Project User Programs Design

Group 36

Name	Autograder Login	Email
Siddharth Shashi	student63	sshashi@berkeley.edu
Matthew Nguyen	student161	matthew.t.nguyen@berkeley.edu
Aman Doshi	student457	aman.doshi@berkeley.edu
Pritish Parmar	student251	pritishparmar@berkeley.edu

Argument Passing

Data Structures and Functions

In order to implement argument passing, we need the **setup_stack** function to take in an additional parameter that specifies the command line arguments being passed in.

```
static bool setup_stack(void** esp, const char* file_name);
```

Algorithms

<u>Overview</u>

Starting a new process requires a call to **pid_t process_execute(const char *file_name)**, where **file_name** is a command line string, e.g. "/bin/ls -l foo bar". **process_execute** creates a new thread, which calls the **start_process** function, which calls the **load** function, which calls the **setup_stack** function. We have modified **setup_stack** to receive **file_name** argument from **load**.

<u>Loading Executable</u>

Update references to file_name inside **load**, so only the file name (not arguments) are passed into the **filesys_open** call. Otherwise, executable cannot be opened.

<u>Stack Structure</u>

The user stack should be setup as follows (from top to bottom): command line strings, array of pointers to command line strings (**argv** array), **argv** (**&argv[0]**), **argc**.

<u>Pushing Arguments onto the Stack</u>

To push an argument onto the stack, we will use the predefined memcpy function:

```
void memcpy(void* dst_, const void* src_, size_t size);
```

More specifically, to push an argument onto the stack, decrement ***esp** by **arg_size** (# bytes needed to store the argument) and then call **memcpy(esp, &arg, arg_size)**.

Set Up Stack (done in **setup_stack** function)

To break **file_name** into individual argument strings, we can use the predefined **strtok_r** function. Upon getting an individual argument string with **strtok_r**, push it onto the stack by decrementing ***esp** and using **memcpy**. Then, increment **argc** (initialized to 0) and store the address of the argument string in an array called **arg_pointers** (declared with size 1024). To get rest of the individual argument strings, repeat the process above. Finally, set **arg_pointers[argc] = NULL**.

Add padding in the form of null bytes to ensure 4B alignment of stack pointer (for **argv** array).

```
while (stack pointer not 4B aligned) {
    <Push NULL Byte onto the Stack>
}
```

Then, memcpy **arg_pointers** array onto stack from index 0 to **argc**. Next, add padding in the form of null bytes to ensure 16B alignment when **argv** and **argc** are pushed onto the stack:

```
while ((stack pointer+sizeof(argv)+sizeof(argc)) not 16B aligned) {
    <Push NULL Byte onto the Stack>
}
```

Push **argv** (**&argv[0]**) onto the stack, then push **argc**. Push dummy "return address" (0x00000000) onto the stack.

Synchronization

We use **strtok_r**, which is "thread safe." Otherwise, no other synchronization schemes are needed, as all procedures are happening inside a single thread. No data is shared

Rationale

An alternative algorithm considered for retrieving the individual argument strings in **file_name** was to iterate through each character in **file_name** and appended each character to a **char *curr_word** until a “ ” found. When a “ ” is found, **curr_word** would be an individual argument string that we can push onto the stack. The algorithm that we decided instead is better because it uses **strtok_r**, which makes our code more readable and less prone to “one-off errors.” Furthermore, **strtok_r** can be optimized by the compiler. We also considered using **strtok** instead of **strtok_r**, but **strtok** is not thread safe. All the algorithms above run in linear time or less.

Process Control Syscalls

Data Structures and Functions

To implement the exec and wait syscall, every parent process must track information relating to their child processes. The following data structures track the relevant child process data. Additionally, they also handle file operation syscalls by storing the file descriptor table.

```
struct process {    // PCB struct, modified
  struct list child_info;    // A proc_info list
  struct proc_info *info;    // Reference own proc_info
  struct list fd_table;   // File descriptor table
  // ... Existing contents ...
}
struct proc_info {  // New struct storing process info
  pid_t pid;  // Identifier of proc_info struct
  bool load_status;  // Load status of the process
  struct semaphore load_sema;  // Synchronize load status
  int exit_status;  // Exit status of the process
  struct semaphore exit_sema;  // Synchronize exit status
  bool waited;  // Indicator if process has been waited on
  int ref_cnt;  // How many places proc_info is referenced
  struct lock ref_cnt_lock;  // Synchronize ref_cnt updates
```

```
  struct list_elem elem;  // elem of child_info Pintos list
}
```

In process.c, remove all references to the **temporary** semaphore for process waiting. Functionality replaced by the schemes described in the algorithm section.
The following helper functions will be used for validating pointers and executing syscalls. These functions will be declared inside the same file as the syscall_handler function (userprog/syscall.c).

```
static void valid_byte_pointer(uint8_t *p);  // 1 byte
static void valid_pointer(uint8_t *p, size_t size);
static char *valid_str_pointer(uint8_t *p);  // String
```

```
static void syscall_practice(struct intr_frame *f, int i);
static void syscall_halt(void);
static void syscall_exit(struct intr_frame *f, int exit_code);
static void syscall_exec(struct intr_frame *f, const char *cmd_line);
static void syscall_wait(struct intr_frame *f, pid_t child_pid);
```

Algorithms

Syscall Handler Function Structure

The following code shows a proposed structure to the syscall_handler function, with calls to the helper functions.

```
static void syscall_handler(struct intr_frame *f) {
  uint32_t *args = (uint32_t *) f->esp;
  valid_pointer(&args[0], sizeof(int));  // Memory check
  uint32_t syscall_type = args[0];  // Dereference
  switch (syscall_type) {
    case SYS_EXEC:
      valid_pointer(&args[1], sizeof(char *)); // Memory check
      char *cmd_line = (char *) args[1];  // Dereference
```

```
      char* s = valid_str_pointer(cmd_line);
      syscall_exec(f, s);
      free(s);
      break;
    // More syscall cases ...
  }
}
```

Each case statement inside the switch statement has two jobs:

1. Validate memory of arguments for the respective syscall.
2. Call the respective syscall helper function, passing in the relevant arguments.

Initialize Data Structures

1. Inside **process_execute** function, malloc the child **proc_info** struct before **thread_create** is called. To setup the struct:
   a. Initialize all semaphores to 0 (**sema_init(...)**) and locks (**lock_init(...)**).
   b. Set **exit_status** to -1 (default value if process crashes).
   c. Set **waited** to false.
   d. Set **ref_cnt** to 2 (parent + child reference).
2. Call **thread_create**, passing in **proc_info** as argument for **start_process**. After **thread_create** returns child PID, set child **proc_info.pid** to child PID. Then, add child **proc_info** to **child_info** list.
3. Inside **start_process**, store the passed **proc_info** into the current PCB's **info** pointer. Also, initialize the current PCB's Pintos lists **child_info** and **fd_table**. After **success = load(...)** call made, set the current PCB's **proc_info.load_status** to **success** and call **sema_up** on **proc_info.load_sema** for data synchronization (exec syscall).
4. Inside **userprog_init**, add the following code for minimal PCB setup.
   a. Initialize the current PCB's **child_info** list.
   b. Malloc the **proc_info** struct and store in current PCB's **info** pointer. Initialize **proc_info** variables **exit_sema** semaphore to 0, initialize **ref_cnt_lock** lock, and set **ref_cnt** to 1 (no parent).

Pointer Memory Validation

The following 3 functions are used to validate pointers passed by the user to the syscall handler.

Function 1: **static void valid_byte_pointer(uint8_t *p)**

- Description: this function validates 1 byte at pointer P

- Pseudocode:
  - Pointer P is valid if the following are all true:
    - P is not NULL
    - P is not in kernel memory (check **p < PHYS_BASE**)
    - P does not point to unmapped memory (check **pagedir_get_page(pagedir, p) != NULL**)
  - If P is an invalid pointer, call **process_exit()** to terminate the user process

Function 2: **static void valid_pointer(uint8_t *p, size_t size)**

- Description: this function validates SIZE bytes, starting from pointer P
- Pseudocode:
  - For each byte [P, P + SIZE - 1], call the **valid_byte_pointer(...)** function

Function 3: **static char *valid_str_pointer(uint8_t *p)**

- Description: this function validates string memory at P. The function returns a copy of the string (malloced, stored in kernel memory).
- Pseudocode:
  - Validate P (call **valid_byte_pointer(p)** function) and dereference character. Repeat process for subsequent bytes until dereferenced character is NULL byte.
  - While validating string, if the length exceeds PGSIZE (arbitrarily large size), terminate process (call **process_exit()**).
  - Malloc memory (length of string + 1), memcpy the string into the allocated memory, null terminate (safety), and return the string pointer.

<u>Practice Syscall Case</u>

Pseudocode

- Call **valid_pointer(&args[1], sizeof(int))**
- Call syscall_practice helper function on args[1]. This function executes as follows.
  - Stores (i + 1) in the EAX return register.

<u>Halt Syscall Case</u>

Pseudocode:

- Call **syscall_halt()** helper function. This function executes as follows.
  - Calls **shutdown_power_off()** function to shutdown the running device.

<u>Exit Syscall Case</u>

Pseudocode:

- Call **valid_pointer(&args[1], sizeof(int))**
- Call **syscall_exit** helper function on args[1]. This function executes as follows.
  - Set the current PCB’s **proc_info.exit_status** to the **exit_code**
  - Store **exit_code** in the EAX register
  - Print the exit status

- Call **process_exit()**. Inside **process_exit**, update the code as follows (before PCB is freed).
  - Call **sema_up** on the current PCB's **proc_info.exit_sema**. This synchronizes the exit syscall data write with the wait syscall data read (on exit status).
  - Call **lock_acquire** on the current PCB's **proc_info.ref_cnt_lock**. Decrement the reference count, storing the result in a local variable, and release the lock via **lock_release** function.
    - If the reference count is 0, free the current PCB's **proc_info** struct.

    Repeat this process for all the child reference counts in the current PCB's **child_info** list.
- Inside **start_process**, repeat the procedure of freeing the **proc_info** struct if the executable fails to load into memory.

<u>Exec Syscall Case</u>

Pseudocode:

- Call **valid_pointer(&args[1], sizeof(char *))** function.
- Set **char *s = valid_str_pointer(args[1])**
- Call the **syscall_exec** helper function on the string pointer S. This function executes as follows.
  - Call **process_execute** on the filename string, storing the returned child PID.
  - Find the child process **proc_info** struct in the current PCB's **child_info** list. Call **sema_down** on the **proc_info** struct's **load_sema** for synchronization. Child process calls **sema_up** on **load_sema** after the **load_status** has been set.
  - Get and check child **load_status** (found child **proc_info** struct). If true, set EAX to child PID, and -1 otherwise.
- Free the malloced string S.

<u>Wait Syscall Case</u>

- Call **valid_pointer(&args[1], sizeof(pid_t))** function.
- Call the **syscall_wait** helper function on args[1]. This function executes as follows.
  - Call **process_wait** and store returned exit code. This function is modified as follows. Note the temporary semaphore has been removed in process.c.
    - Get child process's **proc_info** struct from current PCB's **child_info** list. If not found, return -1.
    - Get child **proc_info.waited** value. If true, return -1. Otherwise, set **waited** to true.
    - Call **sema_down** on child **proc_info.exit_sema** for data synchronization and waiting. Child process calls **sema_up** on same semaphore in **process_exit**.
    - Return the child **proc_info.exit_status**.
  - Set EAX to the return value from **process_wait**.

Synchronization

There are 3 procedures that must be synchronized.

1. Load status data (exec syscall). The child process must set the load status, and the parent process must subsequently read the load status. This is achieved using a semaphore as follows.
   a. The semaphore **load_sema** in **proc_info** is initialized to 0.
   b. The parent process calls **sema_down** on **load_sema**.
   c. The child process calls **sema_up** on **load_sema** after the load status is set.

   After **sema_up** is called, the parent process can execute **sema_down** and safely read the load status.
2. Exit status data (wait syscall). The child process must set the the exit status, and the parent process must subsequently read the exit status. This is achieved using semaphores. The synchronization scheme implemented is identical to the previous one.
3. Freeing **proc_info** struct. All **proc_info** structs (except the main Pintos thread) has a reference count of 2. When reference count is decremented to 0, the **proc_info** struct must be freed. To synchronize decrements and reads, a **ref_cnt_lock** is used before and after the critical sections (acquire lock and release lock).

Rationale

- In the **syscall_handler** function, the original if-else statement is replaced with a switch statement. A switch statement allows for improved compiler optimizations and makes the code easier to read.
- Each switch statement case independetly validates argument memory. This maximizes flexibility, since different syscalls have different arguments.
- The memory validation helper functions minimize code repetition, since the procedure is frequently repeated.
- Each syscall case calls a respective helper function, which simplifies code inside **syscall_handler** and makes the overall code easier to read and maintain.
- Process information is stored inside a separate **proc_info** struct instead of inside the PCB. This is because the information might be needed after the child PCB has been freed (e.g., wait syscall on an already exited child PID). Only child and parent can access **proc_info** (restricted access scope), which prevents accidental accesses and updates by other processes.

File Operation Syscalls

Data Structures and Functions

To implement file operation syscalls, we must add a file descriptor table, which will be a Pintos **struct list**. The **struct list** will consist of **struct fd_entry** elements.

```
struct process {    // PCB struct, modified
  struct list fd_table;   // File descriptor table
  struct file *file;  // Process executable file
  int next_fd;  // Next available file descriptor
  // ... Other contents ...
}
struct fd_entry {
  struct list_elem elem; // Elem of fd_table list
  int fd;  // File descriptor (identifier of elem)
  struct file *file;
}
```

To synchronize file operations, we will have a global lock **file_op_lock**. All file operation syscalls will acquire and release the lock as appropriate. The first helper function below is used to search through the file descriptor table (NULL if **fd** not found in **fd_table**). The second helper function is used to assign file descriptors.

```
struct lock file_op_lock; // File operation lock
static struct fd_entry *get_fd_entry(struct list fd_table, int fd);
static int next_fd(void);
```

The syscall helper function structure from Process Control Syscalls will be carried over to the File Operation Syscall structure. This means, for each file operation syscall, there will be a respective helper function to execute the syscall, provided the memory validated syscall arguments.

Algorithms

<u>Initialize Data Structures</u>

Inside the **start_process** function, after PCB is successfully malloced:

- Initialize the **fd_table** Pintos list (call **list_init(...)** function).
- Set the **next_fd** value to 2 (next available fd, 0 and 1 are reserved for STDIN_FILENO and STDOUT_FILENO).

<u>Code Structure (Per Switch Case)</u>

Each file operation case, in the **syscall_handler** switch statement, will be structured as follows:

- Validate file operation syscall arguments (using helper functions in Process Control Syscall)
- Acquire **file_op_lock**
- Call respective file operation syscall helper function
- Release **file_op_lock**

Given this scheme, we will write error-defensive code so that if anything fails in a helper function, we will always release the lock before exiting, ensuring no deadlocks.

<u>Argument Validation</u>

- Arguments are validated inside the **syscall_handler** function (see Process Control Syscalls).
- Syscall Handler calls the relevant helper functions to validate pointer addresses. Reference examples in Process Control Syscalls.
- The validated syscall arguments are passed into the file operation syscall helper functions

<u>**get_fd_entry** Helper Function</u>

Pseudocode:

- Iterate through the **fd_table** (Pintos list) function argument, getting each **fd_entry** (struct list_elem).
- If any **fd_entry.fd** matches the **fd** function argument, return **&fd_entry**. Otherwise, return NULL (implying file not found).

<u>**next_fd** Helper Function</u>

Pseudocode

- From the current PCB's **struct process**, return **next_fd++.** Note, **next_fd** is the next available file descriptor for the current PCB (unique for each PCB).

<u>Create Syscall Helper Function:</u>

- Call the **bool filesys_create(const char *name, off_t initial_size)** function, storing return value in the EAX register.

<u>Remove Syscall Helper Function:</u>

- Call **bool filesys_remove(const char* name)**, storing return value in the EAX register

Open Syscall Helper Function:

- Call **struct file *filesys_open(const char *name)**, storing the returned file pointer.
- If the file pointer is NULL, set the EAX register to -1 and return.
- If the file pointer is not NULL, malloc a new **struct fd_entry**. Set its **fd_entry.fd** as the value returned by the function call **next_fd().** Set its **fd_entry.file** as the file pointer returned by **filesys_open** function call. Add the **struct fd_entry** to the current PCB's **fd_table** list.
- Store the **fd_entry.fd** value in the EAX register.

Filesize Syscall Helper Function:

- Call the **get_fd_entry** helper function, passing in the file descriptor **fd** on the file descriptor table **fd_table** from the current PCB. If function returns NULL (file descriptor not in table), then set EAX to -1 and return.
- Call **off_t file_length(struct file* file)** on the **fd_entry.file** pointer (**fd_entry** retrieved in previous step). Store return value of function call in EAX register.

Read Syscall Helper Function:

Argument validation (buffer)

- To validate the buffer (arg[1]) of size (arg[2]) in the switch cause, use the following code structure. This structure can be repeated for validation of other syscall arguments.
  - Call **valid_pointer(&arg[1], sizeof(void *))** and **valid_pointer(&arg[2], sizeof(unsigned))**
  - Call **valid_pointer(arg[1], arg[2])**. This validates the buffer memory itself.

Pseudocode:

- If **fd** (syscall argument) is 1 (corresponds to STDOUT_FILENO), store -1 in EAX and return. Data cannot be read from standard out.
- Else if **fd** (syscall argument) is 0 (corresponds to STDIN_FILENO), store **size** into EAX register and execute the following code **size** times.
  - Call **input_getc()**, and store the returned uint8_t at byte specified by **buffer** pointer
  - Advance **buffer** pointer by 1 byte
- Else, get **fd_entry** from a call to **get_fd_entry** helper function on the **fd** function argument. If a NULL value is returned, store -1 in EAX and return. Else, call **off_t file_read(struct file* file, void* buffer, off_t size).** Store the returned value in EAX.

Write Syscall Helper Function:

Argument validation:

- Validate arguments (e.g., buffer) using the previously described validation strategies and the validation helper functions.

Prevent file writes to running executable in Pintos:

- Inside the **load** function in **process.c**, add the following code before **filesys_open** is called.
  - Acquire global **file_op_lock**
- Inside the **load** function in **process.c**, add the following code after **filesys_open** is called.
  - Call **file_deny_write** function on the file pointer returned by the call to **filesys_open**
  - Release global **file_op_lock**
  - Store the file pointer in the current PCB struct's **file** pointer variable.
  - Remove the **file_close** call at the end of the function.
- Inside **start_process**, if a file was not successfully started, execute the following code.
  - Acquire global **file_op_lock.** Then, **c**all **file_allow_write** function on the file pointer inside the current PCB process struct's **file** pointer variable. Then, call **filesys_close** on the file. Then, release global **file_op_lock**
- Inside the **process_exit** function, if the current PCB's **process_name** is not NULL, execute the following code.
  - Acquire global **file_op_lock.** Then, **c**all **file_allow_write** function on the file pointer inside the current PCB process struct's **file** pointer variable. Call **filesys_close** on the file. Then, release global **file_op_lock.**

Pseudocode:

- If **fd** (syscall argument) is 0 (STDIN_FILENO), store 0 in EAX and return. Standard input cannot be written to.
- Else if **fd** (syscall argument) is 1 (STDOUT_FILENO), execute the following code.
  - If **size** (syscall argument) <= 256, call **putbuf(buffer, size)**
  - Else, create a local variable **char output[256]**.
    - In 256 byte chunks, write data from **buffer** to the **output** until full (or **buffer** empty) and call **putbuf** on args **output** and **n** (number of bytes written to **output** from **buffer**). Repeat process, until all of **buffer** has been outputted (by rewriting **output** and recalling **putbuf**).
- Else, call **get_fd_entry** helper function on **fd** (syscall argument) to get a pointer to a **fd_entry**.
  - If a NULL pointer is returned, store -1 in EAX register and return.
  - Call **off_t file_write(struct file* file, const void* buffer, off_t size)** on the **fd_entry.file** pointer, the **buffer** (syscall argument), and the **size** (syscall argument). Store the returned value in EAX register and return.

<u>Seek Syscall Helper Function (takes in fd)</u>

- Find file corresponding to file descriptor **fd** with the **fd_get_entry** helper function**.** If the function returns NULL (file does not exist), return (do nothing).
- Otherwise, call **void file_seek(struct file* file, off_t new_pos)** on **fd_entry.file** and **position (**syscall argument).

Tell Syscall Helper Function (takes in fd)

- Find file corresponding to file descriptor **fd** with the **fd_get_entry** helper function**.** If the function returns NULL (file does not exist), return (fail silently).
- Otherwise, call **off_t file_tell(struct file *file)** on **fd_entry.file** pointer. Store return value in EAX register.

Close Syscall Helper Function (takes in fd)

- Find file corresponding to file descriptor **fd** with the **fd_get_entry** helper function. If the function returns NULL (file does not exist), return (fail silently).
- Call **void file_close(struct file* file)** on found **fd_entry.file** pointer. After file is closed, remove **fd_entry** from the PCB's **fd_table** list and free **fd_entry.**

In **process_exit**, loop through all **fd_entry** in **fd_table** and close the files (same procedure as described above).

Synchronization

For each file operation syscall, we will acquire the global **file_op_lock**, then call the relevant Pintos file operation functions, then release the global **file_op_lock**. This will ensure synchronization between threads, as only one thread can be executing a file operation.

Rationale

- The file descriptor tables are unique to each PCB, so the data is stored inside each PCB's process struct.
- The file lock is global, so all threads can access the lock. A lock is used, since it once it is acquired, only the owner thread can release it.

Floating Point Operations

Data Structures and Functions

Use a 108 byte char array to hold the FPU inside relevant data structures.

```
struct intr_frame { // threads/interrupt.h
  // ... existing contents ...
```

```
  char FPU[108];  };
struct thread { // threads/threads.h
  // ... existing contents ...
  char FPU[108]; };
```

Algorithms

- Set CR0 register with given assembly instructions to indicate the FPU is available. Line 150 in threads/start.s

```
orl $CR0_PE | CR0_PG | CR0_WP | CR0_EM, %eax
```

gets changed to

```
orl $CR0_PE | CR0_PG | CR0_WP, %eax
```

OS startup (inside main function in init.c):

- After we enable FPU in start.S, initialize FPU with FINIT.

Starting a new process:

- In **start_process** (userprog/process.c), add asm FINIT command to initialize the FPU.

Thread switching (**cur** thread, **next** thread):

- To call FSAVE (save) and FRSTOR (restore), we need a stack address. Therefore, we must calculate relative offsets in **switch_thread_frame** struct's **thread** struct to find address of the char FPU[108] for the respective thread.
- In switch.S, we call FSAVE to save **cur** FPU, and we call FRSTOR to restore **next** FPU.

Context switches:

- We must compute the relative offset of char FPU[108] in intr_frame struct.
- START INTERRUPT
  - Store the current FPU in to interrupt frame struct's FPU array by calling FSAVE (also initializes the FPU according to documentation).
- EXIT INTERRUPT
  - Restore the char FPU[108] from the **intr_frame** struct.

<u>Compute-E Syscall Helper Function</u>

- Takes in an int n, already validated (reference Process Control Syscall)
- Set EAX register to **(float) sum_to_e(n).** The function returns a double (8B), so must be cast to a float (4B) before stored in EAX register.

Synchronization

As with GPRs, the FPU has to be independent for each thread. In our design, we store FPUs for each thread context switch or interrupt, so nothing is shared and there is no need for any locking scheme.

Rationale

We plan on using FINIT and FSAVE because of their error-checking functionalities instead of the alternatives FNINIT and FNSAVE (no error checking). Determining byte offsets for each FPU will be difficult to determine, and may change if the structs are updated in the future.

Concept check

1. "sc-bad-sp.c" This test case assigns ESP to a negative address in line 16. Therefore, we must exit and terminate with the error code -1 since negative addresses are not mapped to any valid memory.
2. "sc-boundary-3.c" This test case uses a valid stack pointer when making a syscall, but the pointer is too close to the page boundary, therefore some syscall arguments are located in invalid memory. This test originally gets the boundary where the memory is invalid and assigns it to **char *p** (line 11). **p** is then decremented, so it is pointing valid memory (line 12) and is assigned to 100 (non-Null, line 13). Thus, to read string **p**, memory beyond p[0] (which is all invalid) must be accessed, even though p[0] is in valid memory. Therefore, Pintos should terminate the process (exit with status -1).
3. The remove file operation syscall is not well tested by the existing test suite. The "open-missing.c" test tries to open a non-existent file (this should fail). To test if the remove file operation is working, we should expand this test as follows. Start by creating a file, then opening the file, then removing the, then closing the file, then try to re-open the same file (this should fail). This confirms that the remove operation is successfully removing files after all references to the file are closed. This test also confirms that a previously opened file cannot be reopened if it has been deleted.