Introduction to Operating Systems

Homework 3: Priority Based Scheduling

Introduction

Xv6 is back in your lives again…
In this homework, you will learn how Scheduling and context switching work in xv6. Read chapter 5 in the xv6 book. Round-robin is the default and only scheduling policy in xv6. Regardless of the priority, every runnable process gets an equal time slice in the CPU.

For this assignment, you will have to create a Round Robin Scheduler with 5 levels of priorities in xv6. The higher the value, the lower the priority.

Task 1 - Nice (20)

Having policies is the first step in implementing a non-trivial scheduler. For the sake of simplicity, you can implement the nice UNIX API as an extra system call (in xv6). A higher nice value means lower priority. For more information type – man 2 nice on the Linux system. You should be able to store the nice value in the process structure. Xv6 stores the PCB in `struct proc` in the file `proc.h`

Note: You are not expected to enforce specific permissions on nice. On Linux systems, typically only root can increase priorities. But you can skip this part in the assignment.

Your nice system call should accept `pid` of a process and `value` as parameters and change the nice value of the process to the new value provided. The system call should return the `pid` and old `value` for the process. You also need to implement a CLI program for nice that should be executable as:

`nice <pid> <value>` or
`nice <value>`

Examples:

```
Unset
$ nice 1 4  // PID = 1 and new Value = 4
1 3     // returning the PID and old nice value

$ nice 2              // if only 1 argument is provided, assume it is the value
3 3                   // and change the nice value for the current process.
                      // you have to figure out how to get PID of the current
                      // process.
```

Deliverables

- Implement the nice system call in xv6.
- nice.c with some runs showing nice system call is able to change the nice values.
- Your test runs must include the ability to handle edge cases. What happens if the values passed in are out of bounds?
- You may choose to create different programs called test1.c, test2.c, and so on that run predefined test cases showing how nice values change.

Task 2 - Scheduler (80)

Priority Scheduler works by giving time quantum to higher priority processes first and moving to lower priority processes only if no higher priority process remains in the `RUNNABLE` state.

The design problem is open-ended, you can choose to implement this however you want. Each process entering the queue will have a medium priority assigned at birth.

Note: You should keep both the schedulers as an option. It is fine to select the scheduler (Round Robin vs. Round Robin with Priority) to be a compile-time option.

How to test:

Take a program that runs a complicated process like printing prime numbers!
Pseudo code:

```
C/C++
for i = 1 to infinity:
  prime = true
  for j = 2 to sqrt(i)
    if i % j == 0
      prime = false
      break
  if prime:
    print(i)
```

Fork a bunch of children executing this process and assign different nice values to them. Check which process prints more and whether these results are consistent with their respective nice values.

Deliverables

- Come up with at least 3 test cases for your Priority scheduler
- Explain your implementation in the PDF.
- Provide screenshot(s) of your test cases along with an explanation.
- These test cases should exist as test1.c, test2.c, and test3.c
- When executed, the above files should generate results consistent with your observations mentioned in the supporting PDF.

Extra Credit (50)

Background

In real-time operating systems, priority inversion is a scenario where a high-priority task is indirectly preempted by a lower-priority task effectively "inverting" the expected priority scheduling mechanism. This scenario famously occurred in the Mars Pathfinder mission in 1997, leading to system resets.

Priority Inheritance

When a high-priority task is waiting for a resource that is currently held by a low-priority task, we face a priority inversion problem. The solution is priority inheritance: the low-priority task temporarily inherits the priority level of the high-priority task waiting on it. This elevation in priority ensures that:

1. The low-priority task can complete its critical section faster
2. The inherited higher priority allows it to preempt any medium-priority tasks
3. Once it releases the lock, its priority is restored to its original level
4. The high-priority task can then acquire the lock and proceed

This mechanism prevents the classical priority inversion problem where medium-priority tasks could preempt the low-priority task, further delaying the high-priority task.

Overview

You will implement a priority inversion scenario and its resolution within the XV6 operating system. The implementation will demonstrate the classic priority inversion problem and implement a basic priority inheritance protocol to resolve it.

Requirements

- Implement two processes with different priorities:
    - A low-priority process (like the one that periodically collects data on the Mars rover)
    - A high-priority process (like the one that handles communication on the Mars rover)
- Both processes must share a common resource protected by a lock
- To make things easier, resource list can be an array of integers, let's say 1-7

Create the following system calls:

```C/C++
int lock(int id);
//Attempts to lock resource identified by integer id

int release(int id);
// Release the lock if held by the current process
```

Your implementation should:

- Create a situation where the low-priority process acquires a lock
- Have the high-priority process attempt to acquire the same lock
- Demonstrate the priority inversion problem where the high-priority process is waiting to acquire the lock
- Implement priority inheritance to resolve the situation

Required Behavior

Your system should:

- Allow for the creation of multiple locks (maximum 7)
- Handle priority inheritance when priority inversion is detected
- Restore original priorities after the lock is released
- Properly handle multiple processes attempting to acquire the same lock

Implementation Guidelines

1. Start by implementing the basic lock mechanism
2. Add priority levels to processes
3. Implement priority detection and inheritance
4. Add proper restoration of priorities

Testing Requirements

Your submission must include test programs demonstrating the following scenarios:

1. Basic Lock Functionality

- Test Case 1: Basic Lock/Unlock
    - Create a process that acquires and releases a lock
    - Verify the lock state before and after operations
    - Ensure proper handling of invalid lock IDs (0-7)
    - Create another process at the same priority and show how it waits to acquire a lock on a resource that hasn't been released yet.

2. Priority Inversion Demonstration

- Test Case 2: Basic Priority Inversion
    - Create a low-priority process that:
        - Acquires a lock
        - Performs work for at least 5 seconds while holding the lock
    - Create a high-priority process that:
        - Starts after a delay of approximately 3 seconds (or before the low priority process releases the lock)
        - Attempts to acquire the same lock
    - Your test should log:
        - When the low priority task was started
        - When the lock was acquired (can be in relative units of time)
        - When the high priority task was started
        - Process priorities at key points
        - Lock holder information
- Test Case 3: Priority Inheritance

    - Demonstrate the priority inheritance mechanism:
        - Show the original priority of the low-priority process
        - Show the priority elevation when inversion is detected
        - Show the priority restoration after lock release
    - Log all priority changes with timestamps (relative time is also fine)

Each test case should:

- Print clear start/end markers
- Include timestamp information
- Report success/failure status
- Document expected vs actual behavior
- Log all relevant state changes (priorities, lock status)

Deliverables

1. Modified XV6 source code
2. Documentation explaining your implementation and results. To be included in the final PDF
3. Test suite implementing all specified test cases (This may also be just one program file but the output should be clear to understand or we might not grade it optimally)

Submission:

1. Entire xv6-public Folder (make sure it's the correct one)
2. A README specifying how to run each part of the assignment.
3. A PDF containing an explanation of all parts as well as screenshots of tests.
4. partner.txt