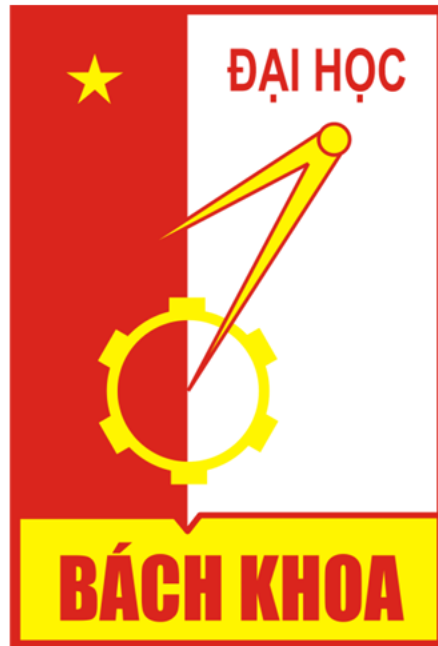


HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF ELECTRONICS AND TELECOMMUNICATIONS



PINTOS SUB-PROJECT REPORT

Argument Passing and Paging

Operating system course

Instructor: Dr. rer. nat. Pham Van Tien

Student: Nguyen Do Hoang Minh

ID: 20210591

HANOI, 12/2023

TABLE OF CONTENTS

ABOUT PINTOS.....	3
CHAPTER 1: ARGUMENT PASSING	6
1.1. Former problems	6
1.2. Solution.....	6
1.3. Results	9
CHAPTER 2: PAGING.....	12
1.1. Former problems	12
1.2. Solution.....	13
Frame Table.....	13
(Supplemental) Page Table	14
Swap table.....	16
Swapping and Evicting	16
Page loading: Page Fault Handler.....	17
Lazy-Load.....	18
Frame Pinning	18
Stack Growth.....	18
1.3. Results	19
CONCLUSION.....	21
REFERENCES	22

ABOUT 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. In the Pintos projects, you and your project team will strengthen its support in all three of these areas. You will also add a virtual memory implementation.

Pintos could, theoretically, run on a regular IBM-compatible PC. We will run Pintos projects in a system simulator, that is, a program that simulates an 80x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it.

Pintos was created at Stanford University by Ben Pfaff in 2004. It originated as a replacement for Not Another Completely Heuristic Operating System (Nachos), a similar system originally developed at UC Berkeley by Thomas E. Anderson, and was designed along similar lines.

Implemented in the C programming language, Pintos provides a streamlined yet functional operating system environment. Offering features such as threads, virtual memory, file systems, and user programs, Pintos serves as a practical platform for students to delve into and implement fundamental operating system concepts.

A notable aspect of Pintos is its modular structure, facilitating a step-by-step approach to the implementation of various operating system components. Through a series of projects, students incrementally improve Pintos' capabilities, gaining valuable insights into the complexities of real-world operating systems.

Stanford introduces CS140 Problem Set 0: Synchronization - Synchronization for students to familiarize themselves with the source code of Pintos. When students successfully deploy (i.e., PASS the TEST tasks) the content outlined in this set, the Pintos operating system will operate efficiently, with high performance, conserving the computer's memory, energy, and time resources. QEMU will be used as the simulation tool in this report.

Let's take a look at what's inside. Here's the directory structure that we should see in *pintos/src*:

threads/:

Source code for the base kernel, which we will modify starting in project 1.

userprog/:

Source code for the user program loader, which we will modify starting with project 2.

vm/:

An almost empty directory. We will implement virtual memory here in project 3.

filesys/:

Source code for a basic file system. We will use this file system starting with project 2, but we will not modify it until project 4.

devices/:

Source code for I/O device interfacing: keyboard, timer, disk, etc. We will modify the timer implementation in project 1. Otherwise, we 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. In both kernel code and user programs, headers in this directory can be included using the *#include* <...> notation. We should have little need to modify this code.

lib/kernel/:

Parts of the C library that are included only in the Pintos kernel. This also includes implementations of some data types that we are free to use in our kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the *#include* <...> notation.

lib/user/:

Parts of the C library that are included only in Pintos user programs. In user programs, headers in this directory can be included using the *#include* <...> notation.

tests/:

Tests for each project. We can modify this code if it helps us test our submission, but Stanford will replace it with the originals before they run the tests.

examples/:

Example user programs for use starting with project 2.

misc/ and *utils/*:

These files may come in handy if we decide to try working with Pintos on our own machine. Otherwise, we can ignore them.

Throughout this project, my main focus has been on the *userprog/* and *vm/* folders, with some minor contributions across other directories. Undoubtedly, this has been the most challenging project I've faced, primarily due to my lack of interest in the C language. Despite my initial reluctance, consulting various online sources became inevitable. Nevertheless, I have exerted my utmost effort to comprehend any modifications I came across, attempting to implement (and, to be honest, replicate) them to the best of my ability. I apologize if there are any imperfections in the code. Thank you for your understanding!

CHAPTER 1: ARGUMENT PASSING

1.1. Former problems

Formerly, every user program will page fault immediately until argument passing is implemented. Until we implement argument passing, we should only run programs without passing command-line arguments. Attempting to pass arguments to a program will include those arguments in the name of the program, which will probably fail.

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, calling *process_execute("ls -ahl")* will provide the 2 arguments, ["ls", "-ahl"], to the user program using *argc* and *argv*.

All of the Pintos test programs start by printing out their own name (e.g. *argv[0]*). Since argument passing has not yet been implemented, all of these programs will crash when they access *argv[0]*. Until we implement argument passing, none of the user programs will work.

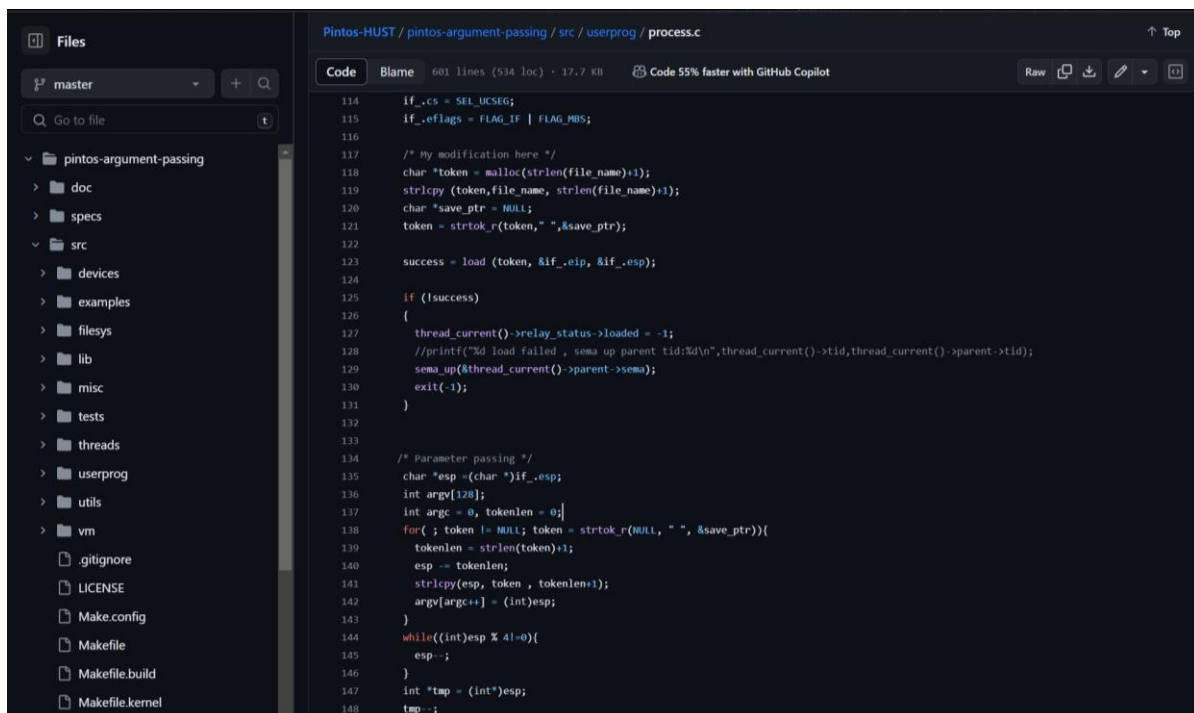
1.2. Solution

I modified the argument passing mechanism by adjusting the *load()* function in *process.c*. The main goals were to parse the command line into the *file_name* and arguments, and then load those arguments onto the stack. Within the *load()* function, I extracted the *file_name* from the command line and passed the remaining command line to the *setup_stack()* helper method.

Firstly, I passed the *file_name* to the *token* function, which splits the string on spaces, creating a list of arguments. This list represents *argv*, and the count of elements indicates *argc*. This operation takes place within the *load* function, called by *process_execute*.

The next step involves passing these arguments to the process by pushing values onto the stack during its creation in *setup_stack*. The values are pushed in the following order:

1. Push the values of elements of *argv*.
2. Push the addresses of these elements from right to left, followed by a sentinel null pointer.
3. Push *argv* (address of *argv[0]*).
4. Push *argc*.
5. Push a fake return address. Once the values are pushed onto the stack, the process reads these arguments during execution.
6. Afterward, free *argv* and continue with the execution.



```
114 if(.cs = SET_UCSEG;
115 if(.eflags = FLAG_IF | FLAG_MBS;
116
117 /* My modification here */
118 char *token = malloc(strlen(file_name)+1);
119 strcpy(token, file_name, strlen(file_name)+1);
120 char *save_ptr = NULL;
121 token = strtok_r(token, " ", &save_ptr);
122
123 success = load(token, &tf_eip, &tf_esp);
124
125 if (!success)
126 {
127     thread_current()->relay_status->loaded = -1;
128     //printf("%d load failed, sema up parent tid: %d\n", thread_current()->tid, thread_current()->parent->tid);
129     sema_up(&thread_current()->parent->sema);
130     exit(-1);
131 }
132
133
134 /* Parameter passing */
135 char *esp = (char *)if_esp;
136 int argc = 0, tokenlen = 0;
137 for( ; token != NULL; token = strtok_r(NULL, " ", &save_ptr)){
138     tokenlen = strlen(token)+1;
139     esp -= tokenlen;
140     strcpy(esp, token, tokenlen+1);
141     argv[argc++] = (int*)esp;
142 }
143
144 while((int)esp % 4 != 0){
145     esp--;
146 }
147 int *tmp = (int*)esp;
148 tmp--;
```

```

139 tokenlen = strlen(token)+1;
140 esp -= tokenlen;
141 strcpy(esp, token, tokenlen-1);
142 argv[argc++] = (int*)esp;
143 }
144 while((int)esp % 4 != 0){
145     esp--;
146 }
147 int *tmp = (int*)esp;
148 tmp--;
149 *tmp = 0;
150 tmp--;
151 int i;
152 for(i=argc-1; i>=0; i--){
153     *tmp = argv[i];
154     tmp--;
155 }
156 *tmp = (int)(tmp+1);
157 tmp--;
158 *tmp = argc;
159 tmp--;
160 *tmp = 0;
161 if(_esp = tmp; // stack update
162
163
164
165 free(file_name);
166 thread_current()->relay_status->loaded = 1;
167 //printf("%d load success, sema up parent tid:%d\n", thread_current()->tid, thread_current()->parent->tid);
168 sema_up(&thread_current()->parent->sema);
169 /* My modification ends here */
170
171 /* Start the user process by simulating a return from an
172    interrupt, implemented by intr_exit (in
173    threads/intr-stubs.S). Because intr_exit takes all of its

```

```

189 does nothing. */
190 int
191 process_wait (tid_t child_tid UNUSED)
192 {
193     /* My modification here */
194     //printf("%d process wait %d\n", thread_current()->tid, child_tid);
195     if(child_tid == TID_ERROR)
196     {
197         //printf("%d TID invalid\n", child_tid);
198         return -1; /* TID invalid */
199     }
200     struct child_process_status *child_status = get_child_status(child_tid);
201     if(child_status == NULL)
202     {
203         //printf("%d no child_status\n", child_tid);
204         return -1; /* not child_tid */
205     }
206     if(child_status->iswaited)
207     {
208         //printf("%d is being waited\n", child_tid);
209         return -1;
210     }
211     child_status->iswaited = true;
212     while(!child_status->finish)
213     {
214         //printf("%d sema down , waits for %d\n", thread_current()->tid, child_tid);
215         sema_down(&thread_current()->sema);
216     }
217     //printf("%d wait over , now free child_status->tid:%d , return %d\n", thread_current()->tid, child_tid, child_status->ret_status);
218     int res = child_status->ret_status;
219     list_remove(&child_status->elem);
220     free(child_status);
221     return res;
222     /* My modification ends here */
223 }

```

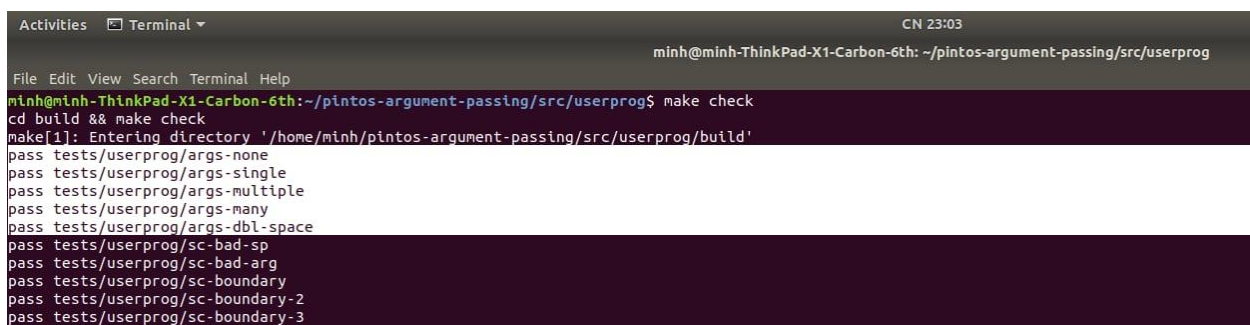
Some additions were made to the argument passing functionality in the *start_process* function within the *process.c* file. The process begins by initializing an interrupt frame

and attempting to load the specified executable file into memory. Command line arguments are tokenized using `strtok_r`, and the first token is stored in the *token* variable. The executable is then loaded, and failure is handled by setting a status flag, signaling the parent thread, and exiting with an error code if the operation is unsuccessful. The subsequent section focuses on parameter passing to the user process, processing command line arguments, preparing the stack, and updating the stack pointer if needed. Cleanup involves freeing dynamically allocated memory, setting a status flag for successful loading, and signaling the *parent thread*. If successful, the program transitions to the user process by setting the stack pointer and invoking the `exit`.

The code first checks if the provided *child thread ID* is equal to `TID_ERROR` (indicating an invalid *thread ID*) and returns -1 in case of an error. The child process status is then checked for validity, and if it's NULL, indicating no child process with the specified thread ID, -1 is returned. Additionally, if the child process has already been waited for, -1 is returned. The child process is marked as waited by setting the *iswaited* flag to true. The function waits until the child process finishes, retrieves the return status, removes the child process status from the list, frees memory, and returns the return status.

1.3. Results

First things first, Pintos passed all the arguments-related tests.

A terminal window titled 'Terminal' with a dark background. The prompt is 'minh@minh-ThinkPad-X1-Carbon-6th: ~/pintos-argument-passing/src/userprog'. The command 'make check' has been executed, resulting in a list of 13 tests, all of which passed. The tests are: tests/userprog/args-none, tests/userprog/args-single, tests/userprog/args-multiple, tests/userprog/args-many, tests/userprog/args-dbl-space, tests/userprog/sc-bad-sp, tests/userprog/sc-bad-arg, tests/userprog/sc-boundary, tests/userprog/sc-boundary-2, and tests/userprog/sc-boundary-3.

```
minh@minh-ThinkPad-X1-Carbon-6th:~/pintos-argument-passing/src/userprog$ make check
cd build && make check
make[1]: Entering directory '/home/minh/pintos-argument-passing/src/userprog/build'
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
```

Additionally, I also created a new *C* file in the *examples* folder to test the functionality of argument passing, then *make* the folder.

The *C* program, named *helloworld.c*, exemplifies command-line argument handling. It takes input from the command line using the *argc* and *argv* parameters in the main function, checking for a minimum of two arguments, including the program name. In

cases of sufficient arguments, the program prints the program name and iterates through the provided arguments using a for loop, displaying each argument along with its index.

Running the command `pintos -q run 'helloworld M i n h'` would output as the following image.

In situations with insufficient arguments, the program provides an error message indicating the issue. This concise and flexible program serves as a clear demonstration of command-line argument passing in a Pintos environment.

While the algorithms used for argument passing in this implementation may not be optimal, they effectively fulfill their purpose by enabling the successful execution of the C program. As demonstrated, the functionality of argument passing is clearly evident.

CHAPTER 2: PAGING

In operating systems, paging is a memory management scheme that allows the physical memory of a computer to be divided into fixed-size blocks called *pages*. Concurrently, the logical memory is divided into blocks of the same size, known as *frames*. This division facilitates a more flexible and efficient memory allocation and helps in implementing virtual memory.

Paging provides several benefits, including efficient use of physical memory, support for larger address spaces than the available physical memory, and simplified memory management. It enables processes to be partially loaded into memory, allowing the operating system to use disk space as an extension of physical memory when needed.

The primary goal of implementing paging in Pintos is to establish an effective mechanism for managing virtual memory. Paging provides a virtualized representation of memory to each process, presenting a seamless and private virtual address space without requiring awareness of the corresponding physical memory locations. This abstraction ensures that processes perceive a contiguous and isolated memory space. Paging also promotes the efficient utilization of physical memory, enabling multiple processes to share the same physical memory without interference. While processes can have overlapping virtual address spaces, the operating system guarantees distinct mappings to physical memory. Additionally, paging facilitates crucial functionalities such as swapping, allowing portions of a process's memory to be temporarily moved to disk when inactive, thus accommodating more processes than can fit into physical memory. Demand paging is supported, ensuring that pages are loaded into physical memory only when accessed, reducing the initial loading time of a process and enhancing system responsiveness. Furthermore, paging provides flexibility in address space management, allowing processes to have larger virtual address spaces than the available physical memory.

1.1. Former problems

Before the implementation of paging in Pintos, several limitations and challenges were present in its memory management model. One prominent issue was the lack of virtual memory support, leading to constrained address spaces for processes. Without virtual memory, processes were restricted by the available physical memory, hindering the system's ability to efficiently handle larger and more complex applications. Another

significant problem was the absence of mechanisms for efficient memory sharing between processes, limiting the potential for concurrent execution and resource utilization. Additionally, the absence of swapping and disk-based storage mechanisms posed challenges in handling processes that exceeded the physical memory capacity, impacting the system's scalability. The lack of demand paging further contributed to slower process loading times, as entire programs had to be loaded into memory at startup. These limitations collectively constrained the flexibility, scalability, and overall performance of Pintos in managing system memory before the introduction of paging.

1.2. Solution

The implementations are Frame Table, Supplemental Page Table, and Swap Table.

Frame Table

Frame refers to the realm of physical memory. One frame has a *PGSIZE* size and is all aligned.

All passable memory areas must be managed via the frame table.

Frame table entry, which is the information contained in the frame table, exists exactly one for each frame page, and the following stores information.

```
29  * Frame Table Entry
30  */
31  struct frame_table_entry
32  {
33      void *kpage;          /* Kernel page, mapped to physical address */
34
35      struct hash_elem helem; /* see ::frame_map */
36      struct list_elem lelem; /* see ::frame_list */
37
38      void *upage;          /* User (Virtual Memory) Address, pointer to page */
39      struct thread *t;      /* The associated thread. */
40
41      bool pinned;          /* Used to prevent a frame from being evicted, while it is acquiring some resources.
42                           * If it is true, it is never evicted. */
43  };
44
```

kpage: This is the kernel page address of the mapped frame, which is the key value in the hash table.

upage: The virtual memory address of the user page where the frame is loaded.

t: Refers to the owner thread that loaded the frame.

pinned: Needed for Frame pinning. These are explained in more detail below.

helem, lelem: element objects needed to put in the Frame hash table and linked list.

There is, of course, only one frame table as a global scope. The supported operations are as follows.

void vm_frame_init(): initialize. It is called once at first.

void vm_frame_allocate(enum palloc_flags, void *upage)*: corresponds to the user's virtual address *upage*. It creates a frame page, performs page mapping, and returns the kernel address of the generated page frame.

void vm_frame_free(void)*: Cancels the page frame. The entry is removed from the frame table and the resource (page) is freed.

vm_frame_pin(), vm_frame_unpin(): Needed for pinning.

To implement the above operations, internally (*kpage* \rightarrow frame table entry) is set to a struct hash *frame_map*. This allows us to quickly look up the information in the frame when given the key value of *kpage*.

(Supplemental) Page Table

Conceptually, the page table maps the user virtual address to the physical address.

When a user program accesses a virtual address, it has the same effect as accessing the memory area of the corresponding frame. In Pintos, due to the format of the page table, a separate data structure is required to store all additional information related to the (user virtual) page. It is called the Supplemental Page Table (SUPT).

One SUPT is generated per thread, *uaddr* (user page) \rightarrow SPTE (supplemental page table entry).

```

10     },
11
12     /* The hash table, page -> spte */
13     struct hash_page_map;
14 };
15
16 /**
17  * Supplemental page table. The scope is per-process.
18  */
19 struct supplemental_page_table
20 {
21     /* The hash table, page -> spte */
22     struct hash_page_map;
23 };
24
25 struct supplemental_page_table_entry
26 {
27     void *upage;          /* Virtual address of the page (the key) */
28     void *kpage;          /* Kernel page (frame) associated to it.
29                            Only effective when status == ON_FRAME.
30                            If the page is not on the frame, should be NULL. */
31     struct hash_elem elem;
32
33     enum page_status status;
34
35     bool dirty;           /* Dirty bit. */
36
37     // for ON_SWAP
38     swap_index_t swap_index; /* Stores the swap index if the page is swapped out.
39                               Only effective when status == ON_SWAP */
40
41     // for FROM_FILESYS
42     struct file *file;
43     off_t file_offset;
44     uint32_t read_bytes, zero_bytes;
45     bool writable;
46 };
47
48
49

```

upage: This is the key value, which stores the user virtual address.

status: Indicates the status of the page.

ON_FRAME: This is the case when it is loaded into a frame. In this case, the address of the loaded frame should be stored in *kpage*, and in the frame table, you can find the corresponding frame table entry with *kpage* as the key.

ON_SWAP: The current page is evicted from the frame and exists on the swap disk. In this case, swap disk in the *swap_index*, and use this value to load the appropriate place on the disk and swap in.

ALL_ZERO: This means that the Page is not loaded in the frame, but all the contents are filled with zeros.

FROM_FILESYS: If the Page is not in the frame either, but its contents should be loaded from the *filesystem* (*executable* or *memory-mapped*, etc.). You can tell which file is offset from the *file*, *file_offset*, *read_bytes*, *zero_bytes*, etc. If *writable* is false, it's *read-only page*.

Dirty: The dirty bit of the page. The dirty bit of a swapped out page cannot be found via *pagedir_is_dirty()*. It is necessary to store it separately because it is dirty or not.

Swap table

It provides operations for Swap.

*swap_index_t vm_swap_out(void *page)*: Perform Swap-Out. Write the contents of the page to the swap disk, returns a *swap_index* that identifies its location.

*void vm_swap_in(swap_index_t swap_index, void *page)*: Perform Swap-In. *swap_index* locations read a single page and write it to the page.

void vm_swap_free(swap_index_t swap_index): Simply discard the swap region.

The total sector of the swap disk divided by PGSIZE is the number of possible swap slots. Page size is the size of the swap sector. In order to store a single page in a swap, *PGSIZE/BLOCK_SECTOR_SIZE* contiguous blocks must be needed. Which swap slots are available is managed using the bitmap data structure, and it is very straightforward to implement swap in/out by mapping the corresponding blocks and pages.

Swapping and Evicting

If the page allocation fails, there is not enough memory, so swapping is required. In other words, swap out a (frame) page with a swap disk and allocate a new page to that empty space. At this time, the following things happen.

- Select a frame to evict (second-chance clock algorithm)
- Unmap the page of the frame. In other words, in the *pagedir* of the owner thread, the *upage* mapping is removed.
- Swap out the contents of the frame (*readonly, filesystem*)
- Remove the frame from the frame table (the page will also be free)
- The page is then reassigned, the newly allocated frame is put into the table and the mapping is processed. To select the frame to be evicted, a second-chance algorithm was used.

- Maintain a victim pointer (*clock_ptr*) to traverse the elements of the frame table in a circular motion.
- If the reference bit is 1, set the reference bit to 0 and move on to the next entry.
- If the reference bit is 0, select it as the target of the eviction.
-

To check/set the reference bit, simply check the *upage* mapped to the frame in the *pagedir*.

We can use *pagedir_is_accessed()* to know the reference bit, and *pagedir_set_accessed()* to set the reference bit to 0.

```

Pintos-HUST / pintos-paging / src / vm / frame.c
Code Blame 256 lines (206 loc) · 7.26 KB Code 55% faster with GitHub Copilot
Raw Copy Download Edit View

166
167 // Free resources
168 if(free_page) palloc_free_page(kpage);
169 free(f);
170 }
171
172 /** Frame Eviction Strategy : The Clock Algorithm */
173 struct frame_table_entry* clock_frame_next(void);
174 struct frame_table_entry* pick_frame_to_evict( uint32_t *pagedir )
175 {
176     size_t n = hash_size(&frame_map);
177     if(n == 0) PANIC("Frame table is empty, can't happen - there is a leak somewhere");
178
179     size_t it;
180     for(it = 0; it <= n + n; ++ it) // prevent infinite loop. 2n iterations is enough
181     {
182         struct frame_table_entry *e = clock_frame_next();
183         // if pinned, continue
184         if(e->pinned) continue;
185         // if referenced, give a second chance.
186         else if( pagedir_is_accessed(pagedir, e->upage)) {
187             pagedir_set_accessed(pagedir, e->upage, false);
188             continue;
189         }
190
191         // OK, here is the victim : unreferenced since its last chance
192         return e;
193     }
194
195     PANIC ("Can't evict any frame -- Not enough memory!\n");
196 }
197 struct frame_table_entry* clock_frame_next(void)

```

Page loading: Page Fault Handler

If you access a page that is not loaded, i.e. if you access a memory area that does not exist in *pagedir*, a page fault will occur. At this time, if there is a page in SUPT, it is not an invalid memory area access, but it is *swapped out*, *lazy-load*, etc.

As a result, the frame is not loaded, and the page needs to be reloaded. This action adds the *vm_load_page()* function.

Lazy-Load

The above paging scheme has been implemented, so instead of loading all segments into memory when Pintos loads process, it is now possible to lazy load only the necessary segments. In *load_segment()* (see process.c:639), you can use *vm_supt_install_filesys()* to load the lazy load, whereas you can fill the allocated page from the file and *install_page* it right away.

It writes the pointer to the file (the file is executable, so the pointer is valid until the process ends), offset, and size information to SUPT.

Frame Pinning

When accessing user memory, the kernel is handling the paging of the virtual memory, and if another page fault occurs, there will be a problem. For example, if you make a write system call to read user memory and write to *filesystem*, a page fault occurs. If it happens (the user page is swapped), the *filesystem* needs another access when you swap in. However, on Pintos, the *filesystem* can only be entered by holding the lock once, which causes a kernel panic caused by the double-lock.

To solve this, you need to use a method called frame pinning. Pinned frames are excluded from eviction. In the current method, a *boolean* field called *pinned* is placed in the *frame table entry* to record whether it is pinned, and it is implemented so that the pinned bit of a specific frame can be set with functions such as *vm_frame_unpin()*, *vm_frame_pin()*, etc.

Stack Growth

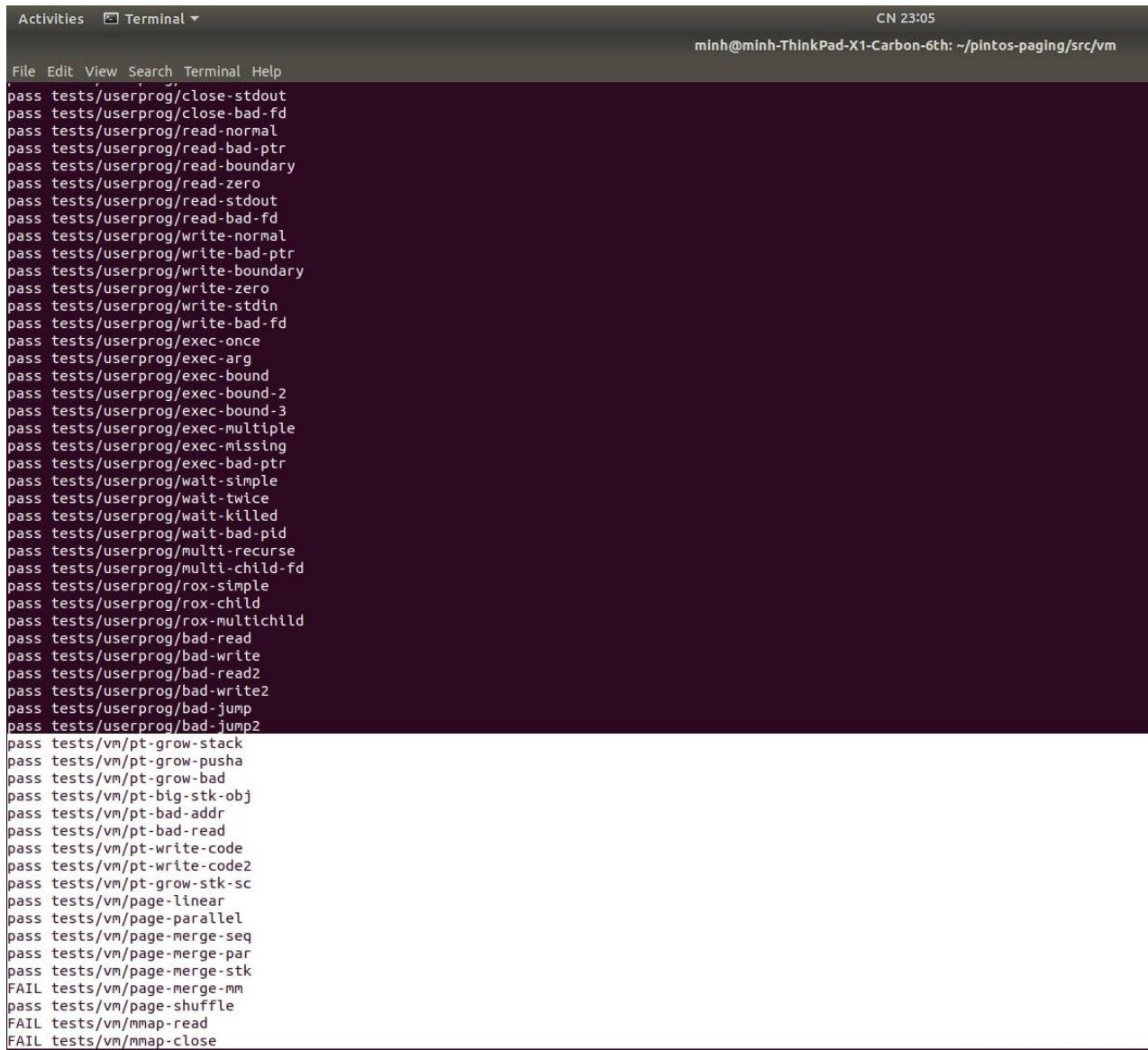
Previously, the stack of a user program consisted of only one page, and if it exceeded that page, a page fault occurred. The goal is to allow the stack to grow as much as it allows.

So, for short, currently, if the user program exceeds the stack, a page fault occurs.

First, we get the *esp*, which is a stack pointer. When *fault_addr* the address where the Page fault occurred, if both of the following conditions are satisfied, we need to allocate new pages to increase the stack.

1.3. Results

At the outset, with a relatively modest amount of implementation, the programs ran unexpectedly successfully without encountering errors.



```
Activities Terminal CN 23:05
minh@minh-ThinkPad-X1-Carbon-6th: ~/pintos-paging/src/vm

File Edit View Search Terminal Help
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
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-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
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/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
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
FAIL tests/vm/mmap-read
FAIL tests/vm/mmap-close
```

As I worked on implementing (and again, reproducing) the source code that has previously proven successful, I encountered failures in several tests. As of the current writing, I am actively engaged in debugging efforts to rectify the issues and achieve satisfactory results for the upcoming presentation day.

CONCLUSION

With the extreme complexity of the current project, I was (and still am) left speechless. Nevertheless, the crucial aspect lies in the extensive learning experience gained from the assignment provided by Dr. Tien for the class. As of now, the project still remains imperfect (which may result in a bad grade), but the value lies in the knowledge acquired by both myself and others in the class. I express gratitude to Dr. Tien, our instructor, for presenting us with this challenging yet intriguing project, which undeniably enhances our knowledge of how real-life operating systems work.

The source code for this project can be found here:

<https://github.com/minh839/Pintos-HUST>

BIG THANK YOU TO THESE REFERENCES

1. Dr. rer. nat. Pham Van Tien's slides uploaded on the class' TEAMS
2. <https://inst.eecs.berkeley.edu/~cs162/fa19/static/projects/proj1.pdf>
3. <https://inst.eecs.berkeley.edu/~cs162/sp16/static/projects/project2.pdf>
4. https://www.scs.stanford.edu/23wi-cs212/pintos/pintos.html#SEC_Contents
5. <https://pintosiiith.wordpress.com/2012/10/01/running-test-cases-for-pintos-assignment/>
6. <https://github.com/jeffzhou95/Pintos>
7. <https://github.com/hangpark/pintos/issues/4>
8. <https://github.com/codyjack/OS-pintos>
9. <https://chat.openai.com/>

and more...