COMP312005 Operating System, Spring 2019

Project 2 : User Program

**---- GROUP ----**

Hyosang Park <latter2005@gmail.com

Yongho Lee <jhyn207@gmail.com>

Jisu Kim <wltn4511@gmail.com>

**---- PRELIMINARIES ----**

Contribution : 1:1:1

Collaboration : debugging, implementation of code, idea sharing.

Hyosang Park : Argument passing, wait system call, test case debugging.

Yongho Lee : Semaphore, implementation of wait system call

Jisu Kim : implementation of System call and test case debugging

ARGUMENT PASSING

================

**---- DATA STRUCTURES ----**

A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

**char\*\* argv** : to save parsed argument, which will put into a stack by esp.

**int argc** : to save number of arguments parsed.

**int argv\_size** : initially set to 5. If argument is longer than 5, size of argv is doubled by realloc()

**char\*\* pos :** pointer to the argument next to filename. Used for parsing arguments.

**char\*\* token :** variable to temporarily save parsed argument.

**int len :** length of parsed argument to decrease esp.

**---- ALGORITHMS ----**

**A2**: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?

**Description :**

When command is passed to program, strtok\_r() and variable pos is used to parse command. argc(number of argument) is increased every time another token is parsed(per 1 for loop). In every loop, length of parsed argument is checked. If length is over 5(Initial value), size of argv is doubled by realloc().

After filling argv is complete, each argument is pushed into stack in reversed order. Needed number of 0 is inserted for alignment.

After last argument is inserted, Address pointing to each argument, address of argv, and then argc is inserted in the stack too. Lastly, fake return address is inserted.

SYSTEM CALLS

============

**---- DATA STRUCTURES ----**

B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

Description

thread.h

**In struct thread,**

**Struct file \*fd[128]** is added for file descriptor.

**struct semaphore child\_lock;**

**struct semaphore pcb\_lock;**

**struct list child;**

**struct list\_elem child\_elem;**

**int exit\_status**

is added for implementation of wait() system call.

process.c

**In process\_execute()**

**char \*temp**

**char \*rest**

**char copy\_ary[30]**

is added for storing first argument, and rest.

**In start\_process()**

**char \*temp**

is added for storing arguments.

**In process\_wait()**

**struct list\_elem \* list\_pos**

**struct thread \* tmp**

**int exit\_status**

Is added for finding terminated process, and return status of the process.

**In load(), setup\_stack()**

**char \*\*pos** is added as parameter to store arguments next to filename.

syscall.c

**struct lock filesys\_lock** is added for implementation of synchronization in file-related system calls.

**struct file** is added for file-related system calls.

**in \_exec()**

**pid\_t pid** is added for return value of **process\_execute()**

**in \_open(), \_seek(), \_tell(), \_close()**

**struct file \*fp** is added for allocation of file descriptor of opened file, and as parameter of each file-related functions.

**---- ALGORITHMS ----**

B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

Ans:

File descriptors are defined in **struct thread** as struct **file\* fd[128].** When each new file is opened, **\_open()** looks up for available file descriptor number and allocates it to the process. Allocated file descriptor is freed when process terminates. This file descriptor is unique within the single process.

B3: Describe your code for reading and writing user data from the kernel.

Ans:

Reading and writing user data from the kernel is implemented in syscall.c as system call, **\_read()** and **\_write().** Each parameter (fd, buffer, buf\_size) is passed to the fuction from stack page by **esp**. both functions check whether passed parameters are in valid stack area, by **valid\_check().** After, each function checks if **fd** is standard I/O(0 or 1).

In **\_read(),** read from stdin(0) is implemented by **input\_getc().** Other read requests are done by **file\_read()** function.

In **\_write(),** write to stdout(1) is implemented by **putbuf().** Other write requests are done by **file\_write()** function.

B4: Briefly describe your implementation of each system call.

\*Total 13 system calls are implemented in this project.

\*Each parameter of system call is checked by valid\_check(), which checks whether parameter is from valid stack address.

\*Each file-related system call checks whether file is valid or not.

\*Lock is implemented in all file-related system call for synchronization.

**void \_halt(void)**

-halt system call is implemented by function shutdown\_power\_off(), defined in shutdown.c.

**void \_exit(int status)**

-exit system call prints exited process’s name and its exit status. status is inserted into thread structure’s exit status member. After, exit invokes **thread\_exit().**

**pid\_t \_exec(const char \*cmdline)**

-exec system call is implemented by **process\_execute().** Command line is passed as parameter, and parsed into separate arguments in fuction process\_execute(). This function invokes **thread\_create()** to execute process. pid is returned as return value of exec system call.

**int \_wait(pid\_t pid)**

-wait system call is implemented by **process\_wait()**. Each child process of a process is stored as list, and accessed by **list\_pos**. When **process\_wait()** is invoked, it finds child with **pid** in the list. **sema\_down()** and **sema\_up()** is used for waiting child process to safely return it’s exit status before terminated. **\_wait()** returns exit status of terminated child process.

**bool \_create(const char \*file, unsigned initial\_size)**

-create system call is implemented by **filesys\_create()**, defined in filesys.c. it returns 1 on success, 0 on failure.

**bool \_remove(const char \*file)**

-remove system call is implemented by **filesys\_remove().**

it returns 1 on success, 0 on failure.

**int \_open(const char \*file)**

-open system call is implemented by **filesys\_open().**

When **filesys\_open()** is invoked, it checks available file descriptor in thread’s **fd** member. also, if program tries to write on itself, it prevents writing by **file\_deny\_write()** function.

**int \_filesize(int fd)**

-filesize system call is implemented by file\_length() function.

It returns length of file if success.

**int \_read(int fd, void \*buffer, unsigned size)**

-read system call is implemented by **input\_getc(), file\_read().**

read from stdin(0) is implemented by **input\_getc().**

Other read requests are done by **file\_read()** function.

**int \_write(int fd, const void \*buffer, unsigned size)**

-write system call is implemented by **putbuf(),file\_write().**

write to stdout(1) is implemented by **putbuf().**

Other write requests are done by **file\_write()** function.

**void \_seek(int fd, unsigned position)**

-seek system call is implemented by **file\_seek()** function.

**unsigned \_tell(int fd)**

-tell system call is implemented by **file\_tell()** function.

**void \_close(int fd)**

-close system call is implemented by **file\_close()** function.

Before closing file, file descriptor allocated for the file is reset to NULL.