TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

Case Study Report On
PintOS

In *partial fulfilment* of the requirements

For the practical course on Operating System [CT 656]

Submitted By

Abhishek Pachhain 078BEI001

Saurav Chaudhary 078BEI038

Shiva Agrahari 078BEI041

Submitted To

Department of Electronics and Computer Engineering, Pulchowk Campus

Institute of Engineering, Tribhuvan University

Lalitpur, Nepal

Under the guidance of

Assistant Professor Santosh Giri & Bikal Adhikari

Department Of Electronics and Computer Engineering

Pulchowk Engineering Campus

Shrawan, 2081

DECLARATION

I hereby declare that the report entitled "Case Study Report on PintOS" submitted to the Pulchowk Campus is a record of original work done by Abhishek Pachhain, Saurav Chaudhary and Shiva Agrahari the guidance of my esteemed mentor Assistant Prof. Santosh Giri & Bikal Adhikari. And this project work is submitted in partial fulfilment of the requirements for the practical course on Instrumentation studied during Baisakh-Shrawan 2080. The results embodied in this report have not been submitted to any other institute for the award of any type of work degree, diploma or other similar title or recognition.

Abhishek Pachhain	Saurav Chaudhary	Shiva Agrahari
078BEI001	078BEI038	078BEI041

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to our mentor Assistant Prof. Santosh Giri and Bikal Adhikari for their valuable guidance. Their continuous encouragement, engagement, support and genuine attention has always been immense source of motivation and enthusiasm for us. We like to thank both of them for his valuable time, effort and help.

The initiative of the case study to actually build, run and make changes to the OS has let us to understand a lot about the operating system and helped with the practical application of the theoretical concepts learnt during the course.

I would also like to thank my classmates and friends for their encouragement and assistance during the course of this project. Their suggestions and insights have helped me throughout the case study. Finally, I would like to acknowledge the use of various online resources, such as Stanford's comprehensive guide on PintOS, YouTube that have been instrumental in learning about PintOS.

ABSTRACT

This case study provides a comprehensive analysis of PintOS, focusing on its core components such as virtualization, synchronization, and persistence. We explored the intricacies of thread and process management, including thread structures, switching, and functions, along with scheduling and memory virtualization techniques like physical memory mapping and paging. The study also delves into system calls, synchronization mechanisms including semaphores, locks, and conditional variables, as well as the file system's role in persistence, specifically inodes, and file representation.

Furthermore, the case study includes a section on modifications and enhancements, where we implemented a wait queue to replace the existing busy wait method, and evaluated the impact of this change on system performance. The findings demonstrate an improved efficiency in thread management, showcasing the practical benefits of the proposed enhancements.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

PintOS is an educational operating system framework specifically designed for teaching fundamental operating system concepts to undergraduate students. It was developed as a part of the course material for operating systems classes at Stanford University and has since been adopted by various institutions around the world for similar educational purposes.

1.1. Purpose and Goals

PintOS is not intended to be a fully-featured or production-level operating system like Linux or Windows. Instead, it is a minimalistic OS that focuses on core operating system concepts, providing students with hands-on experience in implementing and understanding these concepts. The primary goals of PintOS are:

- **Educational Focus**: To offer a learning platform where students can explore and implement essential OS concepts, such as process management, threading, synchronization, memory management, and file systems.
- **Hands-On Implementation**: PintOS is designed for projects where students write and modify kernel code, thereby gaining practical experience in operating system development.
- **Simplified Complexity**: While PintOS implements many key OS concepts, it does so with reduced complexity, making it more approachable for students who may be new to systems programming.

1.2. Historical Background

PintOS was originally created by Ben Pfaff in 2004 as part of his work at Stanford University. It was designed to replace an older teaching OS known as Nachos. While Nachos was written in Java, PintOS is implemented in C, which provides a more realistic experience since C is commonly used in real-world operating systems.

The name "PintOS" is a play on words, suggesting that it is a smaller or simpler (pint-sized) version of a full-fledged operating system.

1.3. Architecture Overview

PintOS has a monolithic kernel architecture, meaning that all core functionalities like scheduling, file management, and device drivers are implemented within a single kernel space. The architecture is divided into several components, each focusing on a specific area of operating system functionality:

- **Threading and Scheduling**: PintOS provides the basic mechanisms for running multiple threads of execution and managing CPU time among them.
- **Process Management**: PintOS implements the ability to create, manage, and terminate processes, including the handling of user programs.
- **Memory Management**: PintOS manages both physical and virtual memory, ensuring that processes have the memory they need and that memory is efficiently utilized.
- **File System**: It implements a basic file system that supports standard file operations like reading, writing, and directory management.

- **Device Drivers**: It includes simple drivers for interacting with hardware components, particularly the disk and console.

2. Virtualization

The Operating System is supposed to create a layer of abstraction between user programs and the hardware by exposing different data structures and API (known as system calls). This kind of abstraction is known as virtualization. The virtualization in PintOS can be explained in following section

2.1. Thread

Since, PintOS is a rudimentary OS, the functionality of process where the OS takes care of available CPU core(s) by using concept of processes. Generally, OS enables the user programs to create arbitrary processes and not worry about the exact time, core they will run on, providing the assurance that they will get CPU time at some point.

PintOS only implements a simple initial thread system which include thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and optimization barriers).

2.1.1 Thread Structure

The struct thread represents a thread or a user process. Each struct thread occupies the beginning of its own page of memory, with the rest of the page used for the thread's stack, which grows downward from the end of the page. The size of struct thread must remain small, ideally well under 1 kB, to ensure there is enough room for the kernel stack.

Code*:/src/threads/thread.c*

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes).
*/
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads
list. */
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};
```

Structure Members

- `tid_t tid`: This is the thread's unique identifier, typically an int, which increments from 1 for each new thread.

- `enum thread_status` status: This member represents the current status of the thread.

```
enum thread_status
  {
    THREAD_RUNNING,     /* Running thread. */
    THREAD_READY,       /* Not running but ready to run. */
    THREAD_BLOCKED,     /* Waiting for an event to trigger. */
    THREAD_DYING        /* About to be destroyed. */
  };
```

- `char name[16]`: This is the thread's name, stored as a string of up to 15 characters plus a null terminator.
- `uint8_t* stack`: Each thread has its own stack to keep track of its state. When the thread is running, the CPU's stack pointer register tracks the top of the stack, and this member is unused. When the CPU switches to another thread, this member saves the thread's stack pointer.
- `int priority`: This represents the thread's priority, ranging from 0 (lowest) to 63 (highest). Although Pintos initially ignores thread priorities, you will implement priority scheduling in this project.
- `struct list_elem allelem`: This list element is used to link the thread into the global list of all threads. Each thread is inserted into this list when it is created and removed when it exits.
- `struct list_elem elem`: This list element is used to place the thread into doubly linked lists, such as the ready list or a list of threads waiting on a semaphore. It can serve dual purposes because a thread waiting on a semaphore is not ready to run, and vice versa.

When a thread is created in Pintos, it sets up a new context to be scheduled. User provides a function to run in this context as an argument to `thread_create`(). The thread starts executing from the beginning of this function and terminates when the function returns. Essentially, each thread acts like a mini-program, with the function passed to `thread_create()` serving as its `main()`.

2.1.2. Thread Switching

At any given time, only one thread runs while others remain inactive. The scheduler decides which thread to run next. If no thread is ready, a special "idle" thread runs. Synchronization primitives can trigger context switches when one thread needs to wait for another.

The mechanics of a context switch are handled in `threads/switch.S`, written in 80x86 assembly code. This code saves the state of the currently running thread and restores the state of the thread being switched to.

Code:*/src/threads/switch.S*

```
pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi

  # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
  mov thread_stack_ofs, %edx

  # Save current stack pointer to
old thread's stack, if any.
  movl SWITCH_CUR(%esp), %eax
  movl %esp, (%eax,%edx,1)

  # Restore stack pointer from new thread's
stack.
  movl SWITCH_NEXT(%esp), %ecx
  movl (%ecx,%edx,1), %esp

  # Restore caller's register state.
  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
ret
```

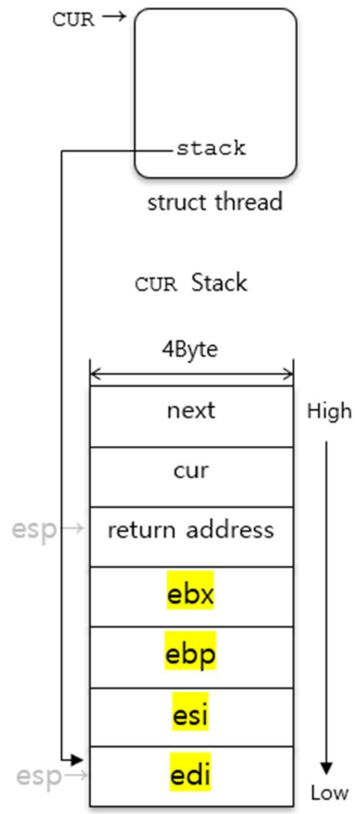

Figure 1 Thread Switching Operation in Memory

2.1.2. Thread Functions

'`src/threads/thread.c`' implements several public functions for thread support. Some of the useful ones are given below.

`void thread_init (void)`: This function initializes the thread system and creates the initial thread. It is called early in Pintos initialization to ensure that thread_current() works correctly.

`void thread_start (void)`: This function starts the scheduler by creating the idle thread and enabling interrupts, which allows the scheduler to run on return from the timer interrupt.

`tid_t thread_create (const char *name, int priority, thread_func *func, void *aux):` This function creates and starts a new thread with the given name and priority. The new thread executes the function func with aux as its argument.

`void thread_block (void)`: This function transitions the running thread to the blocked state. The thread will not run again until thread_unblock() is called.

`void thread_unblock (struct thread *t)`: This function transitions a blocked thread to the ready state, allowing it to resume running.

`void thread_exit (void) NO_RETURN`: This function causes the current thread to exit and never returns.

`void thread_yield (void):`

This function yields the CPU to the scheduler, which picks a new thread to run. The current thread might be scheduled again immediately.

Initial thread consists of only two lists in `pintos/src/threads/thread.c`

```
static struct list ready_list;
static struct list all_list;
```

- `ready_list:` It is a set of threads that are ready for execution
- `all_list:` It is a set of all threads in the system.

When a new thread is created using `thread_create()`, it is initialized with init_thread and inserted at the end of the global list all_list using `list_push_back()`. Once the thread is ready to run, it is moved to the `ready_list` by calling `thread_unblock()`.

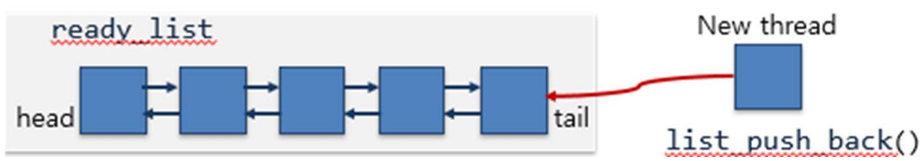

Figure 2 Thread Scheduling in PintOS

2.2. Process

Pintos can run normal C programs, as long as they fit into memory and use only the system calls implemented within the OS which is none in its original form. Notably, `malloc()` cannot be implemented because none of the system calls required for this project allow for memory allocation. Pintos also can't run programs that use floating point operations, since the kernel doesn't save and restore the processor's floating-point unit when switching threads.

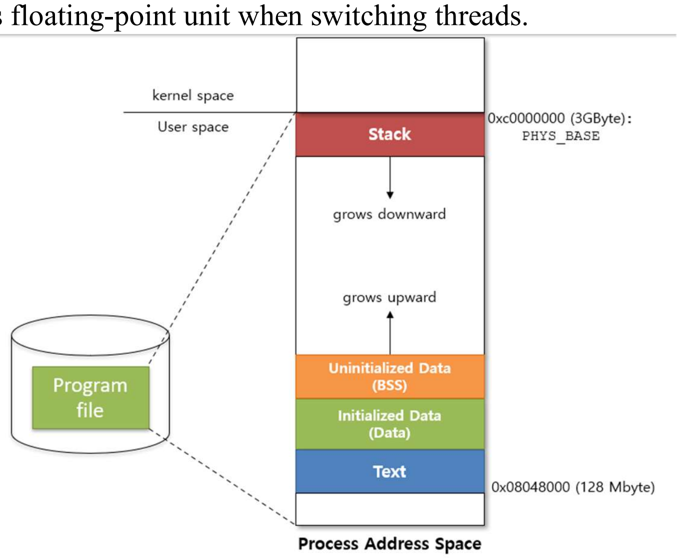

Figure 3 Process Storage in Memory

All the functions related to the process execution is defined on `/src/userprog`. Running a process in pintos constitutes of following steps.

Code: `/src/userprog/process.c`

```
static void
run_task(char **
argv)
{
...
process_wait(proc
ess_excute(argv))
;
...
}



/*
```

```
The kernel
process stops
abruptly
originally
/*
int process_wait
(tid_t child_tid
UNUSED)
{
return -1;
}
```

```
tid_t
process_execute
(const char
*file_name)
{
...
tid =
thread_create
(file_name,
PRI_DEFAULT,
start_process,
fn_copy)
...
return tid;
}
```

The process begins with the `process_execute` function, which is responsible for creating a new process by invoking `thread_create`. This function initializes a new thread structure, allocates a kernel stack, and registers the function to run, subsequently adding the thread to the ready list.

Once the thread is created, the `start_process` function takes over, loading the binary file from disk into memory, initializing the user stack, and setting the entry point of the process. If the loading process fails, the thread exits.
The load function plays a crucial role in this phase by reading the ELF header, loading data and text segments, and setting up the page table and other necessary structures.

But in current case `process_wait()`, the OS quits without allowing user process to finish.

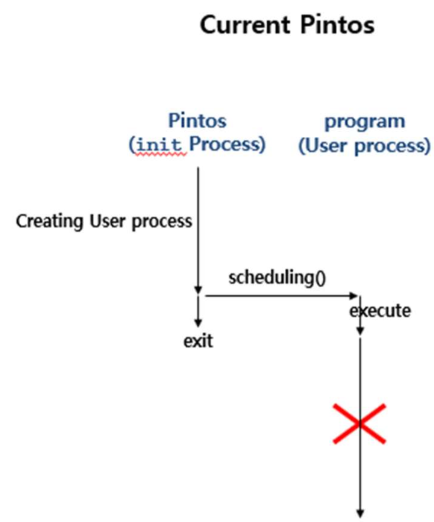

Figure 4 PintOS User Program Flow

2.3. Scheduling

The `schedule()` function initially implemented in PintOS is a **cooperative scheduler**. In a cooperative scheduling system, the currently running thread voluntarily yields control of the CPU, allowing the scheduler to select the next thread to run. This is in contrast to pre-emptive scheduling, where the scheduler forcibly interrupts the running thread to switch context.

The `schedule()` function is responsible for switching threads and is called internally by `thread_block()`, `thread_exit()`, and `thread_yield()`. Before calling `schedule()`, these functions disable interrupts and change the running thread's state.

```
thread_block (void)
{
```

```
 thread_current ()->status =
THREAD_BLOCKED;
  schedule ();
```

```
}

thread_yield (void)
{
  struct thread *cur =
thread_current ();
  enum intr_level old_level;
  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back
(&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}


thread_exit (void)
{
  intr_disable ();
  list_remove
(&thread_current()->allelem);
  thread_current ()->status =
THREAD_DYING;
  schedule ();
  NOT_REACHED ();
}
static void schedule (void)
{
  struct thread *cur =
running_thread ();
  struct thread *next =
next_thread_to_run ();
  struct thread *prev = NULL;
  if (cur != next)
    prev = switch_threads (cur,
next);
  thread_schedule_tail (prev);
}
```

Here, cur is the currently running thread. next is the next thread to run, determined by `next_thread_to_run()`. If cur is different from next, `switch_threads(cur, next)` is called to switch the context from the current thread to the next thread.

`schedule()` records the context of current thread in cur, determines the next thread to run in next by calling `next_thread_to_run()`, and then calls `switch_threads()` to perform the actual switch. The new thread returns from `switch_threads()`, which is an assembly routine that saves and restores the CPU's stack pointer and registers.

The rest of the scheduler is handled by `thread_schedule_tail()`, which marks the new thread as running and frees the resources of the dying thread if necessary.

```
void thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();
  cur->status = THREAD_RUNNING;
  thread_ticks = 0;
  if (prev != NULL && prev->status == THREAD_DYING && prev !=
initial_thread)
    {
      palloc_free_page(prev);
    }
}
```

When a new thread is created by `thread_create()`, it sets up fake stack frames for `switch_threads()`, `switch_entry()`, and `kernel_thread()`. This ensures the new thread starts correctly, enabling interrupts and calling the thread's function. If the function returns, `thread_exit()` is called to terminate the thread.

2.4. Memory Virtualization

Memory virtualization is provided by the Operating System so that each process can have their own isolated memory space without having to worry about the memory usage from other processes. The virtualization technology also enables currently unused memory to be stored in secondary storage such that the memory visible to process is much larger than the actual memory available. The basic structure of memory virtualization in Pint OS is as follows

2.4.1. Physical Memory Map

Memory Range	*Owner*	*Contents*
00000000-000003ff	CPU	Real Mode interrupt table.
00000400-000005ff	BIOS	Miscellaneous data area.
00000600-00007bff	\	\
00007c00-00007dff	Pintos	Loader.
0000e000-0000efff	Pintos	Stack for loader; kernel stack and *struct thread* for initial kernel thread.
0000f000-0000ffff	Pintos	Page directory for startup code.
00010000-00020000	Pintos	Page tables for startup code.
00010000-00020000	Pintos	Kernel code, data, and uninitialized data segments.
000a0000-000bffff	Video	VGA display memory.
000c0000-000effff	Hardware	Reserved for expansion card RAM and ROM.
000f0000-000fffff	BIOS	ROM BIOS.
00100000-03ffffff	Pintos	Dynamic memory allocation.

Each process in Pintos has its own address space, which includes different segments such as the stack, initialized data, uninitialized data (BSS), and text (code) sections. This separation ensures that each process operates in its own isolated memory space, preventing interference from other processes.

2.4.2. Paging

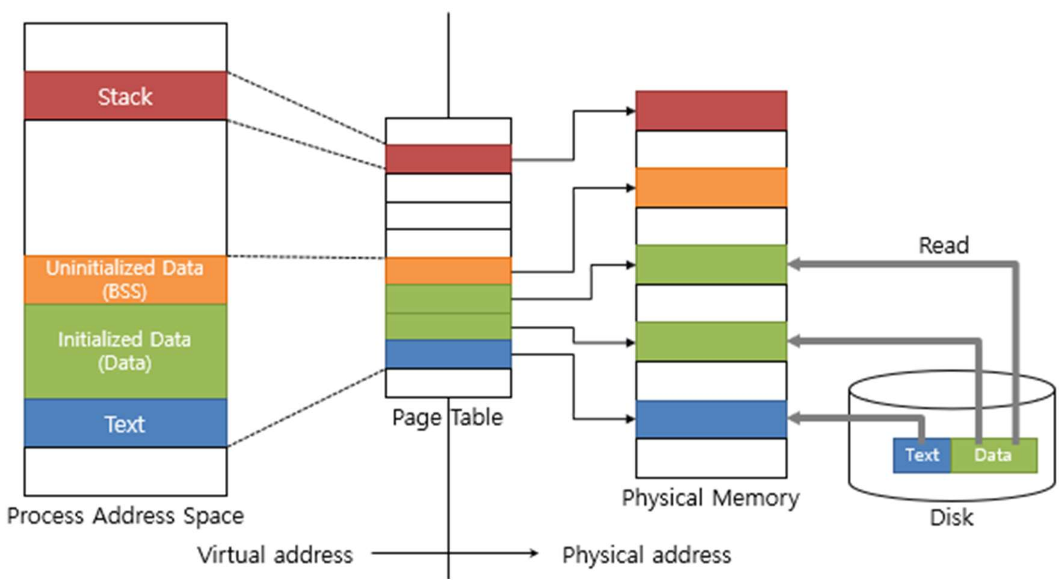

Figure 5 Paging in PintOS

PintOS offers various methods for memory virtualization but originally it only offers paging virtualization. It’s implements are as follows.

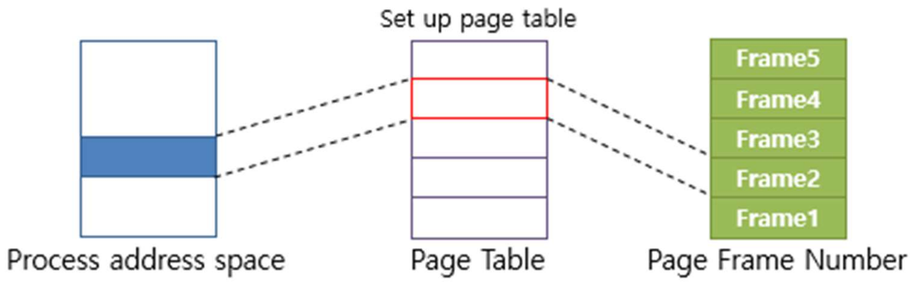

Figure 6 Pageing Table Setup in PintOS

`install_page(void *upage, void *kpage, bool writable):`

This function maps a physical page (kpage) to a virtual page (upage). The writable parameter specifies whether the page can be written to.

`void *palloc_get_page(enum palloc_flags flags):`

This function allocates a 4KB page of memory and returns its physical address. The flags parameter determines the type of memory pool to allocate from: PAL_USER for user memory, PAL_KERNEL for kernel memory, and PAL_ZERO to initialize the page to zero.

`void palloc_free_page(void *page):`

This function takes the physical address of a page as an argument and returns the page to the free memory pool.

Initially, Pintos loads the entire executable file into memory when a process starts. This includes all code and data segments. While this approach is straightforward, it is not efficient.

When a new process is created, Pintos reads the entire ELF (Executable and Linkable Format) image of the executable file. This image contains the program's code, data, and other necessary information.

The load_segment() function is responsible for loading the data and code segments from the ELF image into physical memory. These segments include the executable code and initialized data that the program needs to run.

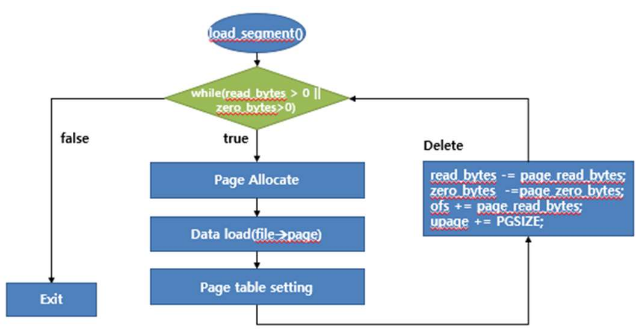

Figure 8 Program Segment Load Flowchart

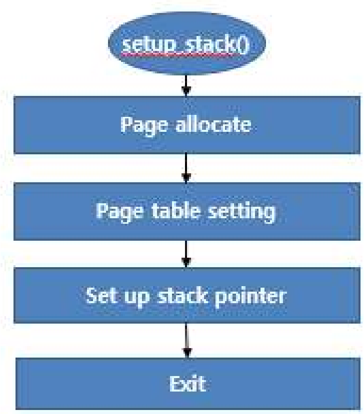

Figure 7 Stack Setup Flowchart

The `setup_stack()` function allocates a physical page for the process's stack. The stack is used for function calls, local variables, and other temporary data. It initializes the stack pointer (ESP register) to point to the top of the allocated stack page

The `page_fault()` function in Pintos, located in pintos/src/userprog/exception.c, is responsible for handling page faults. When a page fault occurs, the function checks the permissions and validates the address. If an error is detected during this process, the function generates a "segmentation fault" and calls kill(-1) to terminate the process.

2.5. System Calls

The programming interface is provided by the operating system allows user mode programs to access kernel features through system calls. These system calls run in kernel mode and then return to user mode which momentarily raises the priority of the execution mode to a special mode. PintOS's original system call handler is empty and only has skeleton of all services to be implemented in pintos/src/userprog/syscall and pintos/src/userprog/process.

```
#define syscall0(NUMBER)                                        \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[number]; int $0x30; addl $4, %%esp"       \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER)                          \
               : "memory");
          retval;
        })
```

Similar to how Pintos addresses how the operating system regains control from user programs through external interrupts from timers and I/O devices, and how the OS handles software exceptions, such as page faults or division by zero, which can also be used for system calls. In the 80x86 architecture, system calls are invoked using the int instruction, specifically int $0x30 in Pintos. The system call handler, syscall_handler(), retrieves the system call number and arguments from the stack, with return values placed in the EAX register. To avoid repetitive code, each system call argument, whether an integer or pointer, occupies 4 bytes on the stack, allowing for efficient retrieval.

3. Synchronization

In Pintos, synchronization is crucial to ensure that multiple threads can operate without interfering with each other, preventing race conditions and ensuring data consistency. PintOS deploys many synchronization patterns which are explained below.

3.1. Disabling Interrupts

This is one method used to prevent race conditions by ensuring no concurrency. This method is primarily used for coordinating data shared between kernel threads and interrupt handlers. The functions involved include `intr_disable()`, which turns off interrupts, `intr_enable()`, which turns them back on, `intr_set_level(level)`, which sets the interrupt state to the specified level, and `intr_get_level()`, which returns the current interrupt state.

3.2. Semaphores

Semaphores are another synchronization method, defined as a nonnegative integer with atomic operations to manipulate it. There are two types of semaphores: binary semaphores, which are initialized to 1 and used for mutual exclusion, and counting semaphores, which are initialized to a value greater than 1 and used for resource counting. The functions associated with semaphores include `sema_init(sema, value)`, which initializes a semaphore, `sema_down(sema)`, which waits for the semaphore to become positive and then decrements it, `sema_try_down(sema)`, which tries to decrement the semaphore without waiting, and `sema_up(sema)`, which increments the semaphore and wakes up waiting threads.

3.3. Locks

Locks are similar to binary semaphores but with ownership restrictions, ensuring that only the thread that acquires the lock can release it. This method is used to ensure mutual exclusion. The functions involved include `lock_init(lock)`, which initializes a lock, `lock_acquire(lock)`, which acquires the lock and waits if necessary, `lock_try_acquire(lock)`, which tries to acquire the lock without waiting, `lock_release(lock)`, which releases the lock, & `lock_held_by_current_thread(lock)`, which checks if the current thread holds the lock.

3.4. Conditional Variables

Condition Variables are used to block a thread until a particular condition is true, working with locks to provide a higher-level synchronization mechanism. The functions associated with condition variables include `cond_init(cond)`, which initializes a condition variable, `cond_wait(cond, lock)`, which releases the lock and waits for the condition to be `signaled`, `cond_signal(cond, lock)`, which wakes up one thread waiting on the condition, and `cond_broadcast(cond, lock)`, which wakes up all threads waiting on the condition.

4. Persistence

All the data used by the processes has been present in the main memory (i.e. the RAM). The main memory of any system is volatile i.e. it is wiped out when the power goes down. To store user data for long-term usage the OS must provide some way to store the information in persistent storage media such as SSD, HDD. Also this data needs to shared with other devices as well. This is acheived by using the network card. In this section, we look at how the Pint OS makes use of persistent storage using its file system and connects with other devices using its networking interface.

4.1 File System

PintOS uses unix based file system convention and urges the same tradition with the system calls. The files are stored to ex4 file system.

4.1.1 Inodes and File Objects

The Pintos file system uses inodes to represent files on the disk, containing information such as file size, pointers to disk blocks, and attributes like permissions and timestamps. An on-disk inode is stored on the disk, while an in-memory inode includes the on-disk inode and its location. File objects represent open files, maintaining the current read/write offset and the filesystem type, which in this case is EXT4.

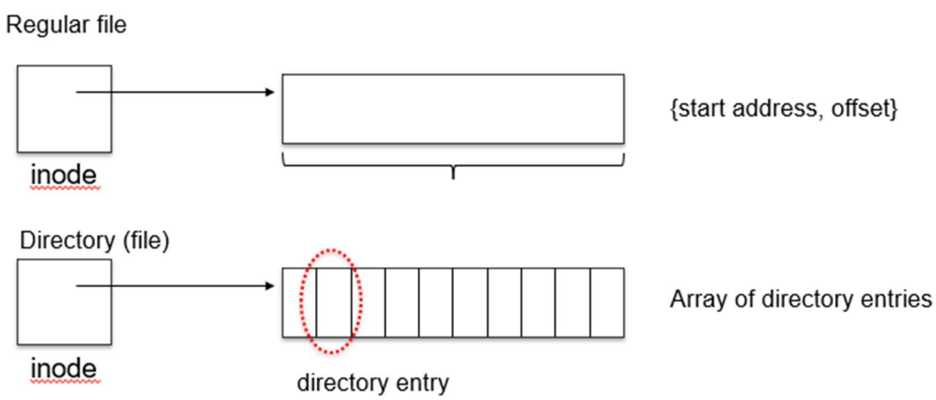

Figure 9 File Representaion as Inode in PintOS

4.1.2. File Representation

Memory inode structure

```
Code: pintos/src/filesys/inode.c
struct inode {
    struct list_elem elem;  /* Element in inode list. */
    block_sector_t sector;
    int open_cnt;           /* Number of openers. */
    bool removed;
    int deny_write_cnt;     /* 0: writes ok, >0: deny writes.*/
    struct inode_disk data; /* Inode content. */
};
```

Here, sector is a block number where inodes are stored, data is the disk_inode data while removed shows whether to delete the file.

Disk Innode

The start attribute indicates the starting address of the file data in block address format, while the length attribute specifies the size of the file in bytes. Additionally, there is an u>nused[125] area reserved for future use or padding. Each inode occupies a single sector on the disk, ensuring efficient storage and retrieval of file metadata.

```
struct inode_disk
{
    block_sector_t start;             /* First data sector. */
```

```
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
};
```

Directory entry Structure

```
File: pintos/src/filesys/directory.c
struct dir_entry
{
    block_sector_t inode_sector;
    char name[NAME_MAX + 1]; /* NAME_MAX = 14*/
    bool in_use;
};
```

The inode_sector refers to the sector number of the inode, which has a size of 512 bytes. Each file name can be up to 14 characters long. The in_use attribute indicates whether the dir_entry is currently being used.

File object

The struct file is created when a file is opened. It includes a pointer, inode, which points to the file's in-memory inode where the read/write operation is being performed. The pos attribute represents the current file offset, and the deny_write attribute indicates whether the file is writable.

```
File: /pintos/src/filesys/file.c
struct file {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;
};
```

5. Modifications & Enhancements

We undertook the projects that is assigned to student as a part of course work of Operating System. The task we undertook is modification of the process scheduling

The goal of this project is to enhance PintOS thread functionality, which currently has only a basic implementation. This involves directly modifying the thread's implementation method or their priority scheduling. The tasks to be implemented are as follows:

5.1. Wait Queue Implementation:

The task is to replace the existing busy wait method with a wait queue. When a thread sleeps, it should enter the wait queue and later move to the ready queue to be executed again after a certain time.

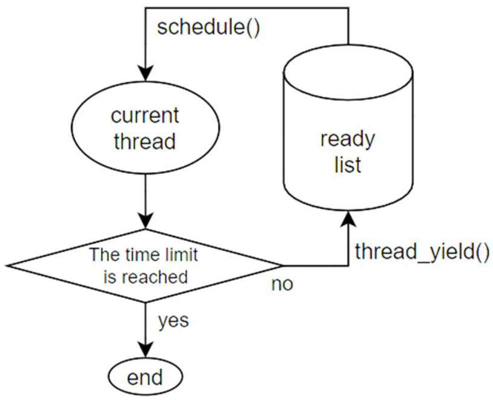

Figure 10 Current Schedule

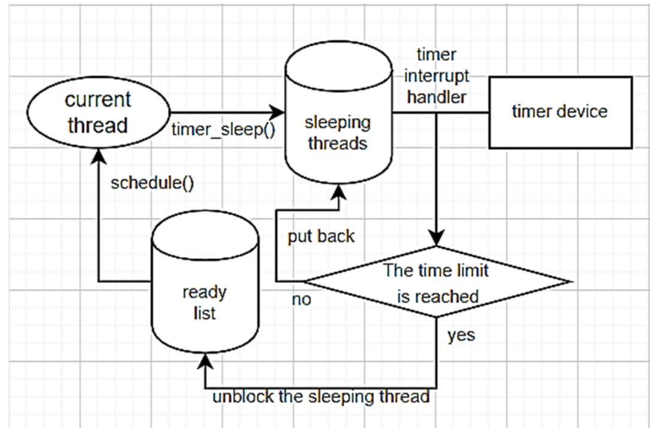

Figure 11 Improved Schedule

There is a ready queue which is a queue that manages threads that will be executed in the past.

Adds a wait queue to manage threads.

• Wait queue added (push) condition: **`Timer_sleep`** adds the thread called to the queue

• Pop condition from Wait queue

For each tick (generated from the timer), whether there is a wake-up dump (completed sleep for the specified time)

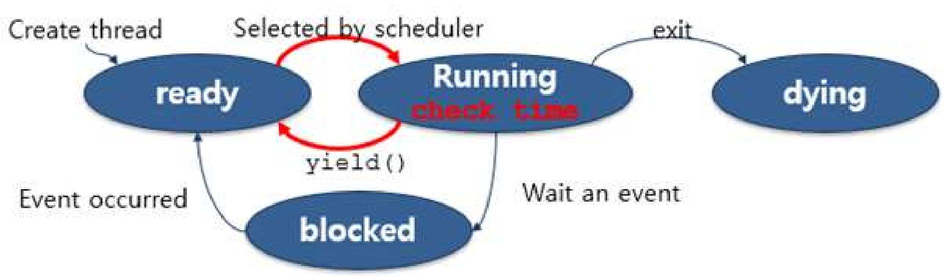

Figure 12 Improved Schedule Design

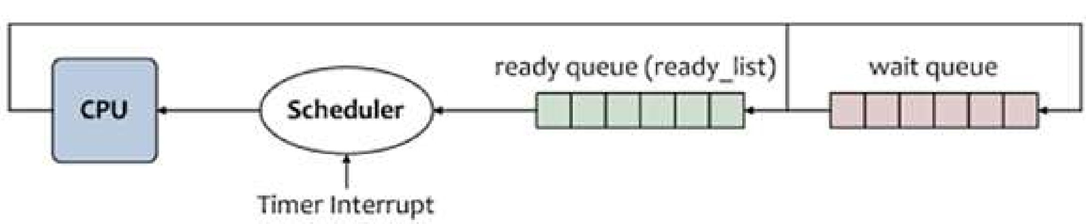

Figure 13 Datastructure in Improved Schedule

After checking, the unused items are dequeued from the wait queue and then enqueued again as the ready queue

Data Structure Used:

If each thread is to sleep, the information about which tick should be slept should be stored. This information

```
static struct list ready_list;
static struct list all_list;
static struct list sleeping_list;
```

Enqueue

The thread_sleep_until(int64_t ticks_end) function was added to handle sleeping. This function makes the thread sleep until the specified ticks_end. The sleep_endtick information is saved and the thread is enqueued. The thread_block() function is then used to block the thread's execution. The timer_sleep function now uses the thread_sleep_until() function to make the thread sleep for the specified duration (current time + sleep time ticks).

pintos/src/device/timer.c

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);
  intr_disable();
  list_push_front (&sleeping_threads, &thread_current ()->sleepElem);
  intr_enable();
  //MRM missed the add to list
  thread_current ()->endTicks = start + ticks;
  sema_down( &thread_current ()->sleep_Sem);
  //while (timer_elapsed (start) < ticks)
    //thread_yield ();
}
```

Dequeue

To check for threads that need to wake up, the timer's interrupt service routine (ISR) is used. This routine, timer_interrupt in timer.c, is called every time the timer tick changes. It uses the current tick information to identify threads that need to wake up.

The function thread_awake(int64_t current_tick) in thread.c iterates through the entire wait queue. If there are threads that need to be woken up, they are removed from the list. These awakened threads are then added back to the ready queue using the thread_unblock() function.

After implementing the wait queue, the results of running the alarm-multiple test case are as follows:

```
thread_unblock (struct thread *t)
{
  enum intr_level old_level;
  ASSERT (is_thread (t));
```

```
    old_level = intr_disable ();
    ASSERT (t->status == THREAD_BLOCKED);
    list_insert_ordered (&ready_list, &t->elem, compare_threads_by_priority,
                         NULL);
    t->status = THREAD_READY;
    intr_set_level (old_level);
  }
```

5.2 Implementation Result:

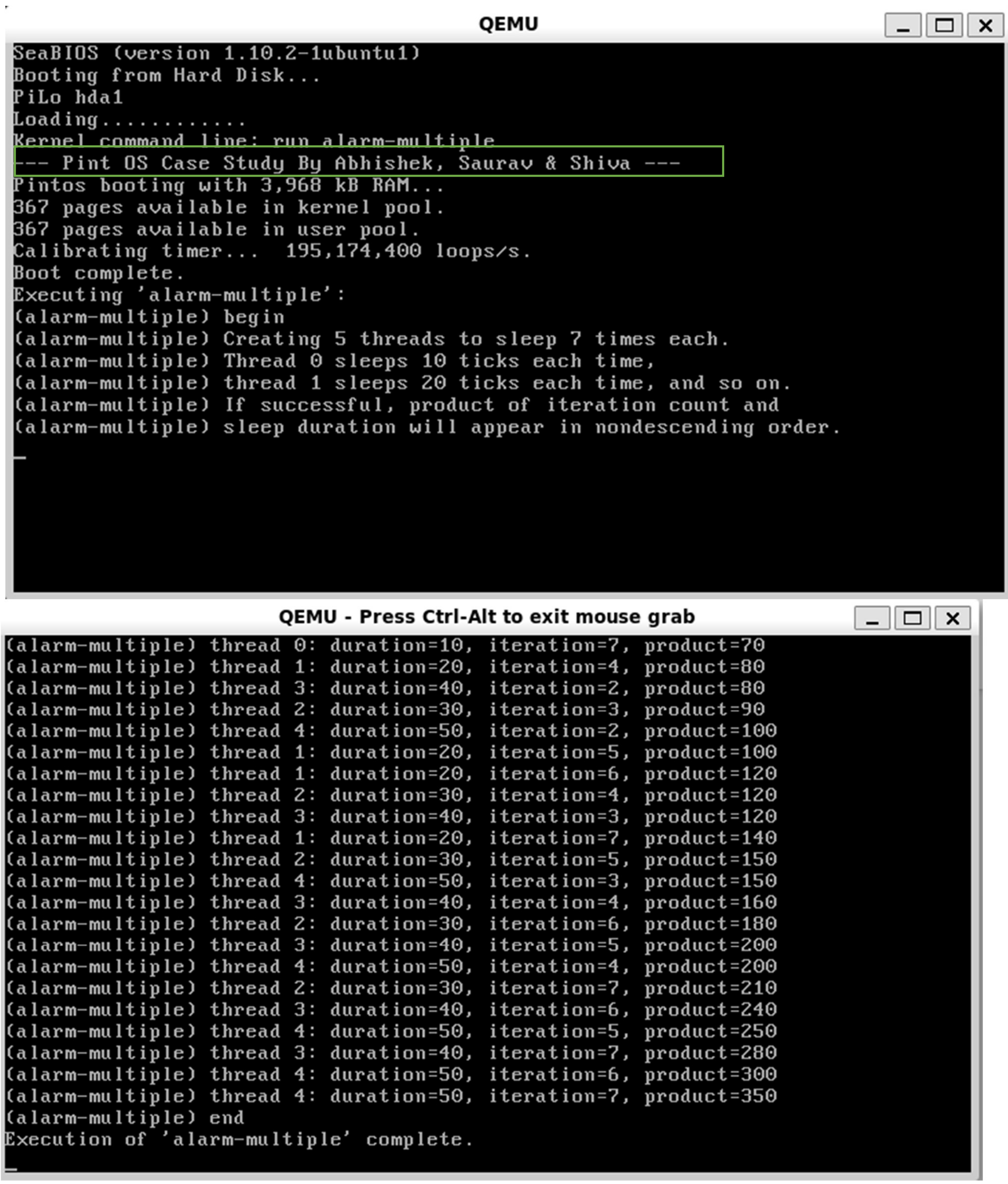

Figure 14 Implementation Result

6. Discussion

Setting up the PintOS environment on a Windows platform presented several challenges, which provided valuable learning experiences. Initially, the primary obstacle was configuring the development environment, as PintOS is traditionally designed for Unix-based systems like Linux. This required the installation of a Linux-compatible environment on Windows, such as the Windows Subsystem for Linux (WSL), which added complexity to the setup process.

During the installation of PintOS in Ubuntu 18.04 WSL, issues arose due to the limited privileges of the user in the file directories which rose a lot of problems and the OS couldn't start. We fixed by mounting the PintOS through windows which provided required execution privileges of OS. Issues regarding the file-system storage crash, path issues, compatibility issues. We observed the implementation of threads, context switching, scheduling, synchronization and implemented a portion of thread handling which gave a lot of insights on low level processing of the threads.

7. Conclusion

The hands-on experience of building, running, and modifying PintOS through this case study has deepened our understanding of operating systems. It effectively bridged the gap between theoretical concepts and their practical applications, reinforcing the knowledge gained during the course.

8. References

1. **Ben Pfaff, et al.** "PintOS: A simple operating system framework for teaching." Stanford University, Computer Science Department. Available at: https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html.
2. **Andrew S. Tanenbaum, Herbert Bos.** *Modern Operating Systems*. 4th ed., Pearson, 2014.
3. **Silberschatz, Abraham, et al.** *Operating System Concepts*. 10th ed., Wiley, 2018.
4. **Tanenbaum, Andrew S.** *Operating Systems: Design and Implementation*. 3rd ed., Pearson, 2006.
5. **The PintOS Project.** "Source Code and Documentation." Available at: https://web.stanford.edu/class/cs140/projects/pintos/pintos.tar.gz.