**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY  
SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING**

**Ảnh có chứa biểu tượng

Mô tả được tạo tự động**

**ET4291E – Operating System**

**ASSIGNMENT**

|  |  |  |
| --- | --- | --- |
| **Instructor :** | Assoc. Prof. Pham Van Tien | |
| **Course :** | ET4291E | |
| **Class :** | 144074 | |
| **Name:** | Nguyen Duc Phuc Hoang | 20210385 |

**HÀ NỘI – 2023**

Table of Contents

[1. Introduction 2](#_Toc153058211)

[1.1. Pintos 2](#_Toc153058212)

[1.2. Threads 3](#_Toc153058213)

[1.3. User Program 3](#_Toc153058214)

[2. Subproject 1: Advanced scheduler (Threads) 3](#_Toc153058215)

[2.1. Background 3](#_Toc153058216)

[2.2. Implementation 4](#_Toc153058217)

[2.3. Test run result 4](#_Toc153058218)

[3. Subproject 2: Argument Passing (User Program) 4](#_Toc153058219)

[3.1. Background 4](#_Toc153058220)

[3.2. Implementation 5](#_Toc153058221)

[3.3. Test run result 5](#_Toc153058222)

[4. Conclusion 5](#_Toc153058223)

# 1. Introduction

## 1.1. Pintos

## Pintos is a simple OS for the 80x86 architecture that supports basic kernel threads, user program loading and execution, file systems, and other core OS functionality, but in a simplified manner.

## The main folders in pintos/src and their purpose are:

## "threads" - contains code for the kernel threads and base kernel functions

## "userprog" - contains code to load and run user programs

## "vm" - a mostly empty folder to later add virtual memory implementation

## "filesys" - contains code for a simple file system

## "devices" - contains code to interface with I/O devices like keyboard, timer, disk, etc.

## "lib" - implements a subset of the C standard library used by both the Pintos kernel and user programs running on top of it. Can be included in code via #include <...>

## "tests" - tests for each project assignment

## "examples" - sample user programs that can run on Pintos starting with project 2.

## 1.2. Threads

## The goal of this pintos assignment is to enhance the existing minimal thread functionality to better understand synchronization challenges that arise from concurrent execution and shared resources.

## I will be working mostly in the threads directory, with some work in devices as well. Compilation takes place in threads.

## Proper synchronization techniques are crucial to solve these kinds of problems correctly. Disabling interrupts can technically address synchronization issues by eliminating concurrency, but has major downsides. Instead, pintos provides synchronization primitives like semaphores, locks, condition variables that should form the core solutions.

## Pintos already implements:

## Thread creation/destruction

## Basic scheduler to switch between active threads

## Synchronization primitives for locking shared resources

## My job will be to effectively utilize and expand these existing components to handle synchronization issues stemming from concurrent thread access.

## 1.3. User Program

The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this subproject, I will enable programs to interact with the OS.

I will be working out of the userprog directory for this assignment, but I will also be interacting with almost every other part of Pintos

Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. A user program can only access its own user virtual memory. An attempt to access kernel virtual memory causes a page fault, handled by page\_fault() in userprog/exception.c, and the process will be terminated. Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process. However, even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault. Every user program will page fault immediately until argument passing is implemented.

# 2. Subproject 1: Advanced scheduler (Threads)

## 2.1. Background

In this project, I have to implement a multilevel feedback queue scheduler similar to the 4.4BSD scheduler to reduce the average response time for running jobs.

This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. At any given time, the scheduler chooses a thread from the highest-priority non-empty queue. If the highest-priority queue contains multiple threads, then they run in "round robin" order.

In summary:

* In every fourth tick, recalculate the priority of all threads:

priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)

* In every clock tick, increase the running thread’s recent\_cpu by 1.
* In every second, update every thread’s recent\_cpu:

recent\_cpu = (2\*load\_avg) / (2\*load\_avg + 1) \* recent\_cpu + nice

with load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads

Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation.

Pintos does not support floating-point arithmetic in the kernel, because it would complicate and slow the kernel. Real kernels often have the same limitation, for the same reason. This means that calculations on real quantities must be simulated using integers.

## 2.2. Implementation

## 2.2.1: devices/timer.c

1. calls the following additional functions related to thread scheduling :

* thread\_mlfqs\_increase\_recent\_cpu\_by\_one() - Increments the current thread's recent CPU time by 1 to track CPU usage.
* thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu() - Called every TIMER\_FREQ ticks to update the load average and recent CPU values used by the multi-level feedback queue scheduler (MLFQS).
* thread\_mlfqs\_update\_priority() - Updates the current thread's priority according to the MLFQS priority calculation formula. This is called every 4th tick.
* thread\_foreach(blocked\_thread\_check, NULL) - Iterates through blocked threads and calls the blocked\_thread\_check function on each. This seems to check if blocked threads should be unblocked.
* thread\_tick() - Common thread scheduling tick activity is done at the end of the handler. ( the source code just have thread\_tick)

## A computer screen shot of a program code Description automatically generated

1. timer\_sleep() implementation:

* new Code calculates timer ticks to sleep, disables interrupts, sets thread sleep values, blocks the thread, and re-enables interrupts :

Checks if sleep ticks <= 0, returns immediately if so.

Disables interrupts.

Sets the current thread's ticks\_blocked value to the sleep duration.

Calls thread\_block() to block the current thread.

Re-enables interrupts.

## A screen shot of a computer code Description automatically generated

## source code busy waits in a loop until the desired ticks have elapsed.

## 2.2.2: Add fixed\_point.h:

## 

## 2.2.3: Change in synch.c:

## 

## 2.2.4: threads/thread.c

1. In summary, new code  adds MLFQS scheduling, a priority scheduler, yield calls, and some additional optimizations around thread blocking.

New code has additional functions for multi-level feedback queue scheduling (MLFQS):

* thread\_mlfqs - a global variable to control whether MLFQS is enabled:
* thread\_set\_nice() and thread\_get\_nice() - to set and get thread nice values:

A screen shot of a computer code

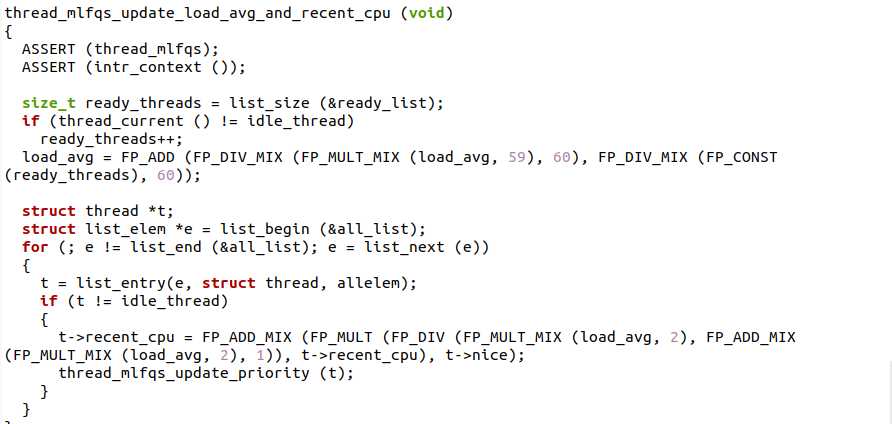
Description automatically generated

* thread\_get\_recent\_cpu() and thread\_get\_load\_avg() - to calculate CPU time and load average

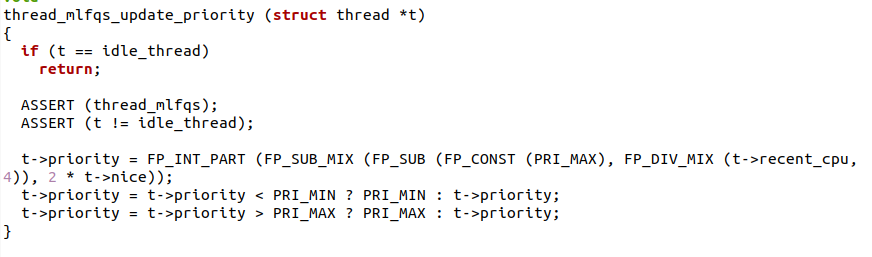
A screenshot of a computer program

Description automatically generated

* thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu() - to periodically update load avg and recent CPU time



* thread\_mlfqs\_update\_priority() - to recalculate thread priorities based on nice, recent CPU, and load avg



* thread\_mlfqs\_increase\_recent\_cpu\_by\_one() - to increment recent CPU time for the running thread

## A computer code with black text Description automatically generated with medium confidence

2. New code uses thread priorities and a priority scheduler instead of round robin scheduling. This includes:

* Storing priorities in struct thread and using them to order threads
* thread\_set\_priority() and thread\_get\_priority()

A computer screen shot of a computer code

Description automatically generated

* list\_insert\_ordered() and thread\_cmp\_priority() to order ready list

3. New code has a thread\_unblock() optimization to insert threads in priority order.

A computer code with black text

Description automatically generated

4. New code has a thread\_ticks\_blocked counter and blocked\_thread\_check() to unblock threads after a certain time.

## A computer screen shot of a computer code Description automatically generated

## 2.2.6: threads/thread.h:

1. New code has additional includes for synch.h and fixed\_point.h.
2. New code has additional thread struct members:

ticks\_blocked - to record time blocked

lock\_waiting - lock thread is waiting for

nice - niceness value

recent\_cpu - recent CPU time

3. New code has a bool thread\_mlfqs - to control mlfqs scheduler.

4. New code has additional thread functions related to the mlfqs scheduler:

blocked\_thread\_check

thread\_cmp\_priority

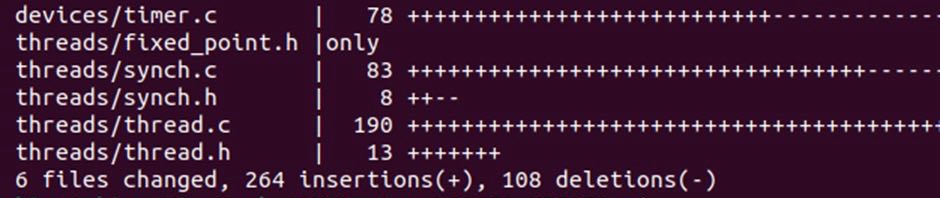
thread\_mlfqs\_increase\_recent\_cpu\_by\_one

thread\_mlfqs\_update\_priority

thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu

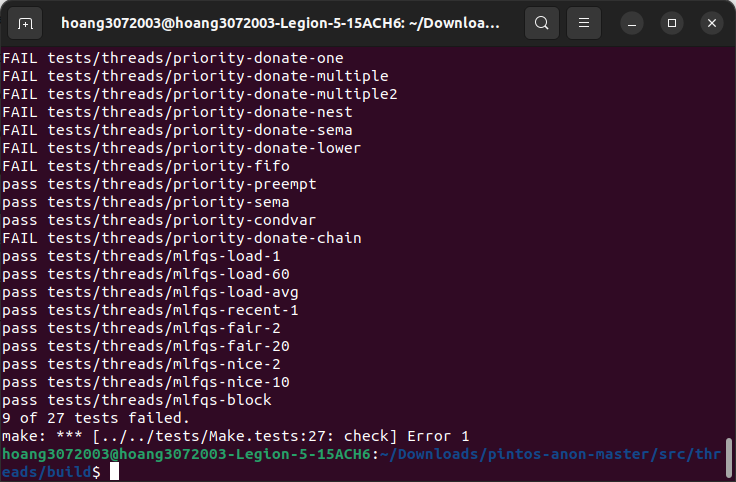
So in summary, New code contains extensions to support the multi-level feedback queue scheduler, including additional thread struct members, a control bool, and scheduler related functions. The main thread struct and basic functionality remains the same between the two snippets.

**Code changed summary:**



## 2.3. Test run result

* **How to run test:** Invoke make from theads folder to create build directory, then run make check from build.
* **Result:**



As we can see, 16/27 tests have been passed, including all the required test of this subproject and except for priority donation tests.

# 3. Subproject 2: Argument Passing (User Program)

## 3.1. Background

Currently, process\_execute() does not support passing arguments to new processes. I implement this functionality, by extending process\_execute() so that instead of simply taking a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, process\_execute("grep foo bar") should run grep passing two arguments foo and bar.

Within a command line, multiple spaces are equivalent to a single space, so that process\_execute("grep foo bar") is equivalent to our original example. I limit the arguments to those that will fit in a single page (4 kB). (There is an unrelated limit of 128 bytes on command-line arguments that the pintos utility can pass to the kernel.)

In this project, I also need to interface to the file system code, because user programs are loaded from the file system and many of the system calls you must implement deal with the file system. However, the focus of this project is not the file system, so pintos have provided a simple but complete file system in the filesys directory. On the other hand, I have to face with these limitations:

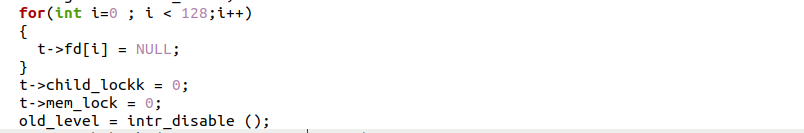
* No internal synchronization. Concurrent accesses will interfere with one another.
* File size is fixed at creation time. The root directory is represented as a file, so the number of files that may be created is also limited.
* File data is allocated as a single extent, that is, data in a single file must occupy a contiguous range of sectors on disk. External fragmentation can therefore become a serious problem as a file system is used over time.
* No subdirectories.
* File names are limited to 14 characters.
* A system crash mid-operation may corrupt the disk in a way that cannot be repaired automatically. There is no file system repair tool anyway.

## 3.2. Implementation

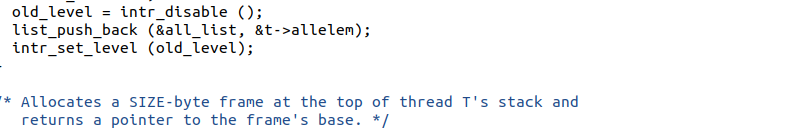
## 3.2.1: threads/thread.c:

1. New code has additional initialization in init\_thread():

It initializes the fd array to NULL and It initializes two new members child\_lockk and mem\_lock to 0:



1. new code disables interrupts around pushing the thread onto the all\_list in init\_thread():



**3.2.2: threads/thread.h:**

 In summary, new code has added a number of new members to the thread struct to support user programs functionality like process control, synchronization, and file descriptors. The conditional compilation allows pagedir to be included only if USERPROG is defined

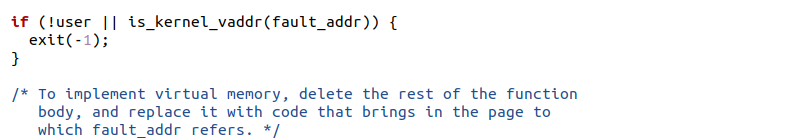
1. In New code, there are additional members added to the thread struct for userprog support:

* int child\_lockk
* int mem\_lock
* struct semaphore\* child\_lock
* struct list child
* struct list\_elem child\_elem
* int exit\_status
* struct file\* fd[128]

1. New code has #ifdef USERPROG conditional compilation around the pagedir member,
2. original code just has the pagedir member, while new code  has additional userprog members listed above.

**3.2.4: exception.c:**

1. the addition of the following code in new code:



This checks if the page fault occurred in kernel virtual address space (by calling is\_kernel\_vaddr()) or if the fault occurred while executing kernel code (user == false).

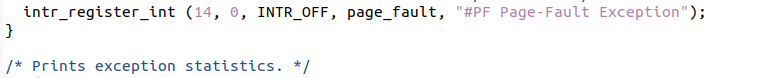
If either of those are true, it calls exit(-1) to terminate the process. This prevents the kernel from trying to handle page faults in its own address space, since those indicate a kernel bug.

1. new code includes an additional header file:

#include "threads/vaddr.h"

This provides the is\_kernel\_vaddr() function used to check if the fault address is in kernel space.

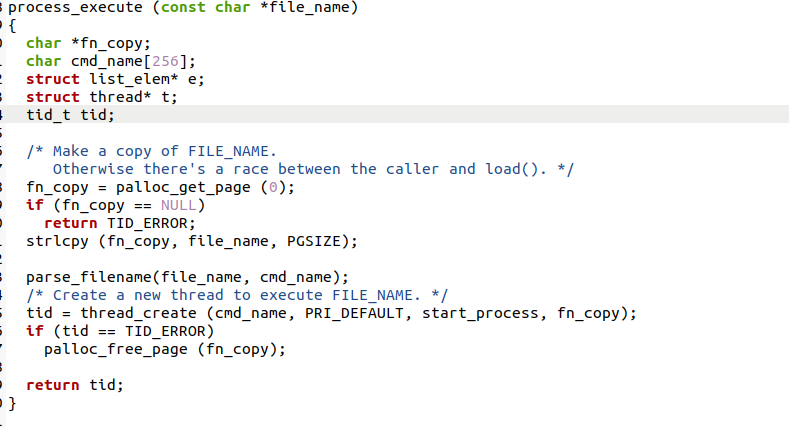
1. new code registers the page fault handler with interrupts disabled:



**3.2.4: process.c:**

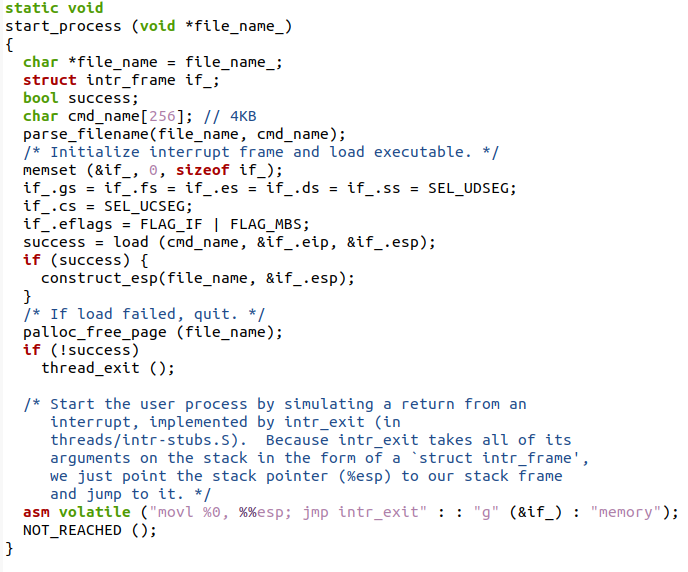
1. process\_execute()

* new code parses out the executable name from the command line into cmd\_name.
* Creates thread using parsed executable name instead of full command line.



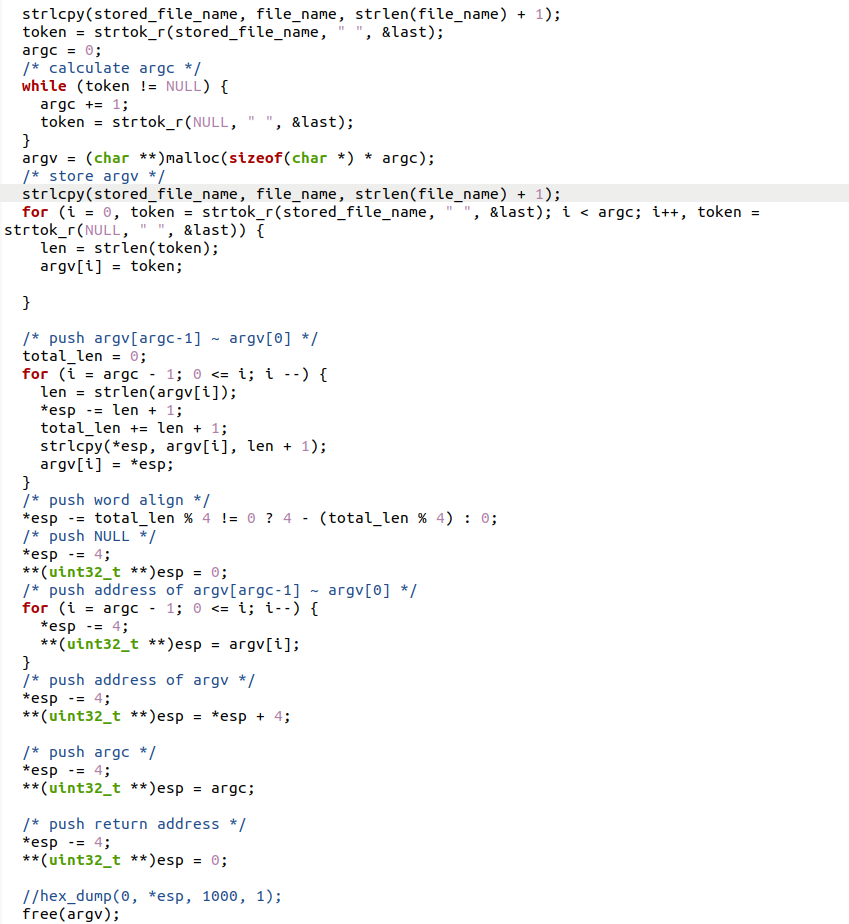
2. start\_process()

* new code parses filename again into cmd\_name.
* Calls construct\_esp() after load() to set up stack.
* Passes cmd\_name to load() instead of file\_name.



1. construct\_esp()

* Parses command line into argc and argv.
* Allocates memory for argv strings.
* Loops to push argv strings onto stack.
* Loops to push argv pointers onto stack.
* Pushes argc, return address, NULL.
* Sets esp to point to constructed stack.

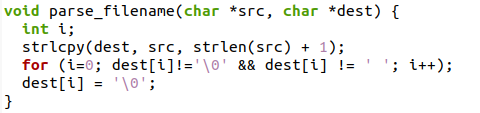


1. load()

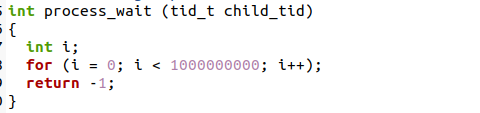
* Both take filename as argument but new code passes cmd\_name.
* Both load executable similarly.

1. Other differences

* new code has function prototypes for parse\_filename and construct\_esp.



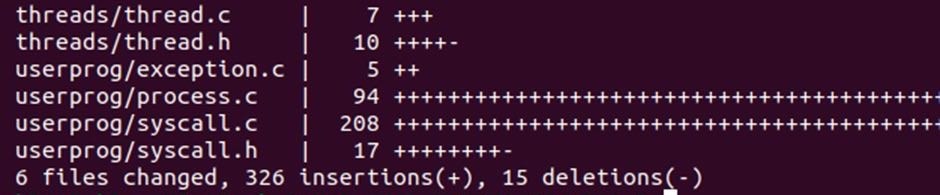
* new code has a dummy process\_wait().



* Minor spacing and formatting differences.

So in summary, new code does much more work to parse arguments, allocate memory, set up the stack properly with argc/argv, before jumping to the newly loaded executable.

**Code changed summary:**



## 3.3. Test run result.

* **How to run test:** for example, I pass ‘echo x y z’ with x, y, z are the arguments passed to run echo

1. Invoke make from userprog folder to create build directory
2. Invoke make from examples folder to create executable file of echo.c
3. Navigate to the build directory of userprog, then run these commands to create a disk with a file system partition, format the file system, copy the echo program into the new disk:

pintos-mkdisk filesys.dsk --filesys-size=2

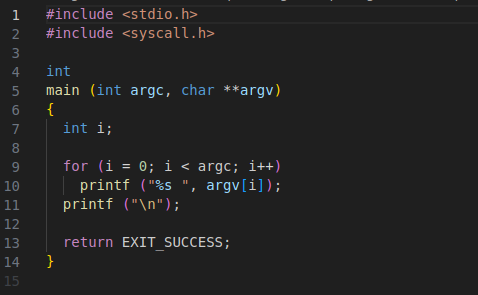
pintos -f -q

pintos -p ../../examples/echo -a echo -- -q

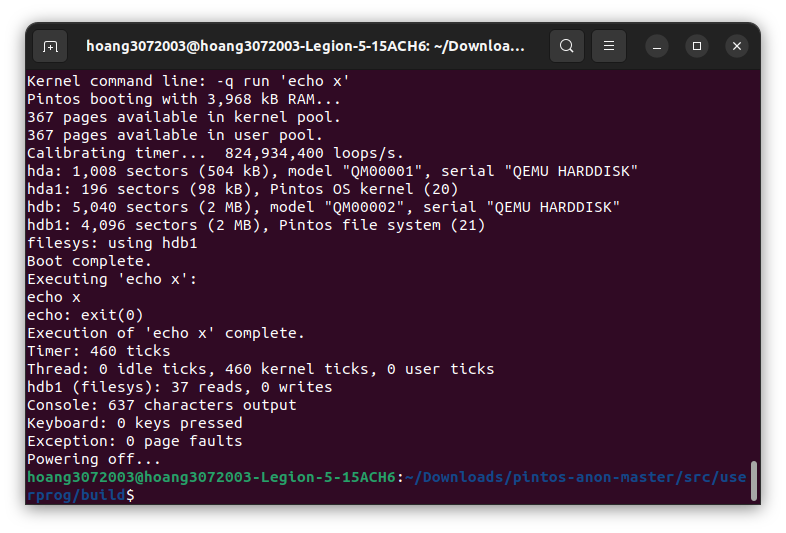
1. Run pintos -q run 'echo x y z' to execute the code inside echo.c

* **Result:**

This is echo.c:



And the result after running all of the above commands:



# 4. Conclusion

Through the Pintos project, I have gained valuable insights and acquired a wealth of knowledge of Advanced scheduler, Arguments passing and many skills related to operating systems and programming techniques. This experience has allowed me to deepen my understanding of the subject matter and hone various essential skills. As a result, I have not only expanded my knowledge base but also developed critical skills crucial for academic and practical applications.

Project’s github repository: https://github.com/hoang3072003/Hoang\_OS.git