Operating System (Fall 2014) Project 1

Wait Queue, Priority Scheduling

Jongwook Choi

Introduction

The goal of this project is to implement only the basic parts Pintos of thread Add a new approach to the implementation and functions through it thread The implementation or their priority scheduling Enlarge directly modifying, viewing and implementation challenges, there are two.

One. **wait queue Implementation:** thread end sleep When, the old busy wait Remove the scheme and wait queue To enter the
And after a certain time to be re-run ready queue Enter the implement so. alarm-multiple
If you try to run Idle tick To be checked busy wait It can be seen or not.

2. **priority scheduling Implementation:** High priority thread The primary purpose is to allow to run first
to be. A thread or newly created, priority It must be run as gekkeum always prioritized based on different scenarios of change or the like.
lock (critical section) When the priority inversion There is a problem can arise,
priority donation The use and should solve this problem.

Task 1. Wait Queue

Design

Queue that manages the threads to be executed on existing ready queue There are, busy wait To prevent sleeping To manage the state of the thread wait queue It adds.

- Wait queue To be added to the (push) Condition: timer_sleep Added to the calling thread in the queue
- Wait queue Falling in (pop) Condition
    - every tick (Occurring in a timer) Every, waking up the thread (as specified hours sleep Check whether there is a completed).

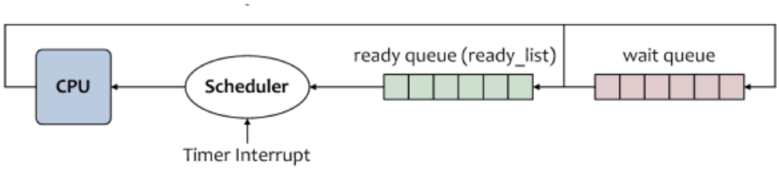

Figure 1: Wait Queue The concept

- Examine awakened thread wait queue in dequeue And, ready queue Again enqueue do.

Data Structures

Each thread sleep If, either tick Till sleep It must store information about what should be. This information is struct thread of sleep_endtick The threads sleep If you wake up with tick It is stored as the.

```
@@ -90,6 +90,7 @@ struct thread
         int priority;                                    / * Priority. * /
         struct list_elem allelem;                        / * List element for all threads list. * /
+        struct list_elem waitelem;                       / * List element, stored in the wait_list queue * /
+        int64_t sleep_endtick;                           / * The tick after which the thread should awake (if the thread is in sleep) * /

         / * Shared between thread.c and synch.c. * / Struct list_elem elem;
                                                          / * List element, stored in the ready_list queue * /
```

Wait queue It is ready_list And similarly wait_list in linked list It is managed by. Which is stored in the list element It is struct thread of waitelem to be.

```
 / * List of processes in THREAD_READY state, that is, processes
     that are ready to run but not actually running. * / Static struct list ready_list;



+/ * List of processes in sleep (wait) state (ie wait queue). * /
+static struct list wait_list;
+
```

avatar: Enqueue

For more information: commit d795340 Make a note.

first, sleep For the thread_sleep_until (int64_t ticks_end) The function was added, the current thread

ticks_end Tick to sleep It serves to. The threads, as described above sleep_endtick Store information and enqueue O 'clock, thread_block () The thread through the block Such that. now timer_sleep Standing the function thread_sleep_until () Using a function (the current time + sleep which tick Number) thread sleep It is when.

```
@@ -92,8 +92,13 @@ timer_sleep (int64_t ticks)
     int64_t start = timer_ticks ();

     ASSERT (intr_get_level () == INTR_ON);
-    while (timer_elapsed (start) <ticks)
-        thread_yield ();
+    enum intr_level old_level = intr_disable ();
+
+    // sleep the thread for `ticks` seconds,
+    // until the tick becomes [start + ticks]
+    thread_sleep_until (start + ticks);
+
+    intr_set_level (old_level);
```

Note that, timer sleep When the interrupt disable It was necessary to. this is timer sleep When an interrupt occurs during wait queue In the heavy load data race Because it can cause.

avatar: Dequeue

For more information: commit 349f854 Make a note.

In order to check the thread wakes up timer For each of the test Matic, the timer For this interrupt service rountine If you are used to. In other words, the timer tick that is called each time change ISR (timer.c: timer_interrupt) There are, where present tick It can be seen kkyae eonal the thread through the information.

wait queue While the whole of the tour, to remove it from the list if you have to wake up the thread thread.c: thread_awake (int64_t current_tick) There is implemented the function. Waking up thread thread_unblock ()
Again through ready queue To be added.

Test Results: alarm-multiple

wait queue Since the implementation alarm-multiple The results of the test case are 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
```

```
(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. Timer: 582 ticks


Thread: 550 idle ticks, 32 kernel ticks, 0 user ticks Console: 2955 characters output
Keyboard: 0 keys pressed Powering off ...
```

In the bottom of each tick There know the ISO model of thread information, busy wait silver idle tick In contrast to the N wait queue Since the implementation, 550 doggy idle tick A problem is that you can see well-behaved. idle tick Iran idle thread

The line reached tick Since the prosthetic, each alarm Waste their sleep While tick (CPU Resources) without consuming block There have been a means waking up at the right time.

```
- Timer: 580 ticks
- Thread: 0 idle ticks, 580 kernel ticks, 0 user ticks
+Timer: 582 ticks
+Thread: 550 idle ticks, 32 kernel ticks, 0 user ticks
```

Task 2. Priority Scheduling and Donation

Design: Priority Scheduling

Thread priority The inde information is required to properly operate, and eventually when the current thread is running, there are specific conditions associated with the current thread priority Chance yield Again by schedule It is key to be.

ready_list Among the threads in the thread, so when the highest priority runs, Considering the conditions

- If the thread is a new generation coming into the queue (thread_create)
- block Is the thread that unblock If you are coming into the queue (thread_unblock)
- The thread that was running yield If you are coming into the queue
- If the priority of the current thread has changed (thread_set_priority)

A.

Also, lock Implementation (semaphore) In priority As it should work. semaphore Structure has, it does not enter the atmosphere (block) doing thread To manage the waiters in list There is, when a thread that is already occupied by a semaphore exit (sema_up) Because one thread can run again unblock It will be. This time out should be the highest priority thread is coming out.

Finally, monitor To implement conditional variables In semaphore And have similar requirements.

conditional variable In waiting semaphore The Figure (struct condition of) waiters in list It is managed by.

To this end, in each case, - (i) ready queue sign ready_list The (ii) semaphore Queue waiters The (iii) condition Queue waiters The - To stay aligned. bracket queue Some sort of priority queue That can be considered to obtain data abstraction Zorro, without the use of particularly complex priority queue implementation (time complexity of inefficiency exists but) Just like the original linked list By keeping the aligned state by being covered and to allow for proper scheduling.

Implementation

Simply stated modifications, one ready_list Whenever you add it to maintain the sort order.

thread_unblock () And thread_yield () Present in, ready_list An additional portion of the,

list_insert_ordered () Using the function thread of priority Sorted in list Gives determined can be inserted into. (comparator_greater_thread_priority The comparison function)

For a high-priority thread to be scheduled (run)

- New thread *t* Been added is generated (thread_create) Current Thread t_0 If more high priority
- Threads *t* end unblock And (thread_unblock), Current Thread t_0 If more high priority
- Threads *t* When I wake up in the queue semaphore (sema_up), Current Thread t_0 If more high priority
- Current Thread t_0 If the priority is changed (donation Also it includes a case receiving), Immediately after the thread run *t* The presence *t* In the case of higher priority

Immediately thread_yield The process helps to make it happen is re-scheduled.

Priority Inversion Problem

Lock If you have priority inversion You may experience problems. Each high / medium / low priority thread *H, M, L* There is, in some lock By *L* It is running and *H* end block State and (lock And independent) *M* end ready list If the, *M* end *L* More is always scheduled first, *H* The (*M* Than the first) It is not running phenomenon. To prevent this priority donation Considering the simplest scenario to be

- Current Thread *H* end lock of acquire When you already lock The biting holder Threads *L* This may *H* The priorities *L* If higher,
    - *L* Of priorities *H* Contribution of priorities should
    - At this time *L* Also you must remember the original priority (struct thread: original_priority)
- (It donated a priority) *L* end lock of release When you try,
    - *L* The memorized priority original_priority Also recovered from the

It acts as a mechanism such as. But the thread priority from one or more threads donate To receive, nesting donation Since this might happen should design a data structure considering it. Its contents are as follows:

Design: Priority Donation

- bracket thread The data is newly stored
  - donation The original priority before downloading (original_priority),
  - what lock By block That the (waiting_lock): this lock Knowing the lock of holder When you see priority You can know all the nested structure of the base (for nested donation)
  - This thread is holding lock The list of objects (locks): Which can be determined by other threads you have contributed to this priority thread (for multiple donation)

```
@@ thread.h (struct thread)
         int priority;                                                       / * Priority. * /
+        int original_priority;                                              / * Priority, before donation * /
@@ thread.h (struct thread)
+        // needed for priority donations
+        struct lock * waiting_lock;                                         / * The lock object on which this thread is waiting (or NULL if not locked) * /
+        struct list locks;                                                  / * List of locks the thread holds (for multiple donations) * /
```

- bracket lock The data that are saved
  - lockelem: thread :: locks To save the list element
  - priority: Now lock of priority This is lock Owned by the thread (holder) of priority This is defined as.

```
@@ synch.h (struct lock)
         struct thread * holder;                          / * Thread holding lock (for debugging). * /
         struct semaphore semaphore; / * Binary semaphore controlling access. * /
+
+        struct list_elem lockelem; / * List element for the thread's 'locks' list. * /
+        int priority;                                    / * Priority of the the thread holding the lock (for priority donation) * /
```

Lock Processing of catch

Threads t end lock l About lock_acquire The verse waiting_lock this l It is set to be. then priority donation chain The next thread l-> holder The priorities t If more low, l-> holder end t Priorities from the donate

under l of priority Donated to the priority t It is set to the priority (according to the definition above).

By the way nested donation To be, l-> holder The following thread (l-> holder It is holding lock of holder) If the priority is still low, Yiwu is also a priority, and then thread donate It should be. We therefore above priority

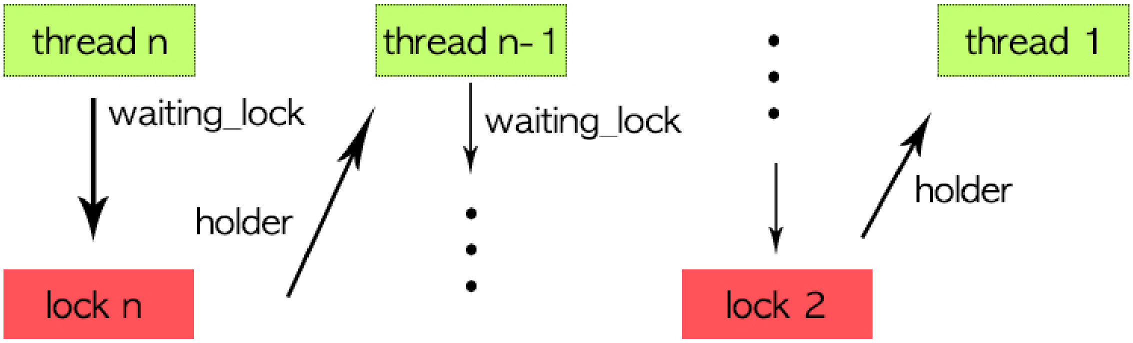

Figure 2: Priority Donation Chain

donation chain Therefore, the t Wow I Move, and **t The priorities t-> waiting_lock-> holder Priority until the higher priority than priority When you donate** do. This lock To occupy running The Threads (chain In the front side) of the thread you want to get a lock on priority Receive donations because priority inversion To be resolved. [One]

First lock I about sema_down Through the semaphore count It is reduced, and finally I of holder Thread locks By updating [2] priority Have the latest update to the list of one thread to donate (it later lock When you turn off, priority The need is to know how, where recovery.

This series of process lock_acquire () There is implemented the function.

Lock When you release the handle

In contrast, the current thread t The Rock I Suppose release. Then one l-> holder And lock list It is set to a data structure, such as properly. now semaphore count Since gajeungga, priority donation And the need to recover. t end priority

There is likely to have received this donation chain The can be nested to follow. therefore locks list Looking for t

to priority To identify the other thread base (this information was stored above)

[One] Omitted strict proof is possible, but through induction.

[2] lock of priority This by defining the sort order in the nested structure lock release Occurrence newly priority That what must be updated to maintain the information. bottom release

See page description.

- priority donor (If you do not have a donate thread) donation chain Since the end of the original_priority The original through priority If you are recovering to.
- priority donor If there are, of highest priority end t New (donated) priority It will be. Therefore, to match this donation To perform again t of priority It makes a change[3] .

This series of process lock_release () There is implemented the function. Note As mentioned earlier in thread donation

If the first priority is changed, through appropriate yield The threads through to haejueoya to run starting with the scheduling with a higher priority.

Test Results: alarm-priority

Priority scheduling After the implementation is completed, alarm-priority The results of the test case are as follows.

```
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': (alarm-priority) 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 ...
```

[3] First, this also must be managed in such priority queue, but a nine symptoms maximum lookup It has been implemented in such a way that the (requires improvement).

This test case is the thread priority 25, 24, · · ·, 21, 30, 29, · · ·, 26 In order to generate schedules. However, because the thread is always the first to run to the priority 30 from 21 To the scheduling priority in order of preference it can be confirmed that the run sequentially.

Conclusion and additional challenges

All trillion won with an understanding of information, discussion, and were involved in coding and debugging / documented. Meet the basic requirements and implementation is completed, the success was all the test cases.

```
SUMMARY BY TEST SET

Test Set                                                   Pts Max% Ttl% Max
---------------------------------------------------------- --- --- ------ ------
tests / threads / Rubric.alarm                               18/18 30.0% / 30.0%
tests / threads / Rubric.priority                            38/38 70.0% / 70.0%
---------------------------------------------------------- --- --- ------ ------
Total                                                                100.0 / 100.0%
```

Some improvements are many. The first priority heap implementation linked list Not the heap or 64 doggy list When implemented (thread priority It is 64 Andoemeuro only type) There can be more efficient, run-time performance critical Not because one is focused on a simple implementation. It will also need to improve on this part, starvation To prevent BSD Scheduler There is more work to implement such projects will be required this will be a very interesting task.