Breakdown of PintOS project

Introduction:

The goal of this project is to add a new implementation to Pinto's thread function out of which only the basic parts have been implemented.

There are two main tasks to directly modify and implement the implementation method of threads and their priority scheduling:

1). Wait Que (Alarm Clock): When a thread sleeps, remove the existing busy wait method and have it enter the wait queue and implement it to enter the ready queue so that it can be executed again after a certain time . When executing alarm-multiple, the number of idle ticks can be checked to determine whether there is a busy wait .

2). Priority and Multi Level Feedback Que Scheduling : The basic purpose is to ensure that threads with high priority are executed first. Depending on various scenarios, such as when a new thread is created or the priority is changed, it must always be executed according to priority. A priority inversion problem may occur when there is a lock (critical section), and this problem must be resolved using priority donation .

Task 1

Implementation:

Enqueue

The thread_sleep_until function was added for sleep which causes the current thread to sleep until ticks_end ticks.

As explained above, after storing sleep_endtick information in the thread and enqueuing it the thread is blocked through thread_block() .

In the timer_sleep function we can sleep the thread (current time + number of ticks to sleep) using the thread_sleep_until() function.

Dequeue

To check which thread will wake up, check the timer at every tick.

To do this we use the timer's interrupt service routine.

There is an Interrupt Service Routine, timer_interrupt that is called every time the timer tick changes and here we can know which threads will be interrupted through the current tick information.

It is implemented in the thread_awake function, which traverses the entire wait queue and removes from the list if there is a thread to wake up.

The awakened thread can be added back to the ready queue through thread_unblock().

Test Results: alarm-multiple

The execution results of the alarm-multiple test case after the wait queue has been implemented is as follows:

```
Loading.........
Kernel command line: -q run alarm-multiple
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 314,163,200 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin (alarm-multiple) Creating 5 threads to sleep 7 times each. (alarm-multiple)
Thread 0 sleeps 10 ticks each time, (alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and (alarm-multiple) sleep duration will
appear in nondescending order. (alarm-multiple) thread 0: duration=10, iteration=1, product=10 (alarm-
multiple) thread 1: duration=20, iteration=1, product=20 (alarm-multiple) thread 0: duration=10,
iteration=2, product=20 (alarm-multiple) thread 2: duration=30, iteration=1, product=30 (alarm-
multiple) thread 0: duration=10, iteration=3, product=30 (alarm-multiple) thread 3: duration=40,
iteration=1, product=40 (alarm-multiple) thread 1: duration=20, iteration=2, product=40 (alarm-
multiple) thread 0: duration=10, iteration=4, product=40 (alarm-multiple) thread 4: duration=50,
iteration=1, product=50 (alarm-multiple) thread 0: duration=10, iteration=5, product=50 (alarm-
multiple) thread 2: duration= 30, iteration=2, product=60 (alarm-multiple) thread 1: duration=20,
iteration=3, product=60 (alarm-multiple) thread 0: duration=10, iteration=6, product=60 (alarm-
multiple) thread 0: duration=10, iteration=7, product=70 (alarm-multiple) thread 3: duration=40,
iteration=2, product=80 (alarm-multiple) thread 1: duration=20, iteration=4 , product=80 (alarm-
multiple) thread 2: duration=30, iteration=3, product=90 (alarm-multiple) thread 4: duration=50,
iteration=2, product=100 (alarm-multiple) thread 1: duration=20, iteration=5, product=100 (alarm-
multiple) thread 3: duration=40, iteration=3, product=120 (alarm-multiple) thread 2: duration=30,
iteration=4, product=120 ( alarm-multiple) thread 1: duration=20, iteration=6, product=120 (alarm-
multiple) thread 1: duration=20, iteration=7, product=140 (alarm- multiple) thread 4: duration=50,
iteration =3, product=150 (alarm-multiple) thread 2: duration=30, iteration=5, product=150 (alarm-
multiple) thread 3: duration=40, iteration=4, product=160 (alarm-multiple) thread 2: duration=30,
iteration=6, product=180 (alarm-multiple) thread 4: duration=50, iteration=4, product=200 (alarm-
multiple) thread 3: duration=40, iteration=5, product= 200 (alarm-multiple) thread 2: duration=30,
iteration=7, product=210
Timer: 582 ticks
Keyboard: 0 keys pressed
Thread: 550 idle ticks, 32 kernel ticks, 0 user ticks
(alarm-multiple) thread 3: duration=40, iteration=6, product=240 (alarm-multiple) thread
4: duration=50, iteration=5, product=250 (alarm-multiple) thread 3: duration=40,
iteration=7, product=280 (alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350 (alarm-multiple) end
Execution of 'alarm-multiple' complete.
Console: 2955 characters output
Powering off...
```

Task 2

Implementation:

Once it is added to the ready_list , it maintains the sorted order.

Modifying the part added to the ready_list that exists in thread_unblock() and thread_yield().

It is then sorted according to the priority of the thread and inserted into the list using the list_insert_ordered() function, for a high priority thread to be scheduled.

When a new thread is created and added, thread_create and has a higher priority than the current thread.

When thread is unblocked, thread_unblock and has a higher priority than the current thread t.

When thread wakes up from the semaphore queue, sema_up.

When the priority is higher than the current thread.

When the priority of the current thread changes, the thread to be executed immediately next, if it exists and has a higher priority immediately process thread_yield so that rescheduling can occur.

Priority Inversion Problem

If there is a lock then priority inversion problems may occur.

There are high/medium/low priority threads: H, M, and L, respectively.

If L is running by a certain lock, H is in a block state and M is in the ready list then M is always higher than L.

This is a phenomenon in which H is not executed because it is scheduled first .

To prevent this we need to make a priority donation, but if we think about the simplest scenario, then:

When the current thread H acquires the lock, there is already a holder thread L holding the lock and the priority of H is If it is higher than L:

- Donate L's priority to H 's priority.
- At this time, the original priority of L must be remembered.

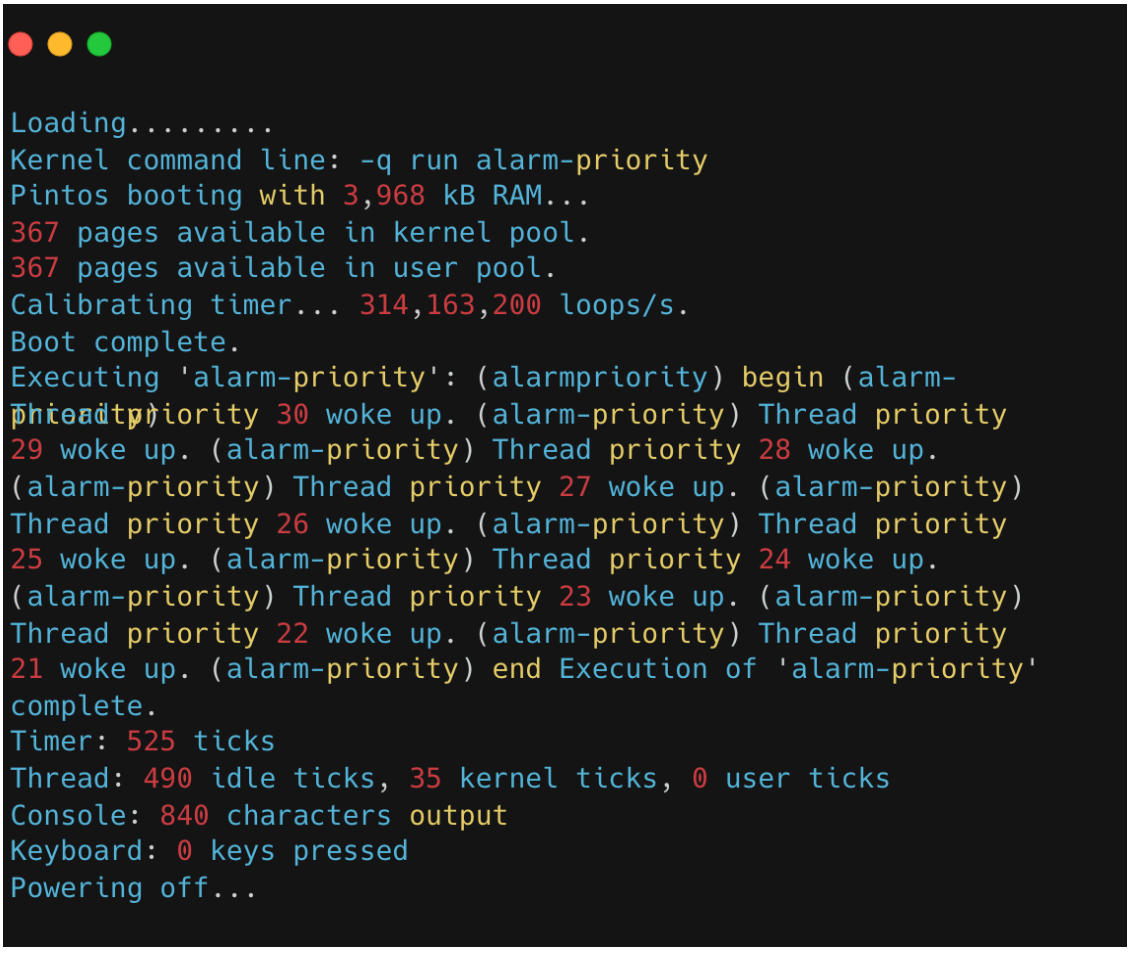
When L tries to release the lock:

- Restore L's priority from original_priority

It operates by the same mechanism.

When one thread receives priority from more than one thread since donations may occur due to overlap, the data structure must be designed keeping this in mind.

Test Results: alarm-priority



```
Loading.........
Kernel command line: -q run alarm-priority
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 314,163,200 loops/s.
Boot complete.
Executing 'alarm-priority': (alarmpriority) begin (alarm-
priority)
Thread priority 30 woke up. (alarm-priority) Thread priority
29 woke up. (alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up. (alarm-priority)
Thread priority 26 woke up. (alarm-priority) Thread priority
25 woke up. (alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up. (alarm-priority)
Thread priority 22 woke up. (alarm-priority) Thread priority
21 woke up. (alarm-priority) end Execution of 'alarm-priority'
complete.
Timer: 525 ticks
Thread: 490 idle ticks, 35 kernel ticks, 0 user ticks
Console: 840 characters output
Keyboard: 0 keys pressed
Powering off...
```