Operating Systems Lab: CS39002

Assignment 6: Extending PintOS to run user programs with arguments (by changing program stack structure) and implementing (mostly file system-related) system calls.

Group 31: Design Document - User Programs

Group Details

Name: Animesh Jain **Roll No:** 18CS10004 **Email ID:** animesh99@iitkgp.ac.in

Name: Abhinav Bohra **Roll No:** 18CS30049 **Email ID:** abhinavbohra@iitkgp.ac.in

PRELIMINARIES

To fix the error - `Pintos doesn't recognize the arguments "-f"`
We made the following changes: -

In utils/pintos changed line number 259 to `/home/<home username>/<pintos dir>/src/userprog/build/kernel.bin`

In utils/Pintos.pm changed line number 362 to `/home/<home username>/<pintos dir>/src/userprog/build/loader.bin`

We referred to this link while solving the issue -
https://stackoverflow.com/questions/20822969/pintos-programming-project-2

TASK 1 - ARGUMENT PASSING

DATA STRUCTURES

No new variables or changes to struct were declared for argument passing.

We changed `*esp = PHYS_BASE;` to `*esp = PHYS_BASE - 12;`
in setup_stack( ) function in userprog/process.c

ALGORITHMS

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

After successfully setting up the stack of the process:

1. Tokenize the command line passed to process_execute( ) function by spaces into arguments the count arguments. Despite this additional loop, pre-counting arguments helps to allocate the exact memory needed without any additional memory.

2. Push tokenized strings into the stack and save their pointers in external array.

3. Do word alignment for faster memory access.

4. Push pointers from the array to stack in ascending order.

5. Push number of arguments into the stack then a dummy return value (0).

To keep arguments in the right order we push their addresses into the stack starting from last argument (in ascending order), so when the main function caller extract them from the stack, they will be popped in the opposite order (Last In First Out).

How do you avoid overflowing the stack page?
We avoid overflowing the stack page by performing a check on the total size of the arguments being passed. If it would overflow the stack page size, we exit.

RATIONALE

A3: Why does Pintos implement strtok_r() but not strtok()?

Pintos implement strtok_r() because in strtok() the point where the last token was found is kept internally by the function to be use on the next call to strtok(). The problem with this is that this will be prone to race condition. Suppose two threads are calling strtok(), there is a possible data race condition where one thread would use the last token held by another thread. This would be incorrect and has potential to crash the kernel.

A4: In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

The kernel does not have to deal with parsing. Instead, the shell will deal with the parsing and error checking before passing the command line to the kernel.

It is safer to do it in the shell itself because it runs in non-kernel mode.

Resolution of path is more convenient (relative paths etc can be used) since shell has more awareness of the high-level execution environment.

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.

```
static struct lock lock_filesys;
```
Using lock to ensure process synch during accessing file system resources

```
#include "threads/vaddr.h"
```
Added threads/vaddr.h file for working with virtual addresses (checking address before syscalls)

```
#include "userprog/pagedir.h"
```
Included userprog/pagedir.h file to use pagedir_get_page function to implement exec syscall

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

File descriptors are unique just within a single process. Each process tracks a list of its file descriptors (list of struct fd, stored in struct thread), as well as its next available fd number.

ALGORITHMS

PSUEDO CODE for syscall_handler:

Step 1: First ensure that the system call argument is a valid address. If not, exit immediately.

Step 2: Get the value of the system call (based on enum in syscall-nr.h) and call corresponding syscall function.

Step 3: Handle WAIT Syscall
-> Fetch arguments from stack : Only one ARG -> esp + 5 (PID of the child process that the current process must wait on.)
-> Call process_wait() function

Step 4: Handle EXIT Syscall
-> Fetch arguments from stack : Only one ARG -> esp + 5 (stores exit status)
-> Set status of current thread = exit status
-> Print the status
-> Terminate the thread using thread_exit()

Step 5: Handle EXEC Syscall
-> Fetch arguments from stack : Only one ARG -> esp + 5 (entire command line text for executing the program)
-> Return if cmdline is NULL
-> Otherwise, lock file_sys to avoid race condition
-> Call process_execute(cmdline), (present in process.c) that parses cmdline and runs the cmd
-> Release lock & return the PID of the thread that is created
-> Store the returned value in eax register

Step 6: Handle WRITE Syscall

-> Fetch arguments from stack : 3 ARGs

-> ARG1 = esp + 5 (File Descriptor)

-> ARG2 = esp + 6 (Pointer to Buffer)

-> ARG3 = esp + 7 (Buffer Size)

-> Lock file_sys to avoid race condition

-> if FD=1 (STDOUT), then write to cosole using putbuf(buffer, size);

-> Release lock & return number of bytes written

-> Otherwise, release the lock & retuen -1

-> Store the returned value in eax register

Step 7: Terminate the program in case of an invalid syscall number