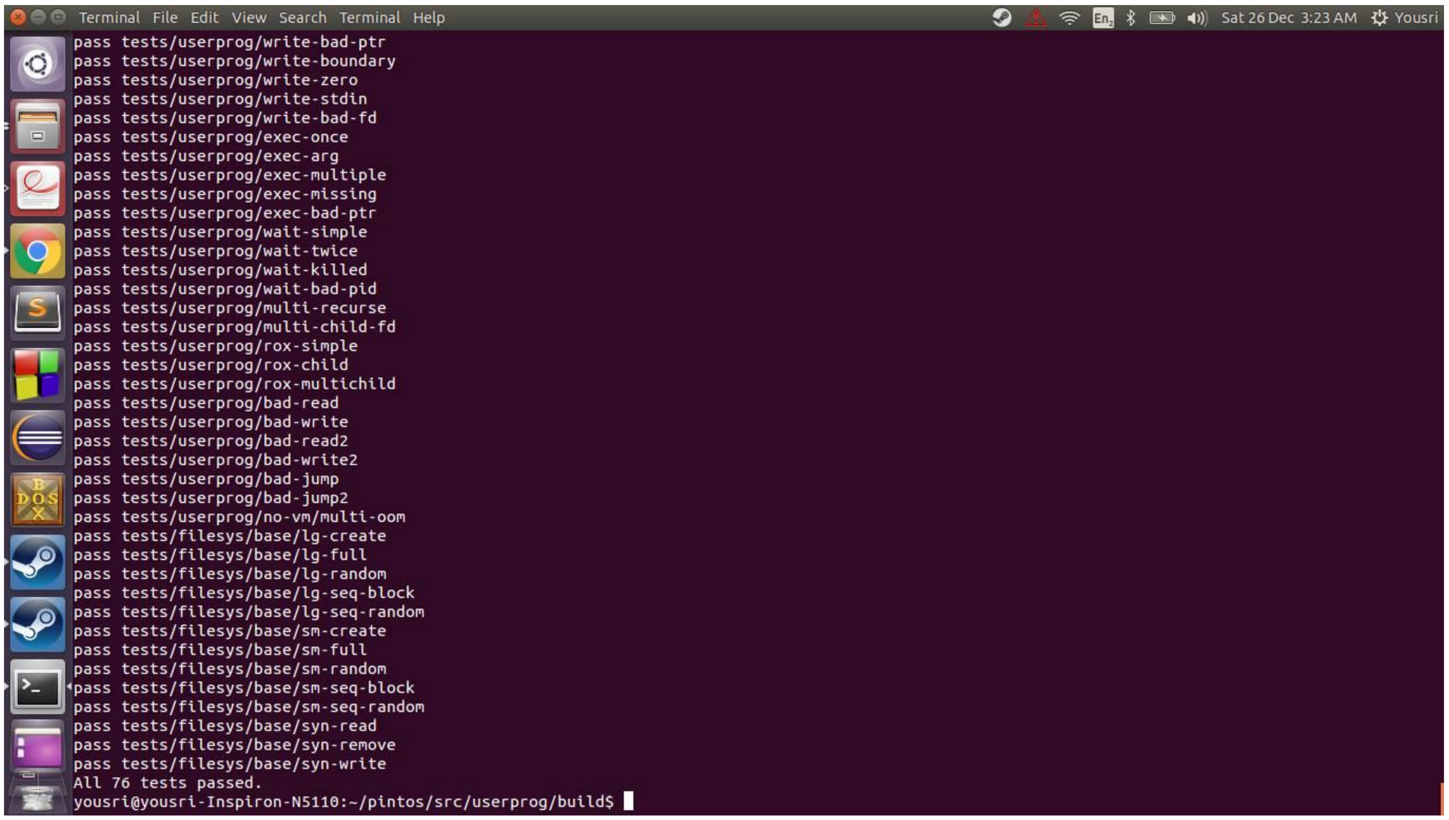
ALEXANDRIA UNIVERSITY



```
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
pass 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
All 76 tests passed.
yousri@yousri-Inspiron-N5110:~/pintos/src/userprog/build$
```

PintOS

PROJECT 2: USER PROGRAMS

CS 333 OPERATING SYSTEMS

Contents

Group Members

Omar Salah Eldin Ebrahim Abdou	(ID: 46)
Ali Mohamed Ali Abdul-Hafez	(ID: 45)
Mohammed Yousri Mohammed Sobhy	(ID: 60)

ARGUMENT PASSING

All of the code that ran under pintos had been a part of the operating system kernel and thereby had full access to privileged parts of the system. But once user programs are allowed to run on the top of the operating system, this is no longer true. Our implementation in this stage extends the thread system developed in the previous stage to allow users to load programs from the disk and run a number of such programs at a time. The thread management functions of pintos are developed into sophisticated process management utilities, which include process creation and termination. Processes in burritos are single threaded. The first process, init, has the process ID 1, and is created by burritos during boot time. All other processes of the system are descendants of the init process.

DATA STRUCTURES

None

ALGORITHMS

- 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?

1) Split the argument line sent from the console into tokens and store these token in an array (argv) then save the count of these tokens in the variable (argc).
2) Hold the tokens' addresses in stack by aid of an array (arr).
3) Set the stack pointer to the phys_base.
4) Traverse through argv from n-1 to 0 , where n is the number of tokens (argc), and for each token subtract its length from the current stack pointer and then store it in the address of the current stack pointer and store this address into (arr).
5) Add a pointer to the first element of the array "argv", and then we add argc to the stack, finally, we add the return address (0).

After adding all the parameters (tokens) we also add pointers of them in the stack and then we add a pointer to the array of pointers to the parameters, in order to avoid stack overflow, parse only the first 100 characters of the line sent from the console.

RATIONALE

- Why does Pintos implement strtok_r() but not strtok()?

The _r versions of functions are reentrant: could be called from multiple threads simultaneously, or in nested loops. Reentrant versions usually take an extra argument, this argument is used to store state between calls instead of using a global variable. The non-reentrant versions often use global state, so if called from multiple threads, it is probably invoking undefined behavior. The program could crash, or worse.

- 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.

In Unix-like systems the shell do most of the hard work separating the parameters finding the file to execute and running it in a new thread, then the kernel has less work to do . Furthermore, will filter any erroneous commands from getting to the kernel this will make the kernel more stable.

SYSTEM CALLS

System calls are programmer's functional interface to the kernel. They are subroutines that reside inside the kernel, and support basic system functions such as halt, exit, exec, wait, create, remove, open, close, read, write, seek and tell.

DATA STRUCTURES

char *prog_name; ➔ Name of the executable file (command name) for printf
struct list children; ➔ the list of the children processes
struct list files; ➔ the list of open files
int next_fd; ➔ the next file descriptor number
struct file *executable; ➔ the executable files of the this process

in process.h

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

File struct that traverses through files list in each thread

fd:	file descriptor
file:	file pointer
elem:	pointer in files list

```
struct process_pid {
        int pid;
        struct list_elem elem;
};
```

Process struct that traverses through children list in each thread

pid:	ID of that process
elem:	pointer in children list

#define LOAD 0 ➔ Indicate synchronization in exec system call
#define WAIT 1 ➔ Indicate synchronization in wait system call

```
struct wait_elem {
        struct semaphore wait;
        int mode;
        int tid;
        int returned;
        struct list_elem elem;
};
```

wait struct to traverse the wait_list

wait:	semaphore to handle synchronization
mode:	LOAD or WAIT

tid: the ID of the child thread
returned: the return value (the statue in exit in mode WAIT or the pid in successful load operation (-1 fail) in exec in mode LOAD)
elem: pointer to wait_list

struct list wait_list;
list that handles receiving and sending the request and the synchronization of (exec and load) and (wait and exit)

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

Each process has a variable called next_fd which is initially equal 2 and contain the file descriptors of the next file, each files get it's descriptors when open system call is invoked then the process save the file in the files list which has the opened files and get the file descriptors file descriptors is unique within a single process.

ALGORITHMS

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

Validate the pointers using get_user function then syscall_handler handles the calls to go to the specific functions, we handle read operation so if STDIN (read from keyboard with input_getc()) or STDOUT (exit (-1)) or read from other opened file by that thread and handle write operation if STDOUT (write the given buffer using putbuf()) or STDIN (exit (-1)) or from other opened file by that thread.

- Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to pagedir_get_page()) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?

If a system call causes a full page of data to be copied, the greatest possible number of inspection could be 2, and the least number could be 1, depending on whether the data lies on two different pages when copying 2 bytes of data, the situation in is similar. the probability of checking two pages in this case is far lower than the above case. A different approach is not to check the address for read and write, but let the operations continue until there is a page fault occurring then we can implement handler to deal with the user page fault situation properly. In this way, if the address is valid, then 0 inspection is needed. For the reading of 2 bytes of data, which would happen much more frequently And if an invalid operation is attempted, the system would write data to the valid address until the invalid addresses.

- Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues.

check the arguments pointers in each system call to check it's a valid user address and we release all the resources in process_exit() which is the only way to terminate the process , so when the system call fail we call thread_exit() which call process_exit() to release the resources

SYNCHRONIZATION

- The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?

Ensure this by making the parent wait until the load of its child is completed in process_execute() the parent process add to wait list that he is wait for the child to load and sleep if the child didn't complete or return if the child is completed and when the child complete his load he is up is parent and put his PID in the wait list.

- Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?

P calls wait(C) before C exits
P add the request in the wait list that P wait for C and want to get the exit statue and sleep then when C exited we release its resources and wake up P

P calls wait(C) after C exits
when C exited we release its resources and add the exited status in the wait list then P wait for C he will find the exited status of the child so he won't sleep and return

P terminates without waiting, before C exits
nothing special will happen except when C exit it will leave its return value in wait list but it will never used

P terminates without waiting, before C exits
C exit it will leave its return value in wait list but P won't wait for it

RATIONALE

- Why did you choose to implement access to user memory from the kernel in the way that you did?

Because it is the easier of the two options described in the assignment document. It also is somewhat more straightforward logically to get the test out of the way to begin with.

- What advantages or disadvantages can you see to your design for file descriptors?
- Advantages:
  Because each thread has a list of its file descriptors, there is no limit on the number of open file descriptors (until we run out of memory).
- Disadvantages:
  1) There exist many duplicate file descriptor structs, for stdin and stdout
  2) Each thread contains structs for these fds. Accessing a file descriptor is O(n), where n is the number of file descriptors for the current thread (have to iterate through the entire fd list). Could be O(1) if they were stored in an array.
- The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

We stick to the default implementation in mapping tid_t to pid_t. If tid_t is not mapped with pid_t, that will support a process running multiple threads, which is a feature not supported by Pinots. So we did not change this part.

The End