**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 Anh Duy | 20210274 |

**HÀ NỘI – 2023**

Table of Contents

[1. Introduction 4](#__RefHeading___Toc431_3784917898)

[1.1. Pintos 4](#__RefHeading___Toc433_3784917898)

[1.2. Threads 4](#__RefHeading___Toc435_3784917898)

[1.3. User Program 4](#__RefHeading___Toc437_3784917898)

[2. Subproject 1: Advanced scheduler (Threads) 5](#__RefHeading___Toc439_3784917898)

[2.1. Background 5](#__RefHeading___Toc441_3784917898)

[2.2. Implementation 6](#__RefHeading___Toc443_3784917898)

[2.3. Test run result 6](#__RefHeading___Toc445_3784917898)

[3. Subproject 2: Argument Passing (User Program) 6](#__RefHeading___Toc447_3784917898)

[3.1. Background 6](#__RefHeading___Toc449_3784917898)

[3.2. Implementation 7](#__RefHeading___Toc451_3784917898)

[3.3. Test run result 7](#__RefHeading___Toc453_3784917898)

[4. Conclusion 9](#__RefHeading___Toc455_3784917898)

# 1. Introduction

## 1.1. Pintos

Pintos is a simple operating system framework for the 80x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way.

Important folders of pintos inside pintos/src are:

1. “threads”: source code for the base kernel
2. “userprog”: source code for the user program loader
3. “vm”: an almost empty directory for implementation of virtual memory
4. “filesys”: source code for a basic file system
5. “devices”: source code for I/O device interfacing: keyboard, timer, disk, etc.
6. “lib”: an implementation of a subset of the standard C library. The code in this directory is compiled into both the Pintos kernel and, starting from project 2, user programs that run under it. In both kernel code and user programs, headers in this directory can be included using the #include <...> notation.
7. “tests”: tests for each project
8. “examples”: example user programs for use starting with project 2.

## 1.2. Threads

In this assignment, pintos gives a minimally functional thread system. Our job is to extend the functionality of this system to gain a better understanding of synchronization problems.

I will be working primarily in the threads directory for this assignment, with some work in the devices directory on the side. Compilation is done in the threads directory.

*Synchronization:* Proper synchronization is an important part of the solutions to these problems. Any synchronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there's no possibility for race conditions. Pintos uses semaphores, locks, and condition variables to solve the bulk of synchronization problems.

Pintos already implements thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and optimization barriers).

## 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

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:

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

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

1. In every clock tick, increase the running thread’s recent\_cpu by 1.
2. 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

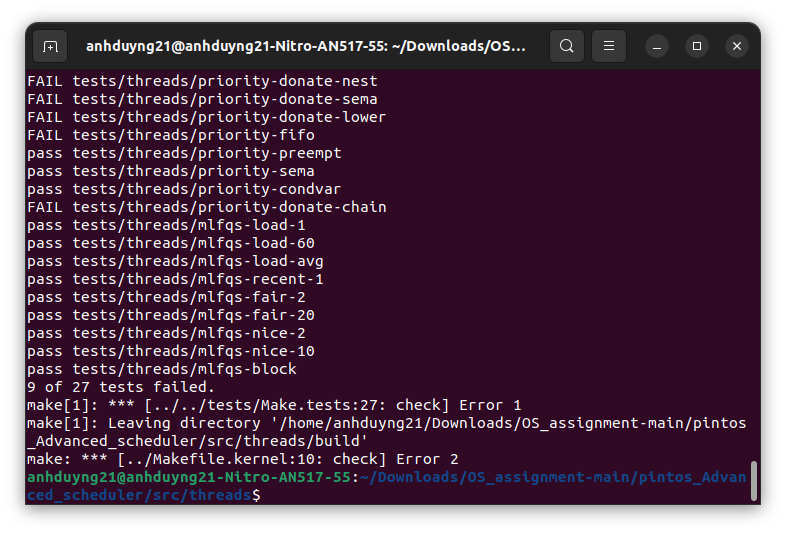
Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation.

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

1. Add nice and recent\_cpu variables to thread structure.
2. Calculate priority by implement thread\_mlfqs\_update\_priority function using recent\_cpu and nice value
3. The thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu function is to recalculate recent\_cpu, load\_avg and priority of all threads.
4. Implement thread\_mlfqs\_increase\_recent\_cpu\_by\_one function to increase recent\_cpu by 1.
5. Customized getter and setter for priority, nice, recent\_cpu and load\_avg

## 2.3. Test run result



FAILED 9/27, ONLY TEST ADVANCED SCHEDULER

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

## 3.1. Background

Function process\_execute() does not support passing arguments to new processes. I extend process\_execute() so that instead of simply taking a program file name as its argument, it divides it into words at spaces. Which 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).

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. I also need to interface to the file system code, and I have to face with these limitations:

1. No internal synchronization. Concurrent accesses will interfere with one another.
2. 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.
3. 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.
4. No subdirectories.
5. File names are limited to 14 characters.
6. 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

* Implement parse\_filename function to extract the first argument from source string to destination string.
* In process\_execute function, parse file\_name using parse\_filename function and forward the first token as name of new process to thread\_create function.
* In start\_process function, parse file\_name and save the tokens on user stack of new process by calling construct\_esp function
* Implement construct\_esp function to set up an user stack for processing the arguments

## 3.3. Test run result

I pass ‘echo x y z’ with x, y, z are the arguments passed to run echo with these instructions below:

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:

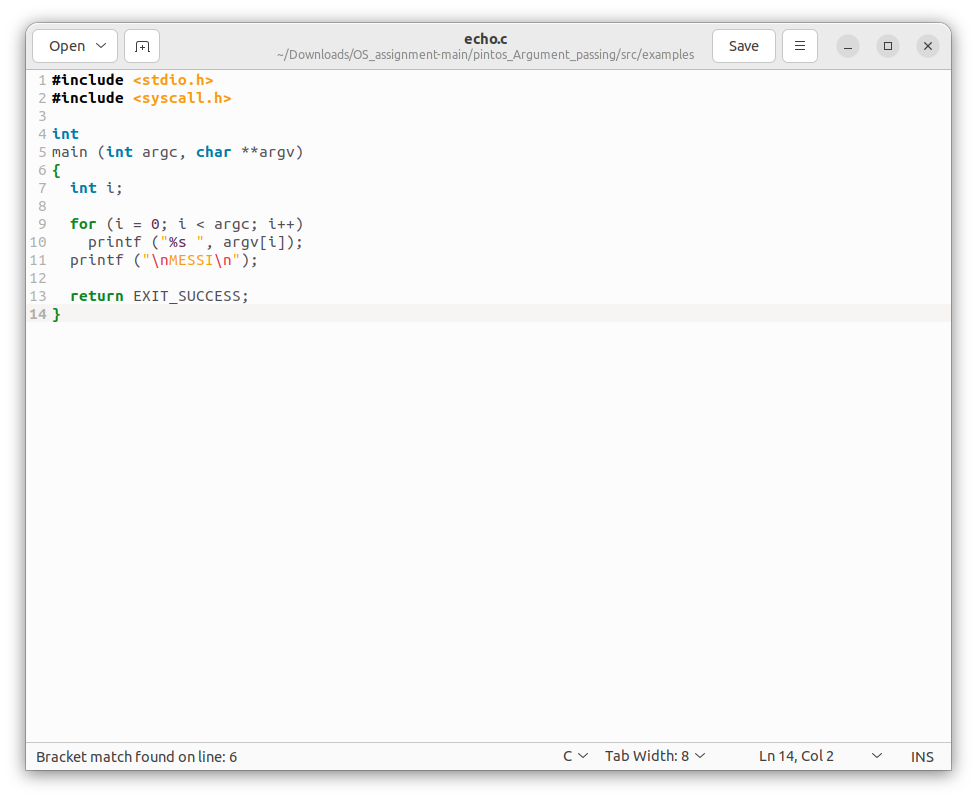
pintos-mkdisk filesys.dsk --filesys-size=2

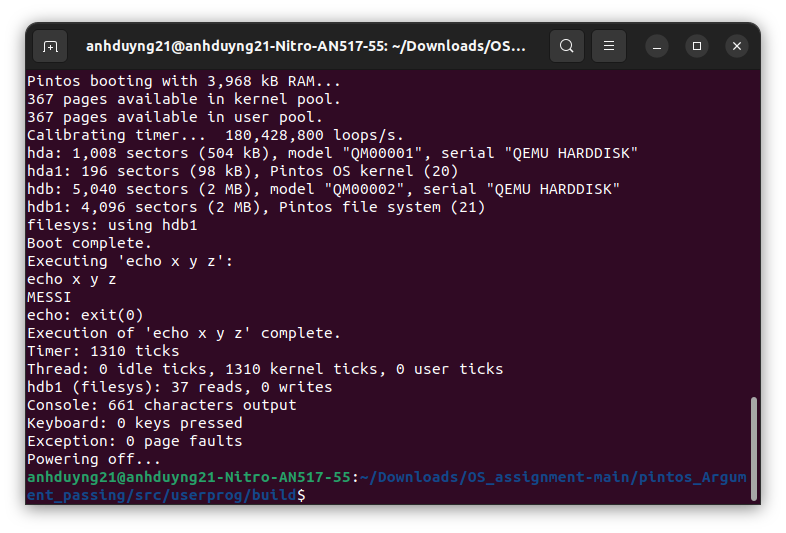
pintos -f -q

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

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

The results are compared to the file echo.c below:





# 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. 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/sentairider0/os-assignment-subsproject.git