Operating System Project 3

System Call

Team 10: 20195008 곽병혁, 20195052 김희수, 20195182 최승훈

June 10, 2024

1 Definition

This third project aims to implement the overall system call procedure in PintOS so that any user program can be executed properly. The System call is a programming interface to the services provided by the Operating System. When a user invokes a system call, a system call interface mediates the procedure between the user application and kernel as **Figure** 1 depicted below. Then by referencing the system call table via an index number, the interface invokes the intended system call in the kernel and returns the status code to users.

Currently, the user program in PintOS terminates processes not in an explicit way so tracing the program execution is considerably challenging to programmers. Hence the first goal of this project, **Problem 1**, is to print the termination messages obeying the following convention; *printf("%s: exit(%d)\n", ...);*. System calls, for sure, require several parameters to be executed properly. That means this scheme should acquire parameters from the user program and send them to the system call. Since a single command line from the user application contains a program name and arguments, the interface has to parse the string into each argument and pass it to the invoked program or system call, which is indeed **Problem 2**. The final step, **Problem 3**, is to write various system calls such as *halt*, *open*, *read* and *write*.

2 Algorithm

2.1 Process Termination Messages

The objective is to print a process name and exit code when the user process terminates for any reason. This is comparably simple to implement; implementing the function *exit(int status)* in syscall.c to print a message can complete this problem.

[Modified]

- syscall.c

2.2 Argument Passing

Creating a user process can send arguments to the system call. To pass the parameters to the proper function, parsing the string (command line) to each token in the *process_execute()* should be preceded. Then the system saves tokens, which are the program name and arguments, at the stack so that the program can be executed properly. After *start_process()* parses arguments, *load()* and *load_segment()* push arguments to a stack that *setup_stack()* sets. Lastly, an exception for the failure of loading arguments is added.

[Modified]

- process.c & process.h
- syscall.c & syscall.h

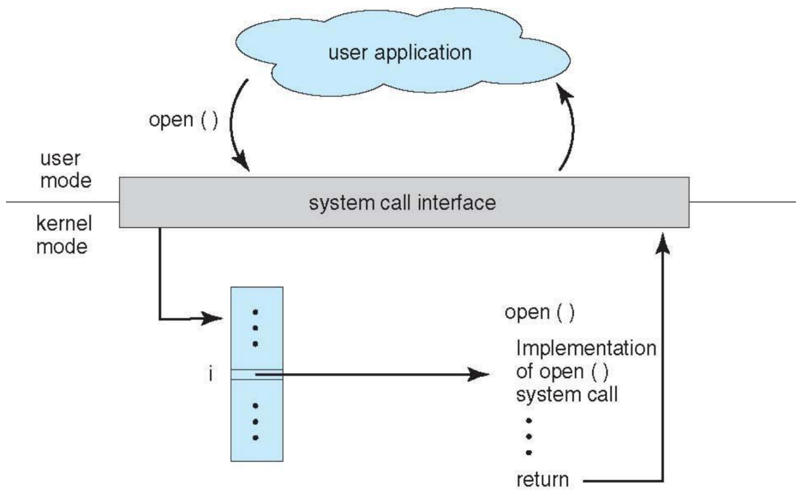


Figure 1: System call diagram.

- exception.c & exception.h
- thread.c & thread.h

2.3 System Calls

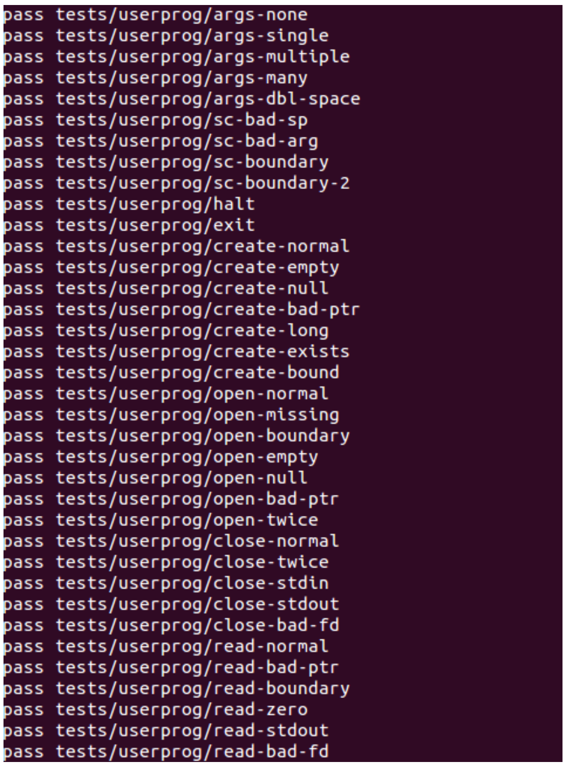
Implementing system calls is based on the skeletons of the system call handler. Below is a list of System calls and short descriptions for each of them we implement.

- void halt (void)
  : turns off the system
- void exit (int status)
  : terminates a process
- pid_t exec (const char *cmd_lime)
  : creates a new process and executes the command line
- int wait (pid_t pid)
  : waits for the termination of a process and returns its exit code. Checking a child and parent process states requires additional thread member variables. Semaphore is used.
- bool create (const char *file, unsigned initial_size)
  : creates a new *initial size*d file with the given function, */filesys/-filesys.c:filesys_create()*
- bool remove (const char *file)
  : removes the file from the system with the given function, */filesys/-filesys.c:filesys_create ()*
- int open (const char *file)
  : opens the file, adds array typed file descriptor to *thread* structure, and returns the file descriptor. This function iterates *fd* to refer the file to open at the space.
- int filesize (int fd)
  : returns the size of the opened file in bytes measured by the array-typed file descriptor
- int read (int fd, void *buffer, unsigned size)
  : reads the given amount of data from the file descriptor and saves it in the buffer
- int write (int fd, const void *buffer, unsigned size)
  : write the given amount of data, which is in the buffer, on the file descriptor
- void seek (int fd, unsigned position)
  : locates the cursor for read/write to the *position*
- unsigned tell (int fd)
  : returns a path of the opened file
- void close (int fd)
  : closes the file after flipping the corresponding *fd* values to *Null*

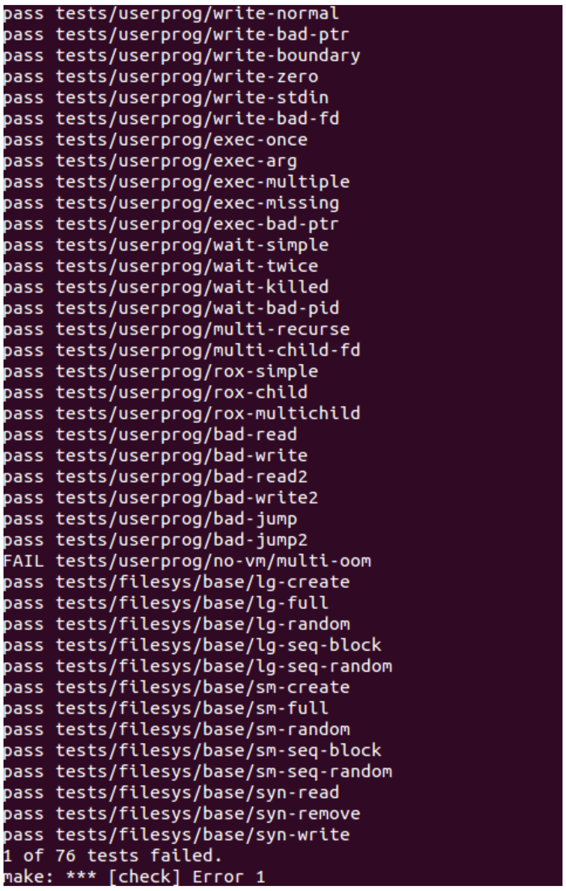
[Modified]

- syscall.c
- process.c
- thread.h

3 Result



```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
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/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
FAIL 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
1 of 76 tests failed.
make: *** [check] Error 1
```

Acknowledgment All source codes are available in HERE.