



TRIBHUVAN UNIVERSITY  
INSTITUTE OF ENGINEERING  
PULCHOWK CAMPUS

A CASE STUDY ON  
**PINT OS**

**SUBMITTED BY:**

SAMIP GHIMIRE, PUL078BCT075  
SANDESH KUIKEL, PUL078BCT076  
SAUGAT ADHIKARI, PUL078BCT081  
SHASHANK BHATTA, PUL078BCT084

**SUBMITTED TO:**

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

# Acknowledgments

We are sincerely grateful to the Operating Systems Lab, Pulchowk Campus for providing the opportunity and platform to explore Pint OS through this case study.

We would like to express our heartfelt thanks to our instructor, Bikal Adhikari Sir, for his guidance, encouragement, and invaluable insights throughout the lab and case study.

Additionally, we are indebted to the Ben Pfaff and the Stanford University team for their tireless efforts in creating a robust, lightweight, educational operating system as an open-source software.

Finally, we extend our appreciation to my peers, colleagues, and all those who offered support and constructive feedback during the course of this work.

# Abstract

This case study explores the implementation of the priority donation feature in PintOS, an educational operating system designed for teaching and learning the principles of OS design. PintOS is known for its simplicity and modularity, making it an ideal platform for hands-on OS development.

The study focuses on the steps involved in building and emulating PintOS and the process of implementing priority donation. This feature addresses priority inversion by allowing a higher-priority thread to donate its priority to a lower-priority thread holding a needed resource. Through an analysis of PintOS's architecture, we examine the modifications required to integrate priority donation and evaluate its impact on thread scheduling and synchronization.

**Keywords:** *PintOS, Priority Donation, Operating System, Thread Synchronization, Multi-threading, Kernel Development*

**Youtube Link explaining the Case Study:** <https://youtu.be/ALXKOGh0a0Q>

# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Contents</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	1
1.2 Objectives . . . . .	1
1.3 Scope . . . . .	2
<b>2 Methodology</b>	<b>3</b>
2.1 Building and Emulating PintOS . . . . .	3
2.2 Implementing Priority Donation . . . . .	3
<b>3 Installation</b>	<b>4</b>
<b>4 Architecture</b>	<b>6</b>
<b>5 Modifications</b>	<b>9</b>
5.1 Changes in <code>threads/threads.c</code> . . . . .	9
5.1.1 Thread Unblock . . . . .	9
5.1.2 Thread Yield . . . . .	9
5.1.3 Compare Priority Function . . . . .	9
5.1.4 Sort Ready List Function . . . . .	9
5.2 Priority Donation . . . . .	10
5.2.1 Initialization in <code>init_thread</code> . . . . .	10
5.2.2 Thread Set Priority . . . . .	10
5.2.3 Search Array Function . . . . .	10
5.3 Thread Creation and Initialization . . . . .	11
5.3.1 Thread Create . . . . .	11
5.4 Changes in <code>threads/synch.c</code> . . . . .	11
5.4.1 Compare Semaphore Function . . . . .	11

5.4.2	Lock Release . . . . .	11
5.4.3	Lock Acquire . . . . .	12
<b>6</b>	<b>Testing the Modifications</b>	<b>14</b>
<b>7</b>	<b>Conclusion</b>	<b>17</b>

# 1. Introduction

Operating systems form the backbone of modern computing, providing the essential interface between hardware and software. PintOS, developed as an instructional tool, allows students to explore OS design through hands-on implementation. One key limitation of PintOS is the absence of priority donation, which can lead to priority inversion, where a high-priority thread gets indefinitely delayed by lower-priority threads.

This report focuses on building and emulating PintOS, analyzing its scheduling mechanism, and implementing priority donation to improve thread synchronization. By modifying the scheduler, thread structure, and locks, we aim to enhance PintOS's functionality and provide an in-depth discussion on its impact.

## 1.1 Problem Statement

While PintOS serves as an excellent learning tool, its lack of priority donation results in priority inversion. This problem occurs when a high-priority thread is blocked by a lower-priority thread holding a resource, while intermediate-priority threads prevent its execution indefinitely.

The priority donation feature temporarily elevates the priority of a low-priority thread holding a needed resource, allowing it to execute and release the resource before restoring its original priority. Implementing this feature improves fairness and efficiency in scheduling.

## 1.2 Objectives

This study aims to:

- Provide a step-by-step guide to building and emulating PintOS.
- Analyze PintOS's existing scheduler and thread synchronization mechanisms.
- Implement and evaluate the priority donation feature.
- Assess the impact of priority donation on OS performance.

## 1.3 Scope

This report covers:

- The process of building and running PintOS in an emulator.
- The necessary modifications to PintOS's scheduler and synchronization primitives.
- The implementation of priority donation.
- Performance evaluation and testing of the new feature.

## 2. Methodology

This study follows a structured approach to analyzing, modifying, and testing PintOS with the priority donation feature.

### 2.1 Building and Emulating PintOS

To work with PintOS, we first build and emulate the OS using QEMU. The following steps outline the process:

- Install dependencies (GCC, GDB, QEMU, Make, etc.).
- Compile PintOS using the provided build scripts.
- Run the OS in QEMU and test its default behavior.

### 2.2 Implementing Priority Donation

The implementation involves modifying PintOS's thread and scheduling mechanisms:

- Update the thread structure to support priority inheritance.
- Modify the lock acquisition and release mechanisms to enable priority donation.
- Ensure proper restoration of priorities after resource release.
- Test the changes using PintOS's test suite.



# 3. Installation

## Step 1: Install Prerequisites

Since we tried our project on two distributions (Linux Mint 24 Cinnamon edition & Ubuntu 22.04 LTS) we only know for certain these will support our installation process. In case of other distributions/ versions/ OS, we can't guarantee if the process works.

To begin the installation of the PintOS, Boot up your Mint / Ubuntu. Before downloading and compiling PintOS, install the necessary dependencies:

```
sudo apt update && sudo apt upgrade -y
sudo apt install build-essential gdb gcc-multilib make -y
sudo apt install python3 libgmp-dev libmpfr-dev libisl-dev -y
sudo apt install perl libmpc-dev qemu-system -y
```

These packages provide essential development tools, a QEMU emulator, and dependencies required to build PintOS.

## Step 2: Download PintOS Source Code

Clone the PintOS repository from GitHub:

```
git clone https://github.com/GlitchyStar717/OS_Case-Study
cd pintos
```

If using a different source, replace the repository URL accordingly.

## Step 3: Configure the Environment

PintOS requires specific paths for its tools. Set the necessary environment variables:

```
export PATH=../path_to_pintos../pintos/src/utils:$PATH
```

To make this change permanent, add it to your `~/.bashrc` or `~/.profile`:

```
echo 'export PATH=/home/hooman/Desktop/OS/new/pintos/src/utils:$PATH' >> ~/.bashrc
source ~/.bashrc
```

## Step 4: Compile PintOS

Navigate to the `src/threads` directory and compile the source:

```

(base) hooman@hooman-Legion-5-Pro-16ACH6H:~/Desktop/05/pintos/pintos-anon-master/src/threads/build$ pintos run alarm-single
qemu-system-i386 -device isa-debug-exit -hda /tmp/sKHyzq071h.dsk -m 4 -net none -serial stdio
WARNING: Image format was not specified for '/tmp/sKHyzq071h.dsk' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

PiLo hda1
Loading.....
Kernel command line: run alarm-single
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 688,947,200 loops/s.
Boot complete.
Executing 'alarm-single':
(alarm-single) begin
(alarm-single) Creating 5 threads to sleep 1 times each.
(alarm-single) Thread 0 sleeps 10 ticks each time,
(alarm-single) thread 1 sleeps 20 ticks each time, and so on.
(alarm-single) If successful, product of iteration count and
(alarm-single) sleep duration will appear in nondescending order.
(alarm-single) thread 0: duration=10, iteration=1, product=10
(alarm-single) thread 1: duration=20, iteration=1, product=20
(alarm-single) thread 2: duration=30, iteration=1, product=30
(alarm-single) thread 3: duration=40, iteration=1, product=40
(alarm-single) thread 4: duration=50, iteration=1, product=50
(alarm-single) end
Execution of 'alarm-single' complete.

QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)
Booting from Hard Disk...
PiLo hda1
Loading.....
Kernel command line: run alarm-single
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 688,947,200 loops/s.
Boot complete.
Executing 'alarm-single':
(alarm-single) begin
(alarm-single) Creating 5 threads to sleep 1 times each.
(alarm-single) Thread 0 sleeps 10 ticks each time,
(alarm-single) thread 1 sleeps 20 ticks each time, and so on.
(alarm-single) If successful, product of iteration count and
(alarm-single) sleep duration will appear in nondescending order.
(alarm-single) thread 0: duration=10, iteration=1, product=10
(alarm-single) thread 1: duration=20, iteration=1, product=20
(alarm-single) thread 2: duration=30, iteration=1, product=30
(alarm-single) thread 3: duration=40, iteration=1, product=40
(alarm-single) thread 4: duration=50, iteration=1, product=50
(alarm-single) end
Execution of 'alarm-single' complete.

```

Figure 3.1: Installation verification

```
cd src/threads
make
```

Repeat the make command in utils directory.

```
cd ../utils
make
```

## Step 5: Run PintOS with QEMU

To test run the successful compilation of the PintOS, use:

```
cd ../threads/build
pintos run alarm-single
```

# 4. Architecture

## System Architecture of Bare Minimal PiñOS

PiñOS is a simple instructional operating system designed for teaching OS concepts. It runs on a minimal hardware abstraction layer and supports basic thread management, scheduling, and synchronization. The architecture of a bare minimal PiñOS consists of the following key components:

### 1. Bootloader

PiñOS has a custom bootloader implemented in `src/loader.s` which loads the kernel in `src/start.s`. This transfers the control to the `main` in the `src/init.c`. The bootloader sets up the CPU state, loads the PiñOS kernel from disk, and transfers control to the kernel entry point.

### 2. Kernel Structure

The kernel is responsible for managing CPU execution, memory, and process scheduling. It consists of:

- **Threads and Scheduling:** The core of PiñOS operates on threads instead of full-fledged processes. It uses a priority-based scheduler to determine which thread runs next.
- **Interrupt Handling:** Basic support for handling interrupts and exceptions is provided to interact with hardware and handle faults.
- **Synchronization Primitives:** Minimal synchronization mechanisms like semaphores and locks allow safe thread coordination.

### 3. Memory Management

It provides the virtual memory and paging feature to allow addressing the memory location on an x86 based system.

### 4. File System

PiñOS implements simple file system for persistent storage but lacks support for larger files.

## 5. Device Abstraction

PintOS provides basic drivers for console I/O. It includes a simple timer to facilitate thread scheduling and sleep operations.

## 6. User Programs

PintOS introduces a user program loader and system call interface. The minimal version of PintOS does not support user-space program execution. It is required by the learner to implement the functionalities for running the user programs.

# Execution Flow in Minimal PintOS

1. **Bootloader Execution:** Initializes the CPU state and loads the PintOS kernel.
2. **Kernel Initialization:** Sets up interrupt handlers, initializes the scheduler, and creates the idle thread.
3. **Thread Scheduling:** The idle thread or any scheduled thread begins execution.
4. **Interrupt Handling:** Timer interrupts allow the scheduler to switch threads.

This minimal architecture provides a foundation for understanding OS design principles while being lightweight enough for educational use.

# Directory Structure of PintOS

PintOS has a well-structured directory hierarchy, which helps in organizing its kernel components, utilities, and test programs. Below is a brief overview of the main directories and their purpose:

1. **Root Directory (`pintos/src/`)** This is the main directory that contains the source code, documentation, build scripts, and test programs.
2. **`threads/`** Contains the core kernel implementation for thread management and scheduling. Key files:
  - `thread.c`: Implements thread creation, scheduling, and synchronization.
  - `synch.c`: Implements synchronization primitives like locks and semaphores.
  - `timer.c`: Manages system timer and sleeping threads.
3. **`userprog/` (Not in Minimal Version)** Introduces support for user programs, system calls, and process management. Key files:

- `process.c`: Handles loading and execution of user programs.
  - `syscall.c`: Implements system calls for user programs.
4. **vm/ (Virtual Memory, Optional)** Contains virtual memory management code, including page tables and swapping.
  5. **filesystem/ (File System, Optional)** Implements a simple file system for PintOS. Key files:
    - `filesystem.c`: Implements basic file system operations.
    - `directory.c`: Provides directory management support.
  6. **devices/** Manages hardware interactions, including I/O devices. Key files:
    - `console.c`: Implements basic console input/output.
    - `timer.c`: Handles system timer and scheduling.
  7. **lib/ and lib/kernel/** Contains utility functions, standard libraries, and kernel-specific helper functions.
  8. **tests/** Includes test cases to validate different functionalities of PintOS.
  9. **utils/** Contains helper scripts for building, running, and debugging PintOS.

This directory structure provides modularity, making it easier to extend PintOS by adding new features like user programs, virtual memory, and file system support.

## 5. Modifications

This chapter describes the modifications made in the code to enhance thread scheduling, implement priority donation, and refine synchronization mechanisms.

### 5.1 Changes in threads/threads.c

#### 5.1.1 Thread Unblock

Threads are now inserted into the `ready_list` in priority order using the `compare_priority` function.

```
list_insert_ordered(&ready_list , &t->elem , compare_priority , NULL);
```

#### 5.1.2 Thread Yield

When a thread yields the CPU, it is reinserted into the `ready_list` based on its priority.

```
list_insert_ordered(&ready_list , &cur->elem , compare_priority , NULL);
```

#### 5.1.3 Compare Priority Function

This function compares the priorities of two threads and returns true if the first thread has a higher priority.

```
bool compare_priority(const struct list_elem *l1 ,
const struct list_elem *l2, void *aux) {
    struct thread *t1 = list_entry(l1 , struct thread , elem);
    struct thread *t2 = list_entry(l2 , struct thread , elem);
    return t1->priority > t2->priority;
}
```

#### 5.1.4 Sort Ready List Function

This function sorts the `ready_list` based on thread priorities.

```
void sort_ready_list(void) {
    list_sort(&ready_list , compare_priority , NULL);
}
```

## 5.2 Priority Donation

Priority donation is a mechanism where a higher-priority thread can temporarily donate its priority to a lower-priority thread that holds a lock it needs. This prevents priority inversion.

### 5.2.1 Initialization in `init_thread`

Each thread now maintains an array of priorities to track donated priorities.

```
t->priorities[0] = priority;
t->donation_no = 0;
t->size = 1;
t->waiting_for = NULL;
```

### 5.2.2 Thread Set Priority

Sets the base priority of the thread and updates the scheduler if no priority donations are active.

```
void thread_set_priority(int new_priority) {
    thread_current()->priorities[0] = new_priority;
    if (thread_current()->size == 1) {
        thread_current()->priority = new_priority;
        thread_yield();
    }
}
```

### 5.2.3 Search Array Function

Removes a donated priority from the priorities array and adjusts the thread's effective priority.

```
void search_array(struct thread *cur, int elem) {
    int found = 0;
    for (int i = 0; i < (cur->size) - 1; i++) {
        if (cur->priorities[i] == elem) {
            found = 1;
        }
        if (found == 1) {
            cur->priorities[i] = cur->priorities[i + 1];
        }
    }
}
```

```

    cur->size -= 1;
}

```

## 5.3 Thread Creation and Initialization

### 5.3.1 Thread Create

After creating a new thread, the scheduler is invoked to ensure the highest-priority thread runs.

```

thread_yield();

```

## 5.4 Changes in threads/synch.c

### 5.4.1 Compare Semaphore Function

Sorts the corresponding semaphores based on the priority of the first thread in the waiting list of each semaphore.

```

bool compare_sema(const struct list_elem *l1,
const struct list_elem *l2, void *aux) {
    struct semaphore_elem *t1 = list_entry(l1,
struct semaphore_elem, elem);
    struct semaphore_elem *t2 = list_entry(l2,
struct semaphore_elem, elem);
    struct semaphore *s1 = &t1->semaphore;
    struct semaphore *s2 = &t2->semaphore;
    return list_entry(list_front(&s1->waiters),
struct thread, elem)->priority >
        list_entry(list_front(&s2->waiters),
struct thread, elem)->priority;
}

```

### 5.4.2 Lock Release

Updates thread priorities and releases the lock.

```

void lock_release(struct lock *lock) {
    ASSERT (lock != NULL);
    ASSERT (lock_held_by_current_thread (lock));

    struct semaphore *lock_sema = &lock->semaphore;

```



```

list_sort(&lock_sema->waiters, compare_priority, NULL);

if (lock->is_donated) {
    thread_current()->donation_no -= 1;
    int elem = list_entry(list_front(&lock_sema->waiters),
        struct thread, elem)->priority;
    search_array(thread_current(), elem);
    thread_current()->priority = thread_current()->priorities[thread_curr
    lock->is_donated = false;
}
if (thread_current()->donation_no == 0) {
    thread_current()->size = 1;
    thread_current()->priority =
    thread_current()->priorities[0];
}
lock->holder = NULL;
sema_up(&lock->semaphore);
}

```

### 5.4.3 Lock Acquire

Handles priority donation when acquiring a lock.

```

void lock_acquire(struct lock *lock) {
    ASSERT (lock != NULL);
    ASSERT (!intr_context());
    ASSERT (!lock_held_by_current_thread(lock));

    if (lock->holder != NULL) {
        thread_current()->waiting_for = lock;
        if (lock->holder->priority < thread_current()->priority) {
            struct thread *temp = thread_current();
            while (temp->waiting_for != NULL) {
                struct lock *cur_lock = temp->waiting_for;
                cur_lock->holder->priorities[cur_lock->holder->size] =
                temp->priority;
                cur_lock->holder->size += 1;
                cur_lock->holder->priority = temp->priority;
            }
        }
    }
}

```

```

        if (cur_lock->holder->status == THREAD_READY)
            break;
        temp = cur_lock->holder;
    }
    if (!lock->is_donated)
        lock->holder->donation_no += 1;
    lock->is_donated = true;
    sort_ready_list();
}
}
sema_down(&lock->semaphore);
lock->holder = thread_current();
lock->holder->waiting_for = NULL;
}

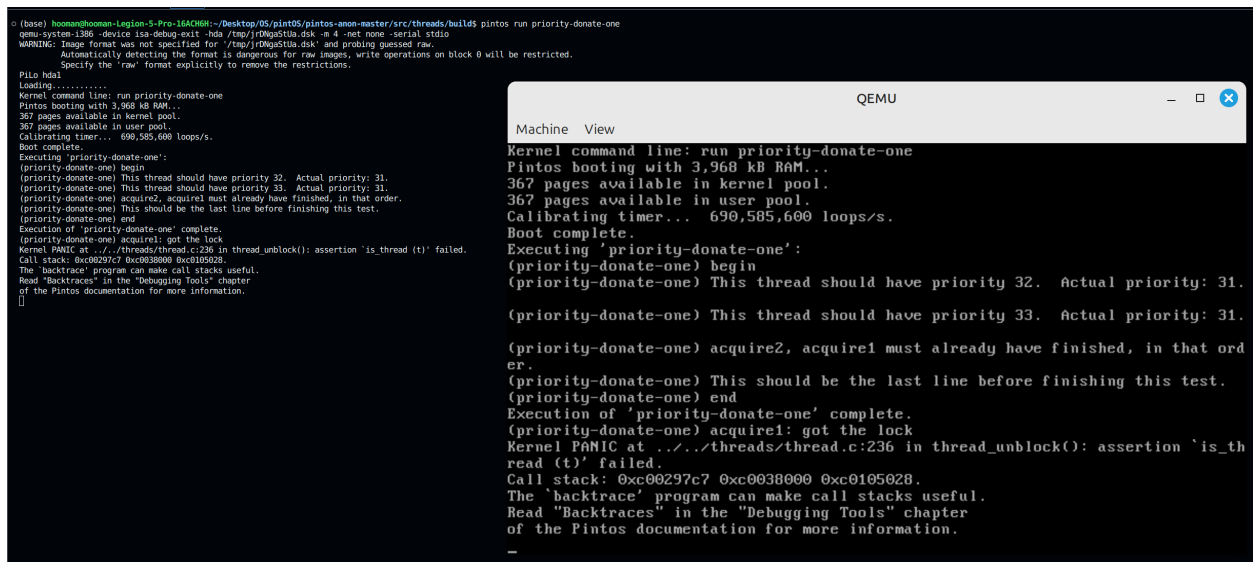
```

## 6. Testing the Modifications

To ensure the modifications to Pintos were successful, comprehensive testing was performed. This involved verifying the correct implementation of priority scheduling, priority donation, and thread creation enhancements.

### Priority Scheduling Testing

Before the modifications, the ready queue did not consistently maintain priority order. This resulted in potential scheduling inefficiencies. To illustrate this, consider the following scenario:



```
o (base) homar@homar-1:~/Pro-16AC68H~/Desktop/pintos/pintos-annot-master/src/threads/build$ pintos run priority-donate-one
qemu-system-i386 -device isa-debug-exit -hda /tmp/jdWg5t5t6.dsk -s 4 -net none -serial stdio
WARNING: Image format was not specified for '/tmp/jdWg5t5t6.dsk' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
Pilo hda1
Loading.....
Kernel command line: run priority-donate-one
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 690,585,600 loops/s.
Boot complete.
Executing 'priority-donate-one':
(priority-donate-one) begin
(priority-donate-one) This thread should have priority 32. Actual priority: 31.
(priority-donate-one) This thread should have priority 33. Actual priority: 31.
(priority-donate-one) acquire2, acquire1 must already have finished, in that order.
(priority-donate-one) This should be the last line before finishing this test.
(priority-donate-one) end
Execution of 'priority-donate-one' complete.
(priority-donate-one) acquire1; got the lock
Kernel PANIC at ../threads/thread.c:236 in thread_unblock(): assertion 'is_thread(t)' failed.
Call stack: 0xc00297c7 0xc0038000 0xc0105028.
The 'backtrace' program can make call stacks useful.
Read 'Backtraces' in the 'Debugging Tools' chapter
of the Pintos documentation for more information.
^
```

Figure 6.1: Before Modification: Not Robust (Example 1)

After implementing the priority-based scheduling changes, the ready queue consistently maintained the correct order. This was confirmed through various test cases, including scenarios with multiple threads of varying priorities.



Figure 6.4 shows a screenshot of a Visual Studio Code window. The main editor area displays the output of a QEMU terminal. The terminal output shows the execution of a program named 'priority-donate-chain'. The program starts by booting a Pintos system and then runs a series of threads and interlopers. The output shows that the threads finish in a consistent order based on their priority, with higher priority threads completing earlier. The output is as follows:

```

(base) huanan@huanan-Legion-5-Pro-16ACN8H:~/Desktop/05/new/pintos/src/threads$ ./pintos run priority-donate-chain
qemu-system-i386 -device isa-debug-exit -hda /tmp/dmE2QgTvc.dsk -m 4 -net none -serial stdio
WARNING: Image format was not specified for /tmp/dmE2QgTvc.dsk and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
Pilo Hda
Loading.....
Kernel command line: run priority-donate-chain
Pintos booting with 3,968 KiB RAM.
367 pages available in kernel pool.
Calibrating timer... 675,020,000 loops/s.
Boot complete.
Executing 'priority-donate-chain':
(priority-donate-chain) begin
(priority-donate-chain) main got lock.
(priority-donate-chain) main should have priority 3. Actual priority: 3.
(priority-donate-chain) main should have priority 6. Actual priority: 6.
(priority-donate-chain) main should have priority 9. Actual priority: 9.
(priority-donate-chain) main should have priority 12. Actual priority: 12.
(priority-donate-chain) main should have priority 15. Actual priority: 15.
(priority-donate-chain) main should have priority 18. Actual priority: 18.
(priority-donate-chain) main should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 1 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 2 got lock.
(priority-donate-chain) thread 2 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 3 got lock.
(priority-donate-chain) thread 3 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 4 got lock.
(priority-donate-chain) thread 4 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 5 got lock.
(priority-donate-chain) thread 5 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 6 got lock.
(priority-donate-chain) thread 6 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 7 got lock.
(priority-donate-chain) thread 7 should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 7 finishing with priority 21.
(priority-donate-chain) interloper 7 finished.
(priority-donate-chain) thread 6 finishing with priority 18.
(priority-donate-chain) interloper 6 finished.
(priority-donate-chain) thread 5 finishing with priority 15.
(priority-donate-chain) interloper 5 finished.
(priority-donate-chain) thread 4 finishing with priority 12.
(priority-donate-chain) interloper 4 finished.
(priority-donate-chain) thread 3 finishing with priority 9.
(priority-donate-chain) interloper 3 finished.
(priority-donate-chain) thread 2 finishing with priority 6.
(priority-donate-chain) interloper 2 finished.
(priority-donate-chain) thread 1 finishing with priority 3.
(priority-donate-chain) interloper 1 finished.
(priority-donate-chain) main finishing with priority 0.
(priority-donate-chain) end
Execution of 'priority-donate-chain' complete.

```

Figure 6.4: After Modification: Consistent Priority Order (Example 2)

## 7. Conclusion

Through this case study, we gained valuable insights into operating system design and implementation. We successfully applied core OS concepts, such as priority scheduling and priority donation, within the Pintos environment.

By refining the ready queue management and implementing priority donation, we addressed real-world OS challenges like priority inversion and inefficient thread scheduling. This exercise allowed us to translate theoretical knowledge into practical implementations, reinforcing our understanding of fundamental OS principles.

The modifications detailed in this document underscore the importance of robust scheduling and synchronization mechanisms in modern operating systems. This case study provided a hands-on experience, enabling us to learn and apply crucial OS concepts that are directly relevant to real-world systems.