

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

o0o



Project Report

Operating System

Argument Passing (User Programs) and Paging (Virtual Memory)

Instructor : Pham Van Tien

Students : Nguyen Thu Huong - 20210423

Class : CTTT ET-E16 01 K66

Hanoi - 2023

Contents

| | | |
|----------|----------------------------------------|-----------|
| 1 | INTRODUCTION | 3 |
| 1.1 | Purpose | 3 |
| 1.2 | About Pintos | 3 |
| 1.3 | Requirements | 4 |
| 2 | Argument passing (User program) | 5 |
| 2.1 | Describe and objectives | 5 |
| 2.2 | Algorithms and codes | 5 |
| 2.3 | Testing | 7 |
| 3 | Paging (Virtual memory) | 8 |
| 3.1 | Describe and objectives | 8 |
| 3.2 | Data structure | 8 |
| 3.2.1 | Page table | 8 |
| 3.2.2 | Frame table | 9 |
| 3.3 | Algorithms and codes | 10 |
| 3.4 | Testing | 11 |
| 4 | CONCLUSION | 12 |

Chapter 1

INTRODUCTION

1.1 Purpose

By completing this Pintos subproject, I aim to solidify my understanding of multiprogramming systems, develop valuable skills in operating system design and implementation, gain a strong foundation for further exploration and contribution to the field of operating systems. This project serves as a valuable stepping stone for anyone interested in pursuing a career in operating systems or exploring the intricacies of how modern computer systems function.

1.2 About Pintos

Pintos is a simple, instructional operating system framework for the x86 instruction set architecture. It is designed to be used in undergraduate operating systems courses, and it provides a variety of features that allow students to learn about the fundamentals of operating systems design and implementation.

Pintos is a complete operating system, with a kernel, user space, and device drivers. It supports a variety of hardware devices, including the keyboard, mouse, and network interface card. It also supports a variety of file systems, including FAT, ext2, and ext3.

Pintos is a modular system, which allows students to experiment with different operating system features. The kernel is divided into a number of independent modules, and students can choose which modules to include in their operating system. This allows students to learn about the different components of an operating system and how they interact.

Let's take a look at what's inside. Here's an overview of what's in `src/`:

- `threads/`: Source code for the base kernel, which you will modify starting in project 1.

- `userprog/`: Source code for the user program loader, which you will modify starting with project 2.
- `vm/`: An almost empty directory. You will implement virtual memory here in project 3.
- `filesystem/`: Source code for a basic file system. You will use this file system starting with project 2, but you will not modify it until project 4.
- `devices/`: Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in project 1. Otherwise you should have no need to change this code.
- `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. You should have little need to modify this code.
- `include/lib/kernel/`: Parts of the C library that are included only in the Pintos kernel. This also includes implementations of some data types that you are free to use in your kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the `#include <...>` notation.
- `include/lib/user/`: Parts of the C library that are included only in Pintos user programs.
- `tests/`: Tests for each project. You can modify this code if it helps you test your submission, but we will replace it with the originals before we run the tests.
- `examples/`: Example user programs for use starting with project 2.
- `include/`: Source code for the header files (`*.h`).

1.3 Requirements

- The final report (both soft and hardcopy) that describes my selected assignments, objectives.
- Explanation of my modified/added source codes, linking them to the aforementioned objectives.
- Screenshots to demonstrate that I successfully run my programs.
- Github repositories of my complete software that is runnable:
<https://github.com/nguyenthuhuog/OS-subproject.git>

Chapter 2

Argument passing (User program)

2.1 Describe and objectives

Currently, `process_execute()` does not support passing arguments to new processes. 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. You can impose a reasonable limit on the length of the command line arguments. For example, you could 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.)

Here are the objectives:

- Parsing the command line
- Setting up argument in stack

2.2 Algorithms and codes

- Parsing the command line:
 - Extracts the program filename and arguments from the command line string.
 - Uses function `strtok_r()` to tokenize the string based on whitespace as delimiters.
 - Just like `strtok()` function in C, `strtok_r()` does the same task of parsing a string into a sequence of tokens. `strtok_r()` is a reentrant version of `strtok()`, hence it is thread safe.

```
token = strtok_r(file_name_copy, " ", &saveptr)
```

– Stores the arguments in an array of strings argv.

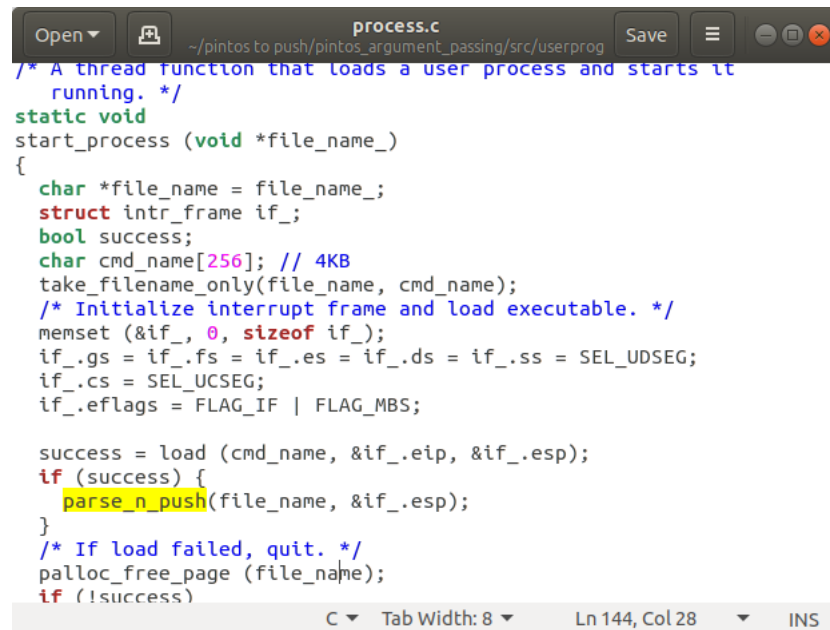
- Argument setup:

- Pushes the addresses of each argument string onto the stack in reverse order, with argv[0] is the program name.
- Push word align to make sure the next item is in different page.
- Push NULL pointer sentinel, ensures that argv[argc], the last element of array argv is a null pointer, as required by the C standard.
- Push address of each argument string in argv (also in reverse order),
- Push address of argument strings array(argv)
- Pushes the number of arguments (argc) onto the user program's stack.
- Push fake return address to maintain the same structure as other stack frame

- Modifying start_process() (userprog/process.c):

- The start_process function needs to be updated to handle arguments.
- If load process successfully, start parsing and pushing arguments (function parse_n_push defined in line 24 same file).

```
void parse_n_push(char *file_name, void **esp)
```



```
process.c
~/pintos to push/pintos_argument_passing/src/userprog
Save

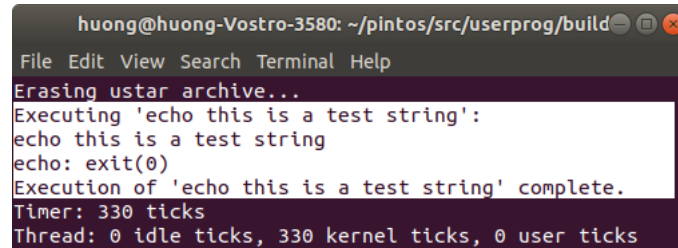
/* A thread function that loads a user process and starts it
   running. */
static void
start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;
    char cmd_name[256]; // 4KB
    take_filename_only(file_name, cmd_name);
    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
    if_.cs = SEL_UCSEG;
    if_.eflags = FLAG_IF | FLAG_MBS;

    success = load (cmd_name, &if_.eip, &if_.esp);
    if (success) {
        parse_n_push(file_name, &if_.esp);
    }
    /* If load failed, quit. */
    palloc_free_page (file_name);
    if (!success)
```

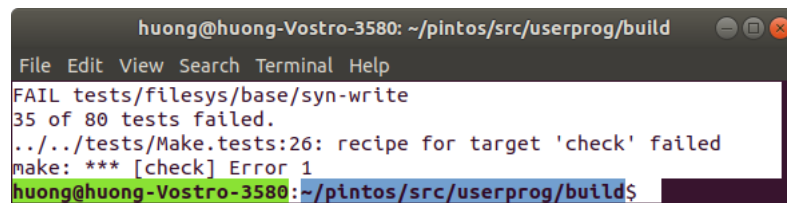
C Tab Width: 8 Ln 144, Col 28 INS

2.3 Testing

The test program `echo.c` take "this is a test string" as arguments. It supposes to print out "echo" and input arguments. As it shows here, the program works.



```
huong@huong-Vostro-3580: ~/pintos/src/userprog/build
File Edit View Search Terminal Help
Erasing ustar archive...
Executing 'echo this is a test string':
echo this is a test string
echo: exit(0)
Execution of 'echo this is a test string' complete.
Timer: 330 ticks
Thread: 0 idle ticks, 330 kernel ticks, 0 user ticks
```



```
huong@huong-Vostro-3580: ~/pintos/src/userprog/build
File Edit View Search Terminal Help
FAIL tests/filesys/base/syn-write
35 of 80 tests failed.
../../tests/Make.tests:26: recipe for target 'check' failed
make: *** [check] Error 1
huong@huong-Vostro-3580: ~/pintos/src/userprog/build$
```

Chapter 3

Paging (Virtual memory)

3.1 Describe and objectives

Implement paging for segments loaded from executables. All of these pages should be loaded lazily, that is, only as the kernel intercepts page faults for them. Upon eviction, pages modified since load (e.g. as indicated by the "dirty bit") should be written to swap. Unmodified pages, including read-only pages, should never be written to swap because they can always be read back from the executable.

Lazy loading is a design where the loading of memory is deferred until the point at which it is needed. A page is allocated, meaning there is page struct corresponding to it, but there is no dedicated physical frame, and the actual content of the page is not yet loaded. The contents will be loaded only at which it is truly needed, which is signaled by a page fault.

Here are the objectives:

- All of these pages should be loaded lazily, that is, only as the kernel intercepts page faults for them
- When a frame is required but none is free, choosing a frame to evict using Clock algorithm, or Second Chance

3.2 Data structure

3.2.1 Page table

The page table contains of all pages that available for that thread, maps from virtual address to page table entry. I defined a page table in hash form, each entry has the below structure:

```
struct supplemental_page_table_entry
{
```



```

void *upage;          /* Virtual address of the page (the key) */
void *kpage;          /* Kernel page (frame) associated to it.
                       Only effective when status == ON_FRAME.
                       If the page is not on the frame, should be NULL.
                       */

struct hash_elem elem;

enum page_status status;

// if ON_SWAP
swap_index_t swap_index; /* Stores the swap index if the page is swapped
                           out*/

// if FROM_DISK
struct file *file;
off_t file_offset;
uint32_t read_bytes, zero_bytes;
bool writable;
};

```

- upage: page in user space, or virtual address, which is later used as key in page table to connect virtual address with physical address.
- kpage: page in kernel space, or physical address.
- page_status: whether that page is on physical memory, on disk, on swap area or is empty, act like valid/ invalid bit

3.2.2 Frame table

The frame table contains all frame available in physical memory. There's only one frame table for every process in global scope. The frame table is managed through hash structure to map from physical address to frame table entry. The frame table entry is also managed through a circular linked list for evict algorithm.

```

struct frame_table_entry
{
    void *kpage;          /* Kernel page address,
                           also the key to hash funtion*/

    struct hash_elem helem;
    struct list_elem lelem;
};

```

```

void *upage;           /* User (Virtual Memory) Address, pointer to page
                        */
struct thread *t;      /* The thread that own this frame. */

bool pinned;           /* Used to prevent a frame from being evicted,
                        while it is acquiring some resources. */
};

```

- t: the thread that owned and loaded that frame.

3.3 Algorithms and codes

- Lazy loading: In src/userprog/process.c function load_segment(), I modified it to use vm_supt.lazy_load(), only create an entry in page table and set it up, the kpage connect to physical memory is still empty so that all pages is loaded only when page faults happen.
- Evict frame using Second Chance algorithm: pick_frame_to_evict() in src/vm/frame.c
As said above, the frame table is managed by a circular linked list and a clock pointer, use pagedir_is_accessed() to get the reference bit. The clock pointer will go in circle to find one victim, if that victim was accessed in the near past then it would be given a second chance. The pointer would go around no more than two loop and return the victim which will be evicted to give place for the coming page.

```

struct frame_table_entry* clock_pick_evict_frame( uint32_t *pagedir )
{
    size_t n = hash_size(&frame_map);
    if(n == 0) PANIC("Frame table is empty, can't happen - there is a leak
                    somewhere");

    size_t it;
    for(it = 0; it <= n + n; ++ it)
        /* to prevent infinite loop, set 2n iterations,
           worst case all frame are given second chance,
           at the second loop there is definitely a victim*/
        {
            struct frame_table_entry *e = clock_next_frame();
            // if pinned, continue
            if(e->pinned) continue;
            // if referenced, give a second chance.
            else if( pagedir_is_accessed(pagedir, e->upage)) {
                pagedir_set_accessed(pagedir, e->upage, false);
                continue;
            }
        }
}

```

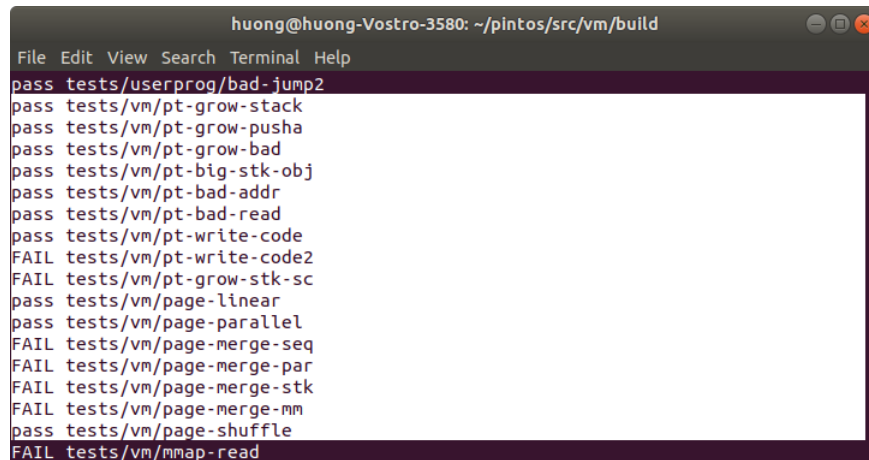
```
}  
  
return e;  
}
```

- Modify Makefile.build to use added file in vm folder.

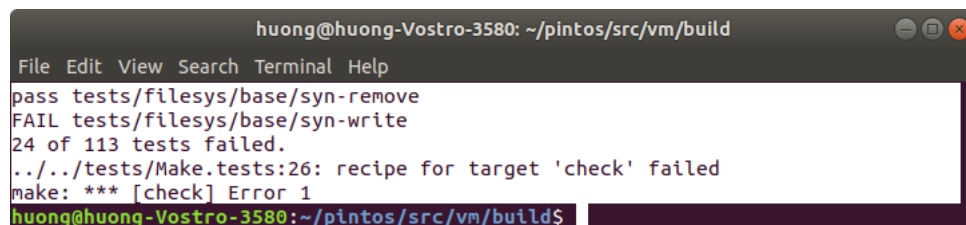
```
# Virtual memory code.  
vm_SRC = vm/frame.c  
vm_SRC += vm/page.c  
vm_SRC += vm/swap.c
```

3.4 Testing

I have implemented requirements in Paging subproject, this is my test result. I failed in running 3 in 13 tests that belong to this subproject (NOT include stack growth and memory mapping).



```
huong@huong-Vostro-3580: ~/pintos/src/vm/build  
File Edit View Search Terminal Help  
pass tests/userprog/bad-jump2  
pass tests/vm/pt-grow-stack  
pass tests/vm/pt-grow-pusha  
pass tests/vm/pt-grow-bad  
pass tests/vm/pt-big-stk-obj  
pass tests/vm/pt-bad-addr  
pass tests/vm/pt-bad-read  
pass tests/vm/pt-write-code  
FAIL tests/vm/pt-write-code2  
FAIL tests/vm/pt-grow-stk-sc  
pass tests/vm/page-linear  
pass tests/vm/page-parallel  
FAIL tests/vm/page-merge-seq  
FAIL tests/vm/page-merge-par  
FAIL tests/vm/page-merge-stk  
FAIL tests/vm/page-merge-mm  
pass tests/vm/page-shuffle  
FAIL tests/vm/mmap-read
```



```
huong@huong-Vostro-3580: ~/pintos/src/vm/build  
File Edit View Search Terminal Help  
pass tests/filesys/base/syn-remove  
FAIL tests/filesys/base/syn-write  
24 of 113 tests failed.  
../../tests/Make.tests:26: recipe for target 'check' failed  
make: *** [check] Error 1  
huong@huong-Vostro-3580:~/pintos/src/vm/build$
```

Chapter 4

CONCLUSION

This project has provided me valuable learning experiences in various aspects of operating system design and implementation, including memory management, process management, scheduling, synchronization, and user-program interaction. I encountered and resolved several technical challenges, solidifying my understanding of these concepts.

I would like to express our sincere gratitude to my instructor, Professor Pham Van Tien, for his guidance and support throughout this project. His insightful feedback and willingness to answer my questions were instrumental in this project.

Overall, this Pintos project has been a challenging but rewarding experience that has deepened my understanding of operating systems and equipped me with practical skills in their design and implementation.

———— **The End** ————