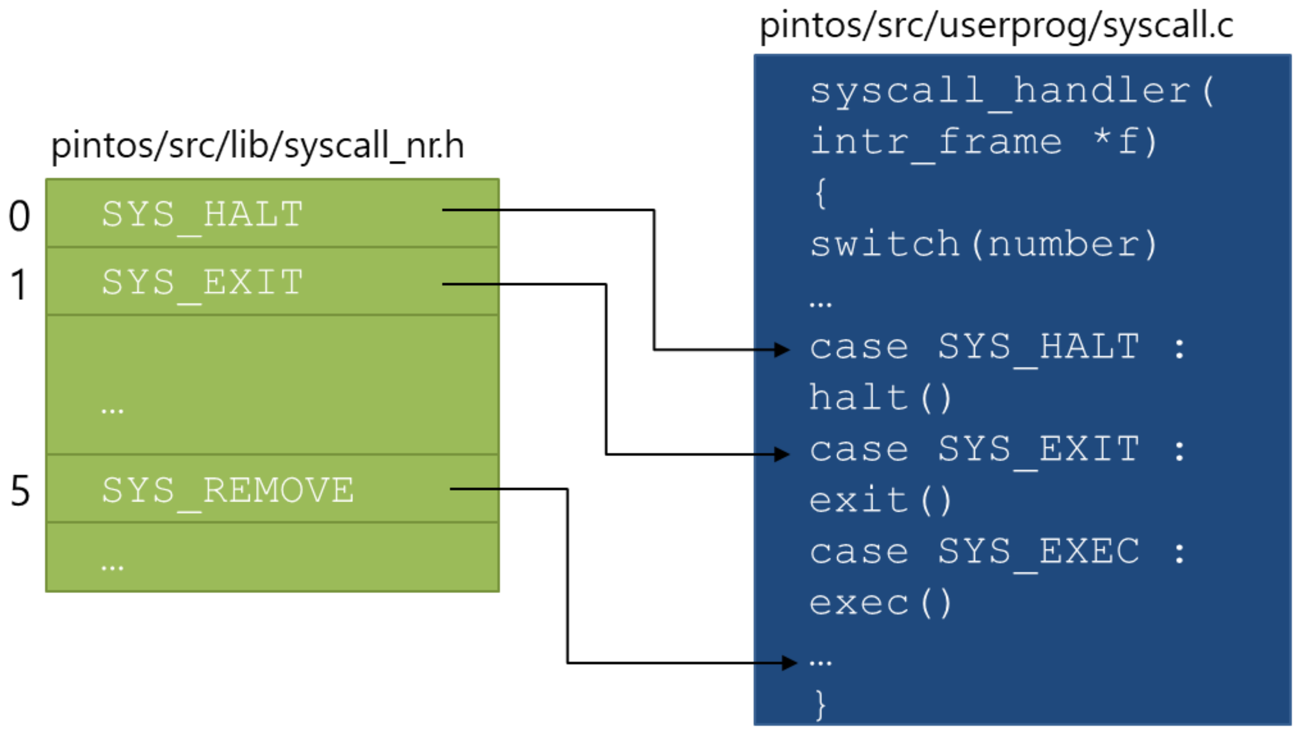
```
#ifdef USERPROG
   /* parent thread id */
   tid_t parent_id;
   /* signal to indicate the child's executable-loading status:
    *  - 0: has not been loaded
    *  - -1: load failed
    *  - 1: load success*/
   int child_load_status;
   /* monitor used to wait the child, owned by wait-syscall and waiting
     for child to load executable */
   struct lock lock_child;
   struct condition cond_child;
   /* list of children, which should be a list of struct child_status */
   struct list children;
   /* file struct represents the execuatable of the current thread */
   struct file *exec_file;
#endif
```

System call execution process example

1. A user program initiates execution. Within this program, the write() function is calling.
2. The write() function acts as a system call, placing its arguments on the user stack before transitioning into kernel mode. At this point, the stack pointer references the area where the arguments are stored.
3. The interrupt vector table maps each address to the corresponding type of interrupt that should be executed. The address 0x30 is mapped to system call handler, thus calling the syscall_handler() function.



- **System call handler**
  - Call the system call from the system call handler using the system call number.
    - The system call number is defined in pintos/src/lib/syscall_nr.h



System call

- **static void syscall_handler(struct intr_frame *f)**
  - Make system call handler call system call using system call number

1. Stack pointer validation
   - Check validation of the pointers in the parameter list using is_valid_ptr() function.
     - These pointers must point to user area, not kernel area.
     - If these pointers don’t point the valid address, it is page fault
2. Extract system call number
   - Obtains the system call number from the value `*esp` at the top of the stack.
3. System call processing
   - call the appropriate processing function based on the system call number.
   - For each system call, extract the arguments using the stack pointer `esp` and pass them to the system call function.
   - For example, for SYS_EXIT, pass `*(esp + 1)` as an argument to the exit function.
4. Return the result of the system call
   - System calls such as halt, exec, wait, create, save return value of system call at eax register.

System call

- **Bool is_valid_ptr (const void *usr_ptr)**
  - Checks if the pointer in the given user space (`usr_ptr`) is valid.
  - This function verifies the validity of a pointer by ensuring it is non-NULL, within the user address space, and corresponds to allocated memory in process address space.

1. Obtain current thread
   - Call thread_current() to get a pointer to the currently running thread.
2. Validate the pointer
   - Check if `usr_ptr` is not NULL. Since a NULL pointer is not a valid memory reference, the function returns false if it is NULL.
   - Call is_user_vaddr(`usr_ptr`) to verify if `usr_ptr` is within the virtual address space of user mode. This function checks to see if the address is within the range accessible to user programs.
3. Check page directory
   - Call pagedir_get_page(`cur->pagedir, usr_ptr`) to look up the page corresponding to `usr_ptr` in the current thread's page directory.
     - The pagedir_get_page() function finds the page table entry corresponding to the given virtual address, and verifies that it is actually allocated in memory.
   - It returns true if the given pointer points to a valid memory address; otherwise, it returns false.

System call

- **int wait (pid_t pid)**
    - Call process_wait(`pid`) function
- **int process_wait(tid_t child_tid)**
    - This function makes the calling thread wait until a specific child process exits. It returns the exit status of the given child process and manages its state.

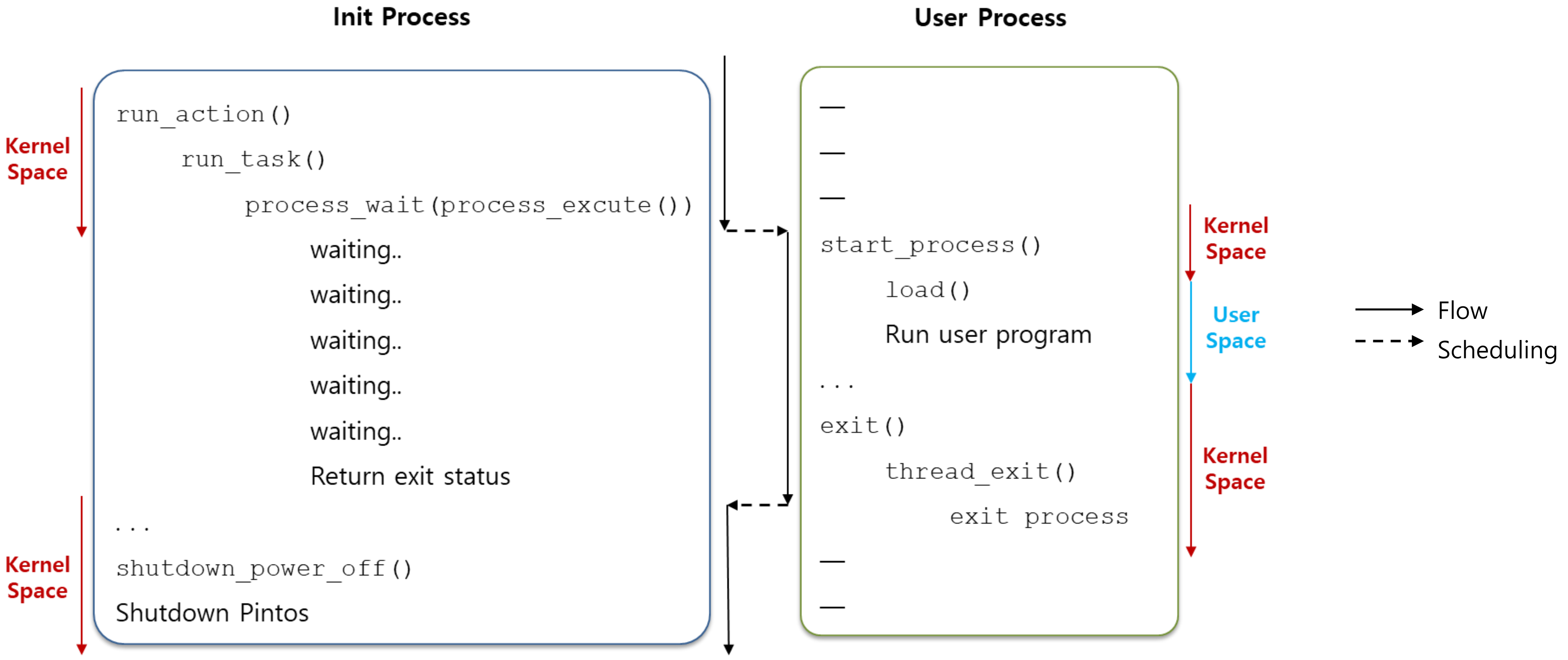
1. Input validation
    - Check if `child_tid` is a valid thread ID. If it's invalid, return an error code immediately.
2. Traversal child thread list
    - From the list of child threads of the current thread, find the thread that matches the `child_tid` given as an argument to the function.
3. Check child thread state
    - If it does not find a matching child thread, set `status` to -1 to indicate that the load failed.
    - If a matching child thread is found, the current thread acquires a lock named `lock_child`.
4. Waiting with condition variable
    - Call thread_get_by_id(`child_tid`) to check if the child thread still exists. If it does exist, make the current thread wait on the `cond_child` condition variable.
    - When the child thread exits, it receives a signal and stop waiting on the condition variable

- **Int process_wait(tid_t child_tid)**
  - This function makes the calling thread wait until a specific child process exits. It returns the exit status of the given child process and manages its state.

5. Check the exit status of the child thread
   - Check if the child thread has exited normally, and if waiting has already been processed.
   - If the child thread has exited normally and waiting has not been processed previously, save the child thread's exit status in status and set `has_been_waited` to true to prevent other threads from waiting on this child again.
6. Resource cleanup and function exit
   - Release the `lock_child` lock acquired by the current thread.
   - Return the `status` value. This value represents the exit status of the child thread or -1 if the child thread was not found.

- Note: `list_head` doesn't point to the first data node but rather to a "head sentinel" of the list, i.e. the sentinel node that precedes the actual data. The first data element in the list would be the next element after list_head(`&cur->children`).

- **Flow of user program execution**



- **pid_t exec (const char *cmd_line)**
  - System call to create child process and execute program corresponds to cmd_line on it

1. Variable declaration and initialization
   - Declare the thread ID of the newly created process (`tid`) and , the pointer to the currently running thread, that is, the thread that calls this exec function (`cur`).
2. User pointer validation
   - Validates that the passed `cmd_line` pointer is a valid user address space pointer. If it is invalid, call the exit() function, to exit the current thread.
3. Initialize `child_load_status` of current thread
   - Sets the `child_load_status` of the current thread to 0. This variable is used to track the load status of newly created child processes.
4. Execute the new process
   - Calls the process_execute() function to create a new process with the command specified in `cmd_line`. This function returns the `tid` of the new process.

System call

- **pid_t exec (const char *cmd_line)**
  - System call to create child process and execute program corresponds to cmd_line on it

5. Acquire a lock for synchronization
   - Call lock_acquire() to acquire the `lock_child` lock. This prevents multiple threads from accessing the child's load state at the same time.
6. Wait for the child process to finish loading
   - Use cond_wait() to suspend execution of the current thread until the newly created child process is fully loaded or fails to load.
   - Release `lock_child` lock
7. Check child process loading status
   - If the child process fails to load, set `tid` to -1 to indicate that the exec call failed.
8. Release lock and exit the function
   - Call lock_release() to release the lock.
   - If the new process was successfully created, return its `tid`; otherwise, return -1.

- **tid_t process_execute(const char *file_name)**
  - This function creates a new thread to run the new user program, and adds its management information to the list of children of the parent process, preserving the process hierarchy.
  - The thread's management information : the information stored in the `child_status` structure, which can include the thread's identifier, status, whether it is terminated, its exit status, and whether it is being waited on by the parent thread.

1. Initializing the child process status structure
   - If thread creation was successful (`tid != TID_ERROR`), get the current thread (`cur`).
   - Using calloc(`1, sizeof *child`), allocate a new memory area to store the process's child management information (`child_status`), and store its memory address in a pointer variable (`child`).
   - Set the fields of the child structure: Set child_id to the ID of the newly created thread (`tid`), and set `is_exit_called` and `has_been_waited` to false, to track the child process's exit status and whether it's been waited for.
2. Add the new child to the list of children
   - Use the list_push_back() function to add the state of the newly created child to the list of children of the current thread.
3. Cleanup and return
   - Release `file_name_` pages that are no longer needed
   - Return the ID of the newly created thread (`tid`) .

- **void exit (int status)**
  - Terminate the current user program, returning status to the kernel.
  - If the process' parent waits for it, this is the status that will be returned.

1. Obtain current thread and parent thread information
   - Find the parent thread (`parent`) using the `parent_id` of the current thread.
2. Update the status of the current thread
   - If the parent thread exists, it searches the children list of parent thread to find the current thread.
   - It checks if the thread ID (`tid`) of the current thread (`cur`) matches the ID of a specific child in the parent thread's list of children (`child_id`).
   - If it does match, it means that the current thread is the child process that you want to exit.
3. Acquire a lock for synchronization
   - Call lock_acquire(`&parent->lock_child`) to acquire a lock on the parent thread's (`parent`) list of children so that it can safely modify them. This prevents other threads from modifying the same child list at the same time, ensuring data integrity.

- **void exit (int status)**
  - Terminate the current user program, returning status to the kernel.
  - If the process' parent waits for it, this is the status that will be returned.

4. Update the exit status of the child process:
   - Set `child->is_exit_called` to true to indicate that the child has exited.
   - Record the exit status of the child by storing the status value in `child->child_exit_status`.
   - stores the exit status of the child process so that the parent process can retrieve it. The `status` parameter represents the exit status code passed to the exit system call.
5. Lock release
   - Unlock the parent thread to allow other threads to access or modify the parent's list of children.
6. Thread termination
   - Terminate the current thread by calling the thread_exit() function. This function terminates the thread, and performs tasks to clean up any associated resources.

System call

- **void halt(void)**
  - Shutdown pintos
  - Use void shutdown_power_off(void)
    - shutdown_power_off() : Powers down the machine we're running on, as long as we're running on QEMU.

System call

- **You need to add/modify ..**

Threads/thread.*

- Modify thread structure (threads/thread.h)

Userprog/syscall.*

- Modify void syscall_init(`void`) (userprog/syscall.c)
- Modify static void syscall_handler(`struct intr_frame *f`) (userprog/syscall.c)
- Add bool is_valid_ptr (`const void *usr_ptr`) (userprog/syscall.c)
- Add int wait (`pid_t pid`) (userprog/syscall.c)
- Add void exit (`int status`) (userprog/syscall.c)
- Add pid_t exec (`const char *cmd_line`) (userprog/syscall.c)
- Add void halt(`void`) (userprog/syscall.c)

Userprog/process.*

- Modify tid_t process_execute(`const char *file_name`) (userprog/process.c)
- Modify int process_wait(`tid_t child_tid`) (userprog/process.c)

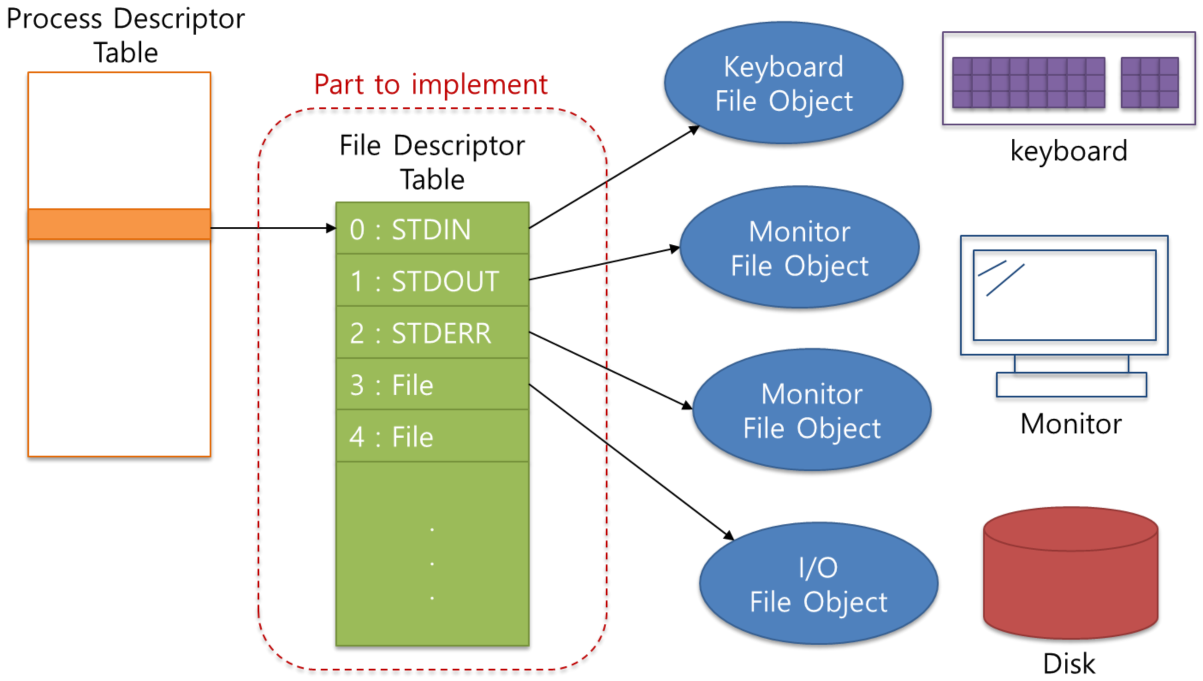
3 File manipulation

File manipulation

- **Main goal**
  - Add system calls to provide services to users in system call handler
  - File related system call: create, remove, open, filesize, read, write, seek, tell, close

File Descriptor in PintOS

- Access to File by using File Descriptor
- STDIN :
  - An input stream that transfers data to a program, typically receives input data from a keyboard.
- STDOUT :
  - An output stream that communicates the results of a program to a user or other system. It usually outputs text to the console or terminal.
- STDERR :
  - A separate output stream for carrying error messages and logs. This stream also outputs text to the console or terminal, and operates independently of the standard output



- **Allocate file descript table**
  - Define FDT as a structure.
  - Allocate FDT at kernel memory area, and add the associated pointer to at the thread structure.
  - Each process has its own file descriptor table (Maximum size: 64 entry).
  - File descriptor table is an array of pointer to struct file.
  - FD is index of the file descriptor table, and it is allocated sequentially.
    - File descriptors are normally expressed as integers.
    - Each file descriptor acts as a handler for real files or other types of input/output resources (e.g., pipes, network connections, etc.)
    - STDIN is file descriptor 0, STDOUT is file descriptor 1, and STDERR is file descriptor 2

- **File descriptor should include the following elements:**
    - int fd_num;
        - This field stores the file descriptor number. The file descriptor number is an integer value used to uniquely identify each file within the process. This value is assigned when the process opens a file
    - tid_t owner;
        - The owner stores the ID of the thread that owns this file descriptor, which indicates that the file descriptor belongs to a specific thread or process and can be used to manage its permissions or life cycles..
    - struct file *file_struct;
        - file_struct stores pointers to real file objects. This pointer is an important reference for accessing file data and metadata within the file system. File objects can contain information such as the file's current location (offset), access mode, and reference count, and are used to perform actual read/write operations on files.
    - struct list_elem elem;
        - Elem is an element of the type structure list_elem that allows this structure to be included in the list. It is used to include file descriptors in the process's file descriptor list, which helps the operating system manage all open files in the process efficiently. For example, you can easily find and close all open files at the end of the process.

File manipulation

- **File Descriptor Table**
  - When the thread is created,
    - Allocate File Descriptor table.
    - Initialize pointer to file descriptor table.
    - Reserve fd0, fd1 for stdin and stdout.
  - When thread is terminated,
    - Close all files.
    - Deallocate the file descriptor table.
  - Use a list of open files
    - Define a list of open files in syscall.c (struct list open_files)
    - Initialize the list on syscall_init()
    - It represents all the files open by the user process through syscalls
  - Use global lock to avoid race condition on file,
    - Define a global lock in syscall.c (struct lock fs_lock).
    - Initialize the lock on syscall_init()
    - Protect filesystem related code by global lock

File manipulation

- **bool create (const char *file_name, unsigned size)**
  - Create a file using given file name and size

1. File name pointer validation
   - Verify that the `file_name` pointer points to a valid memory address via if `!is_valid_ptr (file_name)`.
   - This is a mandatory check for security and stability, and if the pointer is not valid, the function calls exit(-1) to terminate the process.
2. File system lock acquirement
   - Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.
3. Create the file
   - status = filesys_create(`file_name, size`); invokes the filesys_create function to perform the file creation operation.
   - The fileys_create function returns true if the file creation is successful or false if it fails. This return value indicates whether the file creation is successful or not and is stored in the status variable.
4. Unlock File system lock
   - lock_release(`&fs_lock`); unlock previously acquired file system locks by calling, which allows other threads or processes to access the file system.
5. Return function results
   - Return the result of the file creation operation to the caller, which allows the called function or process to determine whether the file creation was successful.

- **bool remove (const char *file_name)**
  - Delete a file with the given name from the file system. This function takes a file name as an parameter and returns a boolean value based on whether the file was successfully deleted

1. File name pointer validation
   - Verify that the `file_name` pointer points to a valid memory address via if `!is_valid_ptr (file_name)`. This is a mandatory check for security and stability, and if the pointer is not valid, the function calls exit(-1) to terminate the process.
2. File system lock acquirement
   - Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.
3. Delete the file
   - Call the fileys_remove(`file_name`) function to delete the file. It returns true if the file is successfully deleted or false if it fails.
4. Release File system lock
   - lock_release(`&fs_lock`); unlock previously acquired file system locks by calling, which allows other threads or processes to access the file system.
5. Return function results
   - Return the result of the file creation operation to the caller, which allows the called function or process to determine whether the file creation was successful.

File manipulation

- **int open** (const char *file_name)
  - This function takes `file_name` as a parameter, opens the file, and returns the file descriptor number if the file is successfully opened.

1.File name pointer validation
   - Verify that the `file_name` pointer points to a valid memory address via if `!is_valid_ptr(file_name)`.
   - This is a mandatory check for security and stability, and if the pointer is not valid, the function calls exit(-1) to terminate the process.

2.File system lock acquirement
   - Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

3.Open the file
   - Call f = filesys_open(`file_name`); to open the file corresponding to `file_name` in the file system.
     - filesys_open() returns a file object pointer and returns NULL if the file cannot be opened.

File manipulation

- **int open (const char *file_name)**

4.File descriptor assignment and initialization

- If the file was opened successfully, assign a new `file_descriptor` structure to memory and initialize all fields to zero.
- Use the allocate_fd() function to assign a new file descriptor number.
  - allocate_fd() used to provide a unique identifier for each file, which allocates a new file descriptor number each time by returning an increased integer value for each call.
- Set the ID of the current thread to the owner of the file descriptor to track which thread opened which file.
- Store the file object pointer in the file descriptor structure for operating system can manage and manipulate the data and status of a particular file through the file descriptor.
  - `File object`: A structure that contains information about files within a file system. This structure typically includes metadata for a file (such as file size, permissions, creation and modification times, etc.), the current read/write location of the file (file pointer), and references to where the actual data is stored.
  - `File object pointer`: pointer to a file object for an open file
  - Add the new file descriptor to the system's open list of files using `list_push_back(&open_files, &fd->lem)`

5.Release File system lock and return the results

- Unlock `fs_lock`, which allows other threads to access the file system.
- Returns the newly assigned file descriptor number; returns -1 if the file could not be opened.

File manipulation

- **int filesize (int fd)**
  - **Return the size of the file open**

1. File system lock acquirement :
   - Use lock_acquire(&fs_lock); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

2. Search file descriptors:
   - Call get_open_file(fd); to find the file_descriptor structure corresponding to the file descriptor number (fd).

3. Look up the file size:
   - Verify that fd_struct indicate to a valid file descriptor.
   - If it is valid, call the file_length(fd_struct->file_struct) function to look up the length (size) of that file, and store the result in status.

4. Release File system lock :
   - Unlock fs_lock, which allows other threads to access the file system.

5. Return result:
   - The size of the file is returned if it was successful; -1 is returned if the file descriptor is invalid or the file's size could not be looked up for some other reason.

File manipulation

- **struct file_descriptor * get_open_**file (int fd):
  - A function that retrieves the file descriptor structure from a list of open files based on the file descriptor number.

1. It traverses the list of open files, checking to see if the file descriptor number of the current structure matches the number you want to retrieve.
2. If the number matches, it returns a pointer to the `fd_struct` so that the called function can use it.
3. If it does not find a matching file descriptor number after searching the entire list of open files, it returns NULL. This indicates that there are no open files with the requested file descriptor number.

File manipulation

- **Int read (int fd, void *buffer, unsigned size)**
  - Performs the function of reading data from a specified file descriptor and storing it in a buffer provided by the user.
  - Parameters
    - `int fd`: The file descriptor of the file to perform the read operation on.
    - `void *buffer`: The address of a memory buffer in user space to store the read data..
    - `unsigned size`: The maximum number of bytes of data to be read into the buffer..

1.File name pointer validation

- Verify that the `file_name` pointer points to a valid memory address via if `!is_valid_ptr(file_name)`. This is a mandatory check for security and stability, and if the pointer is not valid, the function calls exit(-1) to terminate the process.

2. File system lock acquirement

- Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

3.Check the standard output file descriptor

- If the file descriptor indicates standard output(`STDOUT_FILENO`), the read operation is invalid, so call lock_release(`&fs_lock`) and returning -1.
- The read operation is invalid because standard output is to pass the output of the program.

- **Int read (int fd, void *buffer, unsigned size)**

4.Read from Standard Input

- If the file descriptor represents a standard input (`STDIN_FILENO`) (when `fd == 0`), use the input_getc() function to read data from the keyboard and store it in the buffer. This process is repeated by the specified size.
  - input_getc(): read characters from a standard input and returns the read characters.
- After reading all the data, uslock the `fs_lock` and return the number of bytes actually read.

5.Read from a regular file

- If `fd` is not 0 (not standard input/output), call get_open_file(`fd`) to get the file_descriptor structure of the open file corresponding to fd.
- If descriptor structure is valid, call file_read(`fd_struct->file_struct, buffer, size`) to read data from the actual file; the number of bytes read is stored in status.

6.Release File system lock and return function results

- Unlock `fs_lock`, which allows other threads to access the file system.
- Return the value of status. This value is either the number of bytes actually read, or -1if the file descriptor is invalid.

File manipulation

- **Int write (int fd, const void *buffer, unsigned size)**
  - write data through a file descriptor.
  - Parameters:
    - `int fd`: The file descriptor of the file to perform the write operation on.
    - `void *buffer`: containing the data to write,
    - `unsigned size`: size of the data you want to write

1.File name pointer validation

- Verify that the `file_name` pointer points to a valid memory address via if `!is_valid_ptr(file_name)`. This is a mandatory check for security and stability, and if the pointer is not valid, the function calls exit(-1) to terminate the process.

2. File system lock acquirement

- Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

3.Check standard input file descriptor

- If the file descriptor `fd` indicates standard input (`STDIN_FILENO`), the write operation is invalid , release files system lock and returns -1.

File manipulation

- **Int write (int fd, const void *buffer, unsigned size)**

4.Write data in standard output:

- If the file descriptor `fd` represents the standard output (`STDOUT_FILENO`), call putbuf(`buffer, size`); and immediately output size bytes of data from the buffer to the standard output stream. In this case, it returns the size of the size bytes after the operation is successful

5.Write data to a regular file:

- If not standard I/O, call get_open_file(`fd`) to get the file_descriptor structure of the open file corresponding to fd
- If `fd_struct` is valid, call status = file_write (`fd_struct->file_struct, buffer, size`) to write the amount of data in the buffer to size bytes in the actual file, and store the number of bytes written in the status depending on the success of the write operation.

6.Release File system lock and return function results:

- Unlock `fs_lock`, which allows other threads to access the file system.
- Return the value of `status`. This value is either the number of bytes actually write, or -1if the file descriptor is invalid.

File manipulation

- **void seek (int fd, unsigned position)**
  - Allows you to jump directly to the desired data location within a file to start reading or writing operations.

1. File system lock acquirement
   - Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

2. Search for file descriptors
   - Retrieves the `file_descriptor` structure corresponding to the given file descriptor `fd`, which contains all information related to the file, especially the file object.

3. Adjust file pointer location
   - Verify that `fd_struct` is valid. If so, call file_seek (`fd_struct->file_struct; position`); to move the file pointer to the position specified within the file.
   - The file_seek() function is responsible for moving the pointer from the beginning of the file to a position byte away.

4. Release File system lock
   - Unlock `fs_lock`, which allows other threads to access the file system.

File manipulation

- **unsigned tell (int fd)**
  - Return the position of the next byte to be read or written in open file `fd`

1. File system lock acquirement
   - Use lock_acquire(`&fs_lock`); for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

2. Search for file descriptors
   - Retrieves the `file_descriptor` structure corresponding to the given file descriptor `fd`, which contains all information related to the file, especially the file object.

3. Returns the current file pointer location
   - Verify that fd_struct is valid. If so, call file_tell(`fd_struct->file_struct`) to return the current pointer location of the file object.
   - The file_tell() function returns the number of bytes from the beginning of the file to the current pointer

4. Release File system lock
   - Unlock `fs_lock`, which allows other threads to access the file system.

5. Return reult
   - Returns the value indicating the current location of the file. Returns the initial value if `fd_struct` is invalid and cannot get the location information of the file

- **void close (int fd)**
  - Close the file corresponding to the given file descriptor `fd` and release the associated resources.

1. File system lock acquirement
   - Use lock_acquire(`&fs_lock`) for get a global lock for the file system by calling, which is necessary to prevent conflicts of work if there are other threads or processes that simultaneously access the file system.

2. Search for file descriptors
   - Retrieves the `file_descriptor` structure corresponding to the given file descriptor `fd`, which contains all information related to the file, especially the file object.

3. Check file descriptor ownership and validity
   - Verify that the file descriptor is valid and that the current thread is the owner of the file descriptor.
     - Current thread is the owner, call close_open_file(fd) to actually close the file.
     - Current thread is not the owner, you do not close the file

4. Release File system lock
   - Unlock `fs_lock`, which allows other threads to access the file system.

- **void close_open_file (int fd)**
  - Remove file descriptors from the list and close files.

1. Traverse the `open_files` list, getting the file descriptor information for each element.
2. Check if the file descriptor number matches the entered `fd`
3. File descriptor number matches the entered `fd`, call list_remove(e) to remove that element from the list
4. Call file_close(`fd_struct->file_struct`) to close the file object
5. Call free(`fd_struct`) to free memory and exit the function

File manipulation

- **Do not allow the file to be modified when it is opened for execution.**
  - Approach
    - When the file is loaded for execution, call file_deny_write().
    - When the file finishes execution, call file_allow_write().
  - Need to Modify
    - bool load (const char *file_name, void (**eip) (void), void **esp) (userprog/process.c)
      - Call file_deny_write() when program file is opened.
      - file_deny_write(file): prevents other processes or threads from changing the contents of the file while it is running. This ensures the stability of the system by preventing the code of the running program from changing
    - void process_exit (void)(userprog/process.c)

- **void process_exit** (void)
  - Performs resource release, status update, and cleanup operations at the end of the process.

1.Get current thread information

- struct thread *cur = thread_current();Get the information of the currently running thread. Use this information to handle operations related to the resources of that thread.

2.Destroy the page directory

- Gets the page directory address of the current thread.
- If the page directory(pd) is not NULL, it first disconnects the current thread's page directory.
- Call pagedir_activate(NULL);to switch to the kernel-only pagedirectory.
- Destroy the process's pagedir directory using pagedir_destroy(pd). This prevents memory leaks and returns the used pages to the system.

3.Unlisting child processes

- Traverse the list of child processes, freeing the `child_status` structure of each child. Along the way, each element is removed from the list with list_remove(e);and memory is freed with free(child).

File manipulation

- **void process_exit** (void)
  - Performs resource release, status update, and cleanup operations at the end of the process.

4.Unlock write protection

- Check for executable files that are connected to the current thread.
- Call file_allow_write(cur->exec_file);to unlock write protection on the file. This allows the file to become writeable again as a normal file.

5.Close files owned by the current thread

- Call close_file_by_owner(cur->tid); to close all files opened by the current thread.
- This function retrieves all file descriptors owned by that thread, closes the files, and releases sources.

6.Sending a termination signal to the parent thread

- If a parent thread exists, the current thread acquires the `lock_child` of the parent thread. Acquiring this lock prevents concurrency issues before dealing with the parent thread's child list or state information.
- Inspect the `child_load_status` of the parent thread and set it to -1if the load status is still 0. This indicates a load has failed or terminated.
- Send a signal to the parent thread that is waiting, and then release the lock.

File manipulation

- **You need to add/modify ..**

  userprog/syscall.*

  - Define struct `file_descriptor` (userprog/syscall.c)
  - Add bool create (`const char *file_name, unsigned size`) (userprog/syscall.c)
  - Add bool remove (`const char *file_name`) (userprog/syscall.c)
  - Add int open (`const char *file_name`) (userprog/syscall.c)
  - Add int filesize (`int fd`) (userprog/syscall.c)
  - Add int read (`int fd, void *buffer, unsigned size`) (userprog/syscall.c)
  - Add int write (`int fd, const void *buffer, unsigned size`) (userprog/syscall.c)
  - Add void seek (`int fd, unsigned position`) (userprog/syscall.c)
  - Add unsigned tell (`int fd`) (userprog/syscall.c)
  - Add void close (`int fd`) (userprog/syscall.c)
  - Add struct file_descriptor *get_open_file (`int fd`) (userprog/syscall.c)
  - Add void close_open_file (`int fd`) (userprog/syscall.c)
  - Add bool is_valid_ptr (`const void *usr_ptr`) (userprog/syscall.c)
  - Add int allocate_fd () (userprog/syscall.c)
  - Add void close_file_by_owner (`tid_t tid`) (userprog/syscall.c)

File manipulation

- **You need to add/modify ..**

userprog/process.*

- Modify bool load (`const char *file_name, void (**eip) (void), void **esp`) (userprog/process.c)
- Modify void process_exit (`void`) (userprog/process.c)

File manipulation

- Result check
  - $ cd ~pintos/src/userprog
  - $ make clean
  - $ make
  - $ make check

```
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
```

```
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
```

```
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
```

- When successfully implementing, all 76 test cases pass

Reference

- Kaist Pintos Project2 Background, System calls and handlers, file manipulation
  - https://www.youtube.com/watch?v=RbsE0EQ9_dY&feature=youtu.be
  - https://www.youtube.com/watch?v=sBFJwVeAwEk
  - https://www.youtube.com/watch?v=SqMD8rbmEjY

Thank you