International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 3, March 2013)

Priority Based Implementation in Pintos

Deepa D[1], Nivas K S[2], Preethi V[3]

[1, 2, 3]*Students M-Tech, Dept. of Information Technology, SNS College of Engg., Coimbatore.*

***Abstract*—Pintos is a simple operating system framework for the 80*x*86 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.Pintos could, theoretically, run on a regular IBM-compatible PC. But it is difficult to get a regular IBM-compatible PC just for this OS, we will run Pintos projects in a system simulator, that is, a program that simulates an 80*x*86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. In class we will use the Bochs and QEMU simulators. Pintos has also been tested with VMware Player. Pintos uses a basic FIFO round robin algorithm for scheduling threads, semaphores, locks and condition variables. The current system has a minimally functional thread system.Our paper is to understand the working of pintos and extend the functionality of this system to gain a better understanding of synchronization problems byimplement priority scheduling and priority donation.**

***Keywords*— pintos, 80x86 architecture, bochs, gdb-stub, priority donation, priority inversion.**

I. INTRODUCTION

A. Need for the system

The purpose of this project is to understand the working and performance of the existing operating system named "Pintos". It will explain the existing features of the operating system and how it manages all the resources among the processes. This document is intended for the aspirants who are interested in developing an operating system.

B. Objective

Pintos uses a basic FIFO round robin algorithm for scheduling threads, semaphores, locks and condition variables. The current system has a minimally functional thread system.Our project is to understand the working of pintos and extend the functionality of this system to gain a better understanding of synchronization problems byimplement priority scheduling and priority donation. Moreover, theperformance of the existing system and implemented system is analyzed.

II. LITERATURE SURVEY/BACKGROUND

A. Existing system

Pintos is a simple operating system framework designed for the 80*x*86 architecture and it supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way.

Pintos has already implemented thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and optimization barriers).
Proper synchronization is an important part of the solutions to race condition 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. But it is not a recommended way to solve the race condition. Instead, using semaphores, locks, and condition variables will solve the bulk of your synchronization problems.

In the Pintos, the only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks. This means that data shared between kernel threads and an interrupt handler must be protected within a kernel thread by turning off interrupts.

B. Motivation for new method

Pintos is an operating system which is specially made for study purpose by Stanford University. Through Pintos we can learn the basic working and design of an operating system, and implementation of threads, semaphores and locks. There are many projects available at the basic level which motivates students to work and learn.

III. SYSTEM STUDY

A. Proposed system

Pintos is a simple operating system which has basic implementation such as kernel threads, loading and running user programs, and a file system in a very simple way.

B. Hardware specification

[IA32-v1]. IA-32 Intel Architecture Software Developer's Manual Volume 1
[IA32-v3a]. IA-32 Intel Architecture Software Developer's Manual Volume 3A

C. Software specification

[ELF1]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification
Version 1.2
Ubuntu 10.04 and above,
Bochs 2.26 and above – emulator.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 3, March 2013)

IV. Implementation Details

A. Installing Bochs

Pintos works only on 80x86 architecture. In order to provide an 80x86 architecture Bochs should be installed which acts an emulator.

B. Building Bochs

This version of Pintos is designed for use with Bochs 2.2.6. Bochs should be installed manually for use with Pintos, instead of using the packaged version of Bochs included with an operating system distribution.
Two different Bochs binaries should be installed. First one named simply bochs have the GDB stub enabled, by passing"--enable-gdb-stub"to the Bochs configure script. The second one named"bochs-dbg", should have the internal debugger enabled, by passing "--enable-debugger"to configure. (Thepintosscript selects any one based on the options passed to it.) In each case terminal and "no GUI" interfaces should be configured, by passing" --with-x --with-x11 --with-term --with-nogui"to configure.

C. Building Pintos

Build the source code supplied for the first project as the next step. First, cd into the "threads" directory. Then, issue the "make" command. It results in creating a "build" directory under "threads", populate it with a "Makefile" and a few subdirectories, and then build the kernel inside. The entire build should take less than 30 seconds.

Watch the commands executed during the build. On the Linux machines, the ordinary system tools are used.

Following the build, the following are the interesting files in the "`build`" directory:

Makefile: pintos/src/Makefile.build. It describes how to build the kernel.

kernel.o: Object file for the entire kernel. This is the result of linking object files compiled from each individual kernel source file into a single object file.

kernel.bin: Memory image of the kernel, that is, the exact bytes loaded into memory to run the Pintos kernel. This is just"`kernel.o`" with debug information stripped out, which saves a lot of space, which in turn keeps the kernel from bumping up against a 512 kB size limit imposed by the kernel loader's design.

loader.bin: Memory image for the kernel loader, a small chunk of code written in assembly language that reads the kernel from disk into memory and starts it up. It is exactly 512 bytes long, a size fixed by the PC BIOS.

Subdirectories of "build" contain object files (.o) and dependency files (.d), both produced by the compiler. The dependency files let know which source files need to be recompiled when other source or header files are changed.

D. Implementation of Priority Scheduling

In Pintos only FCFS Round Robin Scheduler was implemented. Our part of the project is to implement priority scheduler. To implement that basic functionality of pintos such as threads, semaphores, locks should be learnt.

While a thread is being added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread.While threads are waiting for asemaphore, lock or condition variable, the highest priority waiting thread should be awakened first.A thread can raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.

Thread priorities range from PRI_MIN () to PRI_MAX (). The initial thread priority is passed as an argument tothread_create().

Files involved in Priority Scheduling

Thread.h
Thread.c
Synch.h
Synch.c

V. Thread Functionalities

A. Thread_init

Called by `main()` to initialize the thread system. Its main purpose is to create a `struct thread` for Pintos' initial thread. This is possible because the Pintos loader puts the initial thread's stack at the top of a page, in the same position as any other Pintos thread.
Before thread_init() runs, thread_current() will fail because the running thread's `magic` value is incorrect. Lots of functions call thread_current() directly or indirectly, including lock_acquire() for locking a lock, so thread_init() is called early in Pintos initialization.

B. Thread_start

Called by main() to start the scheduler. Creates an idle thread, which is, the thread that is scheduled when no other thread is ready. Then enables system interrupts, which as a side effect enables the scheduler because the scheduler runs on return from the timer interrupt, using intr_yield_on_return()

C. Thread_create

const char **name*, int *priority*, thread_func **func*, void **aux*

Creates and starts a new thread which is named asname with the given priority, returning the new thread's tid. The thread executes *function*, passing *aux* as the function's single argument.

thread_create() actually allocates a page for the thread's struct thread and stack and initializes its members, then it sets up a set of fake stack frames for it.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 3, March 2013)

The thread is initialized in the blocked state, then unblocked just before returning, which allows the new thread to be scheduled.

D. Thread_block

Transitions the running thread from the running state to the blocked state. The thread will not run again until thread_unblock() is called on it, so you'd better have some way arranged for that to happen. Because thread_block() is so low-level, you should prefer to use one of the synchronization primitives instead.

E. Thread_unblock

Transitions *thread*, which must be in the blocked state, to the ready state, allowing it to resume running. This is called when the event that the thread is waiting for occurs, e.g. when the lock that the thread is waiting on becomes available.

Thread_get_priority: To get thread priority.

Thread_set_priority: To set thread priority.

Thread_yield: Yields the CPU to the scheduler and it picks a new thread to run. The new thread may be the current thread, and hence we can't depend on this function to keep this thread from running for any particular length of time.

Thread_name: Returns the name of the running thread. Equivalent to `thread_current ()->name`

Thread_exit: Causes the current thread to exit.Returns nothing, hence `NO_RETURN`

F. Priority Scheduling For Threads

Functions involved in priority Scheduling:

thread_insert_less_tail (): Takes the list elements as arguments.

Compare the priorities and returns if left_elem > right_elem.

list_insert_ordered(): Inserts ELEM in the proper position in LIST, which must be sorted according to LESS given auxiliary data AUX. Runs in O(n) average case in the number of elements in LIST.

Thread_yield(): First come First Serve(FCFS) is neglected and the threads are sorted according to the priority.

Old implementation:list_push_back (&ready_list, &cur->elem); If any new thread comes it is pushed into the `ready_list.`

*NewImplementation:*list_insert_ordered (&ready_list, &cur->elem,

thread_insert_less_tail, NULL);

Inserts cur->elem in the proper position in ready_list, which must be sorted according to thread_insert_less_tail.

G. Structure of semaphore

Semaphores are used for counting. The structure of semaphore consists of two variables.

Value (unsigned).
Waiters (struct list).

Semaphore functionalities:

*Sema_init:*struct semaphore **sema*, unsigned*value*
Initializes`sema`as a new semaphore with the given initial`value.`

*Sema_down:*struct semaphore **sema*
Executes the "down" or "P" operation over*sema*, waiting for its value to become positive and then decrementing it by one.

*Sema_try_down:*struct semaphore **sema*
Tries to execute the "down" or "P " operation over*sema*, without waiting. Returns the value astrue if*sema*was successfullydecremented, else false if it was already zero and thus could not be decremented without waiting. Calling this function wastes CPU time, so use`sema_down()` or find a different approach instead

*Sema_up:*struct semaphore **sema*.
Executes the "up" or "V" operation over*sema*, incrementing its value. If any threads are waiting on*sema*, wakes one of them up. Unlike most synchronization primitives, `sema_up ()` can be called inside an external interrupt handler.

Priority Scheduling for Semaphore:

*thread_insert_less_tail ():*Takes the list elements as arguments.

Compare the priorities and returns if left_elem > right_elem.

*list_insert_ordered ():*Inserts ELEM in the proper position in LIST, which must be sorted according to LESS given auxiliary data AUX. Runs in O(n) average case in the number of elements in LIST.

*Sema_down ():*First come First Serve (FCFS) is neglected and the threads are sorted according to the priority.

*Old Implementation:*list_push_back (&sema->waiters &thread_current ()->elem);
If any new thread comes it is pushed into the `sema->waiters.`

New Implementation:list_insert_ordered (&sema->waiters, &thread_current()->elem,
thread_insert_less_tail, NULL);

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 3, March 2013)

Insertsthread_current()->elem in the proper position in sema->waiters, which must be sorted according to thread_insert_less_tail .

*Structure of Locks:*Locks are used to avoid two or more threads accessing the same resource. Otherwise it can be said that it avoids two or more threads entering the critical section at the same time. The structure of locks consists of four variables.

Holder (struct thread *)
Semaphore (struct semaphore)

Our Implementation

Holder_elem (struct list_elem)
Lock_priority (int)

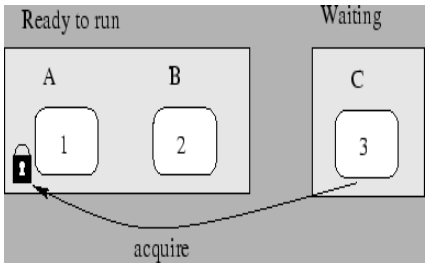
VI. ALGORITHMS

A. Priority Inversion

There is one problem in priority scheduling. The problem is known as Priority Inversion. Consider high, medium, and low priority threads *H*, *M*, and *L*, respectively. If *H* needs to wait for *L* (for instance, for a lock held by *L*), and *M* is on the ready list, then *H* will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is for *H* to "donate" its priority to *L* while *L* is holding the lock, then recall the donation once *L* releases (and thus *H* acquires) the lock.

Consider three threads A, B and C with priorities 1, 2 and 3 respectively and Thread A is having the lock. Thread C must wait for thread A to release the lock.

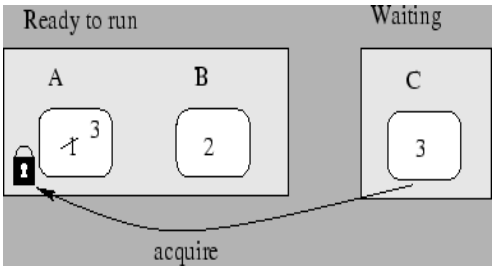
The highest priority thread, thread C, is inadvertently being blocked from running by a lower priority thread, thread B.



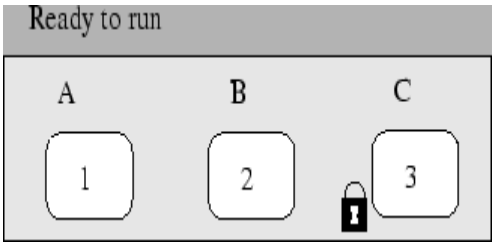
This condition is known as Priority inversion and it avoids the lower priority thread from using the resource.

B. Priority Donation

The solution for priority inversion is priority donation.
Thread C donates its priority to thread A.
Thread A finishes its work and releases its lock.



Now thread C acquires the lock and in ready list



*Priority donation in Pintos:*When new thread tries to acquire the lock, using lock_acquire, the lock_acquire function checks the priority of the thread which has the lock with the present thread.
If the present thread's priority is higher it donates priority to the lock acquired thread.

Test Case 1
Priority-Change:
Old Output:
squish-pty bochs -q

PiLo hda1
Loading.....
Kernel command line: -q run priority-change
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
Timer: 61 ticks
Thread: 0 idle ticks, 64 kernel ticks, 0 user ticks
Console: 559 characters output
Keyboard: 0 keys pressed
Powering off...

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 3, March 2013)

New Output:
PiLo hda1
Loading.....
Kernel command line: -q run priority-change
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
Timer: 75 ticks
Thread: 0 idle ticks, 77 kernel ticks, 0 user ticks
Console: 645 characters output
Keyboard: 0 keys pressed
Powering off...

VII. Experimental Results And Performance Analysis

In Pintos Performance is measured in terms of ticks. In simple terms ticks are used to represent the time taken. Performance of pintos can be found before and after implementation of priority scheduling and priority donation by just considering the number of ticks in their corresponding outputs.

Testcase1 Priority Change
Old output:
Timer: 61 ticks
Thread: 0 idle ticks, 64 kernel ticks, 0 user ticks

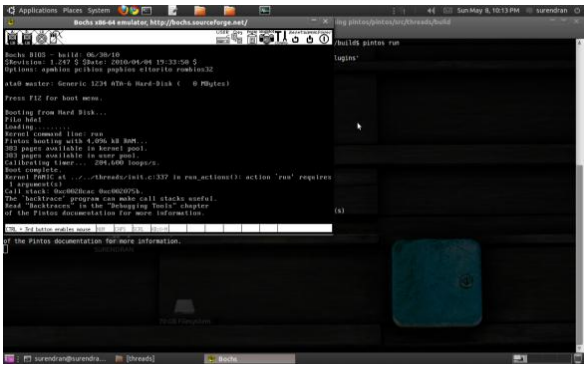
New output:
Timer: 75 ticks
Thread: 0 idle ticks, 77 kernel ticks, 0 user ticks

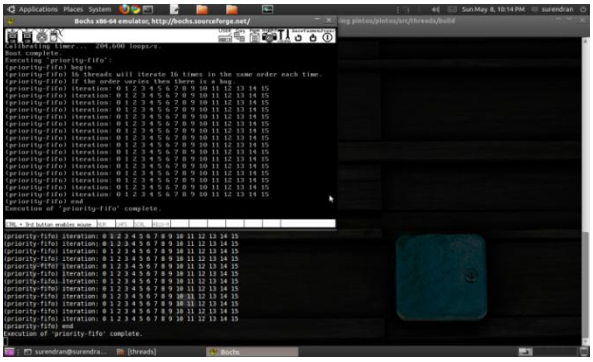
VIII. Conclusion And Future Enhancements

Pintos is an operating system which is especially made for study purpose. In this, everything is implemented in a basic way. In this operating system, priority scheduler for thread, condition variables, locks and semaphores are successfully implemented. And also priority donation is implemented so that a higher priority thread can donate its priority to a lower priority thread and later regain its priority in order to avoid the condition called priority inversion. And outputs are obtained for all the test cases.

As a next project in pintos we are working in implementing advanced scheduler. Advanced scheduler is also known as multilevel feedback queue scheduler. In this multi queues are implemented and in each queue threads will be assigned according to their priority. Higher priority threads are assigned to one queue, average priority threads to another one and so on. And each queue will get cpu according to the round robin algorithm. In this scheduler there is no need for priority donation.

Snapshots:





REFERENCES

A. Hardware References:

[1] IA32-v1. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture. Basic 80x86 architecture and programming environment. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[2] IA32-v2a. IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference A-M. 80x86 instructions whose names begin with A through M. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[3] IA32-v2b. IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference N-Z. 80x86 instructions whose names begin with N through Z. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[4] IA32-v3a. IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming.

B. Software references:

[1] ELF1. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book I: Executable and Linking Format. The ubiquitous format for executables in modern Unix systems.

[2] ELF2. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book II: Processor Specific (Intel Architecture). 80*x*86-specific parts of ELF.

[3] ELF3. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book III: Operating System Specific (UNIX Unix-specific System V Release 4). parts of ELF.

[4] SysV-ABI. System V Application Binary Interface: Edition 4.1. Specifies how applications interface with the OS under Unix.