Project User Programs Design

group48

Name	Autograder Login	Email
Ben Plate	student219	bplate@berkeley.edu
Jeffrey Liang	student50	jeffreyliang@berkeley.edu
Teddy Cho	student139	teddycho@berkeley.edu
Theophilus Pedapolu	student270	theopedapolu@berkeley.edu

Argument Passing

Data Structures and Functions

Creating new data structures or modifying existing data structures is not necessary for argument passing.

The logic for argument passing goes in the `load` function in `userprog/process.c`. It will be helpful to use the `strtok_r` and `strlcpy` functions in `lib/string.h`

Algorithms

These additions and modifications all go in the `load` function in `userprog/process.c`.

Before the call to `filesys_open` on line 286, we do the following.

- Using `strtok_r`, split `file_name` into tokens with space as a delimiter. We store these tokens in a fixed-size `char*[]` on the kernel stack, so we impose a maximum size on the arguments list.
- Pass the first string returned from `strtok_r` (i.e., the executable file name) to `filesys_open`.

Then, after the call to `setup_stack` on line 351, we push the following to the user stack starting at `PHYS_BASE`.

- On the user stack, initialize `char*[] argv` with size equal to the number of tokens plus one. We choose this length and set its last element (i.e., `argv[argc]`) to `NULL`.
- For each token, copy it to the user stack using `strlcpy` and store a pointer to its location on the user stack in `argv`.
- Insert padding so that `*esp` will be 16-byte aligned after pushing arguments to the user stack.
- Push the `argv` pointer to the user stack.
- Push the number of tokens (i.e., `argc`) to the user stack.
- Push a `NULL` return address to the user stack.

Once this is done, we can set `*eip` as before, close the file, and continue with the rest of `start_process`.

Synchronizations

We must ensure that the executable currently running cannot be modified on disk. We use the same synchronization strategy as for the `exec` syscall (discussed later in this document).

Rationale

The logic for argument passing follows from Program Startup Details in the Pintos Documentation. We choose to first store tokens in the kernel stack and then copy them to the user stack because the first token is required for the call to `filesys_open` but we cannot interact with the user stack until after the call to `setup_stack`.

No additional synchronization strategy is necessary because the variables discussed can be accessed only by the current thread.

Process Control Syscalls

Data Structures and Functions

In `userprog/process.h`, we add `struct wait_info` and add fields to `struct process` as follows.

```
struct exit_status {
  pid_t pid;                   /* Process id */
  int status;                  /* Exit status */
  int ref_cnt;                 /* Initialized to 2 */
  struct lock ref_cnt_lock;    /* Lock for ref_cnt */
  struct semaphore exit_wait; /* Down'd by parent's process_wait, up'd by process_exit */
  struct list_elem elem;      /* List element for child_exit_statuses in struct process */
};
struct process {
  ...
  struct list_elem child_elem;     /* List element for child list */
  struct list children;            /* List of child processes */
  struct lock children_lock;       /* Lock on children */
  struct exit_status* exit_status; /* Pointer to this process's exit status */
  struct list child_exit_statuses; /* List of childrens' exit statuses */
};
```

In `userprog/process.c`, we add `struct start_process_args`.

```
struct start_process_args {
  char* file_name;                 /* Executable's file name */
  struct process* parent_process; /* PCB of the parent process */
  struct semaphore* exec_wait;    /* Down'd by process_execute, up'd by start_process */
  bool success;                    /* Set by start_process */
};
```

Syscalls are handled by the `syscall_handler` function defined in `userprog/syscall.c`. For each syscall number (found in `args[0]`), we execute the corresponding handler logic (described below).

We add the following functions to `userprog/syscall.c`.

```
static bool is_valid_uaddr(const void* vaddr);
static bool is_valid_user_memory(const void* vaddr, size_t size);
static bool are_valid_args(const int32_t* args, size_t num_args);
static bool is_valid_string(const char* str);
```

We modify the signature of the `process_exit` function in `userprog/process.h`.

```
void process_exit(int status);
```

We modify the signature of the `start_process` function in `userprog/process.c`

```
static void start_process(void* args_);
```

Algorithms

Accessing User Memory

`is_valid_uaddr` checks if the given `vaddr` is a valid user address. Specifically, it returns true if `vaddr` is not `NULL` and `is_user_vaddr(vaddr)` returns true and `pagedir_get_page(thread_current()->pcb->pagedir, vaddr)` is not `NULL`, and false otherwise.

`is_valid_user_memory` checks that the `size` bytes of memory addressed by `vaddr` are in valid user memory. It does so calling `is_valid_uaddr` for each address from `vaddr` to `vaddr + size - 1` (treating `vaddr` as a `char*` that addresses one byte of memory).

`are_valid_args` checks that the `num_args` syscall arguments at `args` exist in valid user memory. It does so by calling `is_valid_user_memory(args, num_args * sizeof(int32_t))`. In particular, we should call `are_valid_args(args, 1)` before accessing the syscall number in `args[0]`.

`is_valid_string` checks that the given string `str` is `NULL` terminated and exists entirely in user memory. It performs the following procedure.

- For each virtual address starting at `str`, call `is_valid_user_memory` on that address.
  - If it returns true, check if the `char` at that address is `NULL`. If so, return true; otherwise, continue.
  - If it returns false, return false.

practice

The `syscall_handler` logic is as follows.

- Call `are_valid_args(args, 2)`, and call `process_exit(-1)` if false.
- Set `f->eax` to `args[1] + 1`.

halt

The `syscall_handler` logic is as follows.

- Call `shutdown_power_off` in `devices/shutdown.h`.

te

exit

The `syscall_handler` logic is as follows.

- Call `are_valid_args(args, 2)`, and call `process_exit(-1)` if false.
- Print the exit status of the user program with format `%s: exit(%d)`.
- Call `process_exit(args[1])` in `userprog/process.c`.

The `process_exit` function should be modified to do the following.

- Get the current `pcb` from `thread_current()->pcb`.
- For each element in `pcb->child_exit_statuses`, pop it from the list using `list_pop_front`, get a pointer to the corresponding `exit_status` structure, decrement `exit_status->ref_cnt` (acquiring and releasing `exit_status->ref_cnt_lock` as appropriate), and free `exit_status` if its `ref_cnt` reaches zero.
- For each element in `user_files` (described in File Operation Syscalls), pop it from the list using `list_pop_front`, get a pointer to the corresponding `user_file` structure, call `file_close` (in `filesys/file.h`) on `user_file->file`, and free `user_file`.
- Decrement `pcb->exit_status->ref_cnt`.
  - If it equals zero, free `pcb->exit_status`.

    - If it is greater than zero, set `pcb->exit_status->status` to the given `status`.
- Remove `pcb` from `pcb->parent_process->children` (acquiring and releasing `pcb->parent_process->children_lock` as appropriate).
- Free `pcb`.
- Call `sema_up` on `pcb->exit_status->exit_wait` if `pcb->exit_status` was not freed. To do this, we have to store a pointer to the semaphore before freeing the PCB.

In addition, we must modify `userprog/exception.c:81` to `process_exit(-1);` so that a process that has been killed by the kernel has exit status `-1`.

exec
- Call `are_valid_args(args, 2)`, and call `process_exit(-1)` if false.
- Call `is_valid_string(args[1])`, and call `process_exit(-1)` if false.
- Call `process_execute(args[1])`.
- If `process_execute` returns `TID_ERROR`, return `-1` in `f->eax`; otherwise, return the pid returned from `process_execute` in `f->eax`.

We modify `process_execute` in `userprog/process.c` as follows.
- Create a semaphore `exec_wait` initialized to 0
- Create a `start_process_args` struct with `file_name`, `thread_current()->pcb`, and `exec_wait`, and pass it to `thread_create` as the `aux` argument.
- Call `sema_down` on `exec_wait`.
- If `args->success` is true, return `tid` (as the function already does); otherwise, return `TID_ERROR`.

We modify `start_process` in `userprog/process.c` as follows.
- Cast `args_` to `struct start_process_args* args`.
- Get `char* file_name` from the `args->file_name`.
- Add `new_pcb` to `args->process->children` (acquiring and releasing `args->process->children_lock` as appropriate).
- Create a `struct exit_status` for the new process, set `new_pcb->exit_status` to point to it, and add it to `args->parent_process->child_exit_statuses`.
    - `exit_status` should be initialized with the appropriate `pid`, `ref_cnt` equal to 2, and `exit_wait` initialized to 1.

- On failure, set `args->success` to false and call `sema_up` on `args->exec_wait` before calling `thread_exit`.
- On success, set `args->success` to true call `sema_up` on `args->exec_wait` before simulating a return from an interrupt to start the process.

wait

- Check `are_valid_args(args, 2)`, and call `process_exit(-1)` if false.
- Call `process_wait` with `args[1]`, and return its value in `f->eax`.

We implement `process_wait` in `userprog/process.c` with the following procedure.

- Get the current `pcb` from `thread_current()->pcb`.
- Iterate through `pcb->child_exit_statuses` to get the `exit_status` struct `child_exit_status` corresponding to the given `child_pid`. If it is not found, return `-1`.
- Iterate through `pcb->children` to get the `child_pcb` (acquiring and releasing `pcb->children_lock` as appropriate).
    - If `child_pcb` is not found, the child process has already exited, so we return `child_exit_status->status`.
    - If `child_pcb` is found, the child process has not yet exited, so we wait for it by calling `sema_try_down` on `child_exit_status->exit_wait`.
        - If this returns true, the current process is now waiting for the child process. Return `child_exit_status->status` after the semaphore is eventually up'd by `process_exit`.
        - If it returns false, `wait` was already called on this child, so we return `-1`.

Synchronization

We must ensure that the file passed to `process_execute` cannot be modified on disk while it is running. We can do so by calling `file_deny_write` (from `filesys/file.h`) on `file` in the `load` function in `userprog/process.c`.

We discuss synchronization for the following variables. At the moment, each process has only one thread, so race conditions happen only between processes.

- `struct exit_status`'s `pid` and `status` are modified only by the process that this structure describes and are read only after this process exits, so no synchronization is necessary here.

- `struct exit_status`'s `ref_cnt` is modified on exit by both the parent and the child. Therefore, we require a lock on `ref_cnt`.
- `struct process`'s `children` (and `child_elem`) may be modified by multiple processes exiting at the same time. We therefore require a lock on `children`.
- `struct process`'s `exit_status` is set with the PCB is created a never modified, so no synchronization is necessary here.
- `struct process`'s `child_exit_statuses` (and `struct exit_status`'s `elem`) is modified only by a starting child process. A process can start only one child process at a time, during which the parent process is waiting, so no synchronization is necessary here.
- `struct start_process_args` is read by `process_execute` only after `start_process` as succeeded or failed, so no synchronization is necessary here.

Rationale

We choose to keep two lists of children in `struct process` (`children` and `child_exit_statuses`) because a process must be able to get the exit status of a child that has already exited. Within each process's PCB, we store a pointer to its exit status structure (`exit_status`) so that it can set its value before it exits. However, this exit status structure is freed only when its parent exits so that this information persists after the process has exited. Similarly, we store the `exit_wait` semaphore in `exit_status` because it still needs to be accessed by a process's parent after it has exited and its PCB has been freed.

We must modify `start_process` to accept `struct start_process_args` because we want to pass the parent process's PCB, a semaphore, and a success boolean. The parent processes's PCB is necessary because the child PCB and exit status must be added to the parent PCB, the semaphore is necessary to make `process_execute` wait for `start_process` to succeed or fail, and the success boolean is modified to return the status or `start_process` back to `process_execute`.

We require the functions discussed in Accessing User Memory because we can't trust the user to pass arguments correctly on a syscall. We choose to check the validity of a user-provided pointer before dereferencing it, rather than handling the page fault, because it is simpler and speed is not a major consideration for this project.

File Operation Syscalls

Data Structures and Functions

To ensure that all file operations done in user code are funneled through system calls and handled by the kernel, we will need to define a structure created for every given process that maps user-level file descriptors to kernel-level files.

To this end, we will make use of the list data structure defined in `lib/kernel/list.c`. a list element will be defined as follows
:

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

The `elem` attribute is used to allow this struct to be used in a list. We cannot use the position of the `user_file` in the list in order to determine its file descriptor as this raises two problems:

1. `stdin` and `stdout` are allocated to `fd = 0` and `fd = 1` respectively, but these do not have corresponding `file*` structs and must be handled specially
2. If a file in the middle of the list is closed, we would have to add a special attribute to keep track of the fact that it is closed as removing it would offset all future files' descriptors by 1. This would also lead to fragmentation and more complex code.

For these reasons, we have the `fd` attribute define the descriptor for a given open file in a process. The `file` attribute points to the kernel-level file that will be used to run kernel-level functions, like reading and writing.

We will also add the following attributes to the `process` struct:

```
struct process {
  ...
  struct list user_files;
```

```
  int num_opened_files;
}
```

The `user_files` list will contain all the `user_file` structs mentioned above. We store `num_opened_files` to keep track of which file descriptors have not been used. When a process is initialized, it will be set to 2 (`stdin` and `stdout` are reserved) and when a file is opened, it will be given the file descriptor `num_opened_files` (assuming it is successfully opened) and `num_opened_files` will be incremented by 1.

Algorithms

create

This syscall will make use of the `filesys_create` function defined in `filesys.c`. Since the arguments in the syscall have a one-to-one correspondence to `filesys_create`, we can just pass these through in the handler. We will have to be attentive to malicious inputs however, especially given that we work with a `char*` argument that will need to be checked for valid memory access and proper null-termination. Since filenames are a maximum of 14 characters long, we don't have to worry too much about the latter issue. The returned value will be the success of this operation.

remove

This syscall has a similar idea to the create syscall, but corresponding to the `filesys_remove` function defined in the same file. However, one important thing to keep in mind for this syscall is the side effect of processes that have the file to be removed open at the same time. Since the existing file function implement the appropriate UNIX file semantics, we don't have to think much about this. The returned value will be the success of this operation.

open

To open a file, we use the `filesys_open` function in `filesys.c` using the supplied name, again checking that the argument is safe and located in memory that the process has access to. Using the structure mentioned above, we won't be returning the file returned by `filesys_open` to the user, but rather we'll create a `user_file` struct and add it to the process' list of open files. The descriptor will be the value stored in the process' `num_files_opened` that will subsequently be incremented if the operation is successful, and this is the value that will be returned.

filesize
The filesize syscall will use `file_length` function found in `file.c`. For this function (and subsequent file operation functions), we will have to convert the file descriptor passed as an argument to the file struct that the kernel uses to perform these operations. To do this, we can iterate through the list searching for the file descriptor and retrieving the `file` where `fd` is matched. If there is no match, we just return -1 and stop the function. Otherwise, we return the value returned by calling `file_length` with the `file` struct. The filesizes of `stdin` and `stdout` are undefined so we'll return 0.

read
To read a file, we first need to perform a check on the buffer being passed into the syscall by the user. Not only does it need to be accessible by the process, but it and all addresses between `buffer` and `buffer + size` must be writable. If this is not the case, we'll return -1. Otherwise, we'll retrieve the `file` from the file descriptor using the above method and run the `file_read` function from `file.c` using the arguments given. This function copies the contents to `buffer` and advances the file position as necessary, returning the number of bytes actually read, which is what the syscall will return too.

There is a special case we also need to address: reading from `stdin`. Attempting to read from `stdout` (`fd = 1`) is not possible so we can just return -1. For `stdin`, (`fd = 0`), we'll use a for loop to call `input_getc` (located in `devices/input.c`) `size` times and store each character in `buffer`'s position (incremented with each call to `input_getc`).

write
Writing from a file is similar to reading: we need to ensure all addresses from `buffer` to `buffer + size` are accessible by the process. We'll use the `file_write` function from `file.c` and return its value.

For `stdout`, we'll need to make use of the `putbuf` function found in `lib/kernel/console.c`. According to the spec, `putbuf` can cause problems with interleaving text across different processes if the buffer is more than a few hundred bytes, so the solution is to use a `size_t` in `putbuf` of 256 and call it multiple times until `buffer` is exhausted. Writes to `stdin` will simply return -1.

seek
Seek will make use of `file_seek` in `file.c` to seek the file to the given position. There is no return value. Seeking `stdin` or `stdout` has no effect.

tell
Tell will make use of `file_tell` in `file.c` to return the position of the next byte to be read for the file. Calling tell for `stdin` or `stdout` will return 0.

close
Calling close on a given file descriptor will search through the process' `user_files` to find the element with the given file descriptor. Calling this function on `stdin`, `stdout`, or a file descriptor that isn't allocated has no effect. When the file is found, the `file_close` function in `file.c` is called with the given `file` struct. We will also remove the respective list element from the list by calling `list_remove` from the list library.

Synchronization

Using the file system across multiple processes can cause many complications, however for this project we will just make use of a global lock to prevent multiple file operations from occurring at the same time. To this end, we'll add a global variable `lock_t filesys_lock` in `syscall.c`; we'll call `lock_init` in `syscall_init` and for every filesystem-related syscall, it will be prepended by `lock_acquire(&filesys_lock)` and appended with `lock_release(&filesys_lock)`.

Rationale

The principal design decision for this section is to create a `user_file` struct and have the method of converting between file descriptor and kernel-level file being the `user_files` list. We decided that this would be the best method of addressing this problem as other methods either produced more complexity or introduced undesirable behaviors.

One idea would have been to simply cast the kernel file to an integer and use that as the file descriptor. This would have made the process of getting the file descriptor from the file very easy, but leads to many problems that makes this unacceptable. Although it's easy to get the file descriptor from the file, going the other way would be difficult and would itself require a mapping to keep track of it. It also opens the door to possible collisions in previously opened files or `stdin`/`stdout`, which would not be trivial to address. Furthermore, doing this method would reveal information about the kernel file struct, and it may not be desirable for user code to have access to this information. Although not part of the spec, it would also make `dup` impossible without more complexity. All in all, this method did not seem like the answer at all and it made more sense to just assign the numbers arbitrarily and associate the two using a list.

In addition to being easy to conceptualize/code, it also provides good performance. The list method may not be fastest as it involves searching for a file/file descriptor each time we need the other, but considering that a process is unlikely to have more than 1000 open files at a given time, this seemed like a minor detail. If it really made a difference, we could modify `list.c` to use binary search based on the file descriptor, since each opened file is added to the end of the list and the file descriptor is strictly increasing. Again though, this didn't seem like a necessary implementation considering that the list wouldn't be big.

In terms of the other design choices, we were able to find solutions that only used either one or two kernel functions for a given syscall with a one-to-one correspondence between the syscall arguments and kernel function arguments.

Floating Point Operations

Data Structures and Functions

Modify the `struct intr_frame` struct in `threads/interrupt.h` as follows:

```
struct intr_frame {
  /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
  char fpu_state[128] /* NEW ADDED FIELD to store entire FPU s
tate */
```

```
  uint32_t edi;       /* Saved EDI. */
  uint32_t esi;       /* Saved ESI. */
  uint32_t ebp;       /* Saved EBP. */
  uint32_t esp_dummy; /* Not used. */
/* Rest of the fields were truncated for brevity */
...
};
```

Modify the `struct switch_threads_frame` in `threads/switch.h` as follows:

```
struct switch_threads_frame {
  char fpu_state[128]  /* NEW ADDED FIELD to store entire FPU state */
  uint32_t edi;         /*  0: Saved %edi. */
  uint32_t esi;         /*  4: Saved %esi. */
  uint32_t ebp;         /*  8: Saved %ebp. */
  uint32_t ebx;         /* 12: Saved %ebx. */
  void (*eip)(void);    /* 16: Return address. */
  struct thread* cur;   /* 20: switch_threads()'s CUR argument. */
  struct thread* next;  /* 24: switch_threads()'s NEXT argument. */
};
```

We will use the `fsave` instruction in the x86 ISA to save the FPU environment and register stack; it also re-initializes the FPU state. We restore the FPU state using the `frstor` instruction, and use `fwait` for synchronization purposes. We also use the `finit` instruction to initialize the FPU

Change: we use fninit instead of finit so that the FPU doesn’t use gargabe data in the FPU to error check. Only use fninit, fsave, frstor

Change: we don’t need to use fwait because we don’t need to wait for frstor

Algorithms

OS System Startup

Add `finit` in line 31 of `threads/start.S`, the section that initializes the segment registers, to initialize the FPU as well.

Thread/Process creation

In the `thread_create` function in `threads/thread.c`, after line 182, create a `char temp[108]` local variable. Using asm volatile:

- `fsave 0(&temp)` to store the current thread's FPU state in the local variable. Also re-initializes the FPU
- `fsave 0(&(sf→fpu_state))` to save the initialized state in the new thread's `switch_threads_frame` so the next context switch to this thread will restore a clean FPU state
- `fwait` to handle any pending FPU exceptions first
- `frstor 0(&temp)` to restore the current thread's FPU state

Because `process_execute` calls `thread_create` to create the main thread for a new process, this modification will also ensure new processes have an initialized FPU state.

context switches

Add the following x86 instructions to the `switch_threads` function of `switch.S` after
`pushl %edi`. We save the current FPU state on `switch_threads_frame` struct

```
fsave 0(%esp)
addl $-128, %esp // Don't need to change esp fsave automatically changes esp
```

Add the following before `popl %edi` in the `switch_threads` function. After the context switch, we restore the new thread's FPU state from the stack

```
addl $128, %esp // Don't need to move esp .
fwait
frstor 0(%esp)
```

interrupts

Add the following the `intr_entry` function of `intr-stubs.S` after `pushal`. We save the current FPU state on the `intr_frame` struct

```
fsave 0(%esp)
addl $-128, %esp
```

Add the following to the `intr_exit` function before `popal`. After the interrupt is handled, we store the thread's FPU state

```
addl $128, %esp
fwait // No need to use fwait
frstor 0(%esp)
```

compute_e

if n < 0, return -1 in `f->eax`, otherwise return the result of `sys_sum_to_e(n)` in `f->eax`

Synchronization

We want to ensure that no FPU operations by other threads are being conducted while we are saving, restoring, or initializing the FPU state. We use the `fsave` and `finit` instead of `fnsave` and `fninit` so that we check and handle pending floating-point exceptions before saving or initializing. Additionally, we use `fwait` before `frstor` instructions to wait for the floating point unit before proceeding to load the state.

Rationale

We chose to add an `fpu_state` field to the `intr_frame` and `switch_threads_frame` structs so the entire 108-byte FPU state can be saved/restored on interrupts and context switches. Just like the GPR registers, we save the FPU state on a thread's stack and restore it (a different thread if we are doing a context switch) using assembly. Another approach we considered was to simply increment/decrement the stack pointer to make space for the FPU state, without having an `fpu_state` field. However, this would have caused memory to possibly go outside the thread's allocated stack since we are using more memory than the `intr_frame` and `switch_threads_frame` structs have.

For initializing the FPU when new threads or processes are created, we decided to use a temporary local variable in `thread_create` to store the current thread's FPU state so it is not lost when we clean the FPU. Then we store the cleaned state into the new thread's `switch_threads_frame` struct, so when the new thread is scheduled and runs, the context switch will restore a clean FPU state. Finally, we restore the current thread's FPU state. Another approach we looked at was creating a variable within the `thread` struct itself to act as the temporary storage location instead of a local variable within `thread_create`. However, this would just add unnecessary complexity and take up more space than needed, since in this case the 108-byte storage location would persist even after `thread_create` exits.

Concept check

1. One test that uses an invalid stack pointer is `sc-bad-sp.c`. This test uses assembly to move the stack pointer down by 64 MB, which is an invalid region of memory for the process. It then proceeds to use the `int $0x30` instruction, which triggers an interrupt. The syscall handler should recognize that the stack pointer is in an invalid region of memory and exit the program with a status of -1. If that doesn't happen, the process is restored and the next line of code is run, which fails the test as the program should have exited earlier.
2. `sc-boundary-3.c` is a test with a valid stack pointer but arguments in invalid memory. On line 11, it gets a memory address on the boundary of valid and invalid memory with address stored in `p`. The code then decrements the address by 1, placing it in valid memory, but stores the integer 100 in that address. While `p` is a valid address, since an integer is 4 bytes it partially goes into invalid memory and therefore should be rejected by the kernel as an invalid interrupt, should it be passed as an argument. The next line of assembly sets the stack pointer and calls an interrupt that uses the address at `p` as an argument, which is valid but because the integer is greater than 1 byte, goes into invalid memory and so the syscall should kill the program.
3. Something that isn't tested is a syscall made that has syscall number that doesn't have a corresponding handler. In other words, it could be possible that a number greater than 31 is pushed to the stack, then `int $0x30` is called. Because syscall numbers are only defined from 0 (SYS_HALT) to 31 (SYS_INUMBER), it's possible that a number outside of this range is pushed as the first argument to the syscall

handler, but the kernel has no way of handling that syscall and should therefore kill the process. The test would therefore use the following code:

```
asm volatile("pushl $0x20; int $0x30");
fail("should have killed process");
```

The first line pushes 32 to the stack and makes a syscall interrupt. Since 32 is not a defined syscall, the program should be killed. If it isn't, the kernel will return to the user code and call the fail function, which fails the test.