CSEx61 - Operating Systems

Pintos

Phase 1: Threads

Group:

Ali Ahmed Ibrahim Fahmy 18011064 <es-ali.ahmed2018@alexu.edu.eg>

Fares Waheed Abd-El Hakim AbadAllah 18011224 <es-FarisWaheed2023@alexu.edu.eg >

PRELIMINARIES:

For fixed point: we read this stack overflow.

Alarm Clock

A1.Data Structures:

In timer.c, we initialize a list and put sleeping threads in ascending order according to their number of ticks .

In thread.h, there is an attribute for each thread named “total_ticks”, which is assigned for the time it would wake up .

```
/* Waiting queue of timer_sleep */
```

```
struct list sleeping_queue;
```

```
struct thread{

int64_t total_ticks; /* timer ticks thread should sleep + Number of timer ticks since OS booted */ }
```

ALGORITHMS:

A2. In timer_sleep

First we assign the thread total_ticks and add the thread to the sleeping queue in order then block this thread.

In timer interrupt

Go through the sleeping queue in order and if the total ticks of a thread is smaller than or equal to the current ticks then remove it from the sleeping queue and unblock this thread.

A3. As we don't check all the blocked threads since we go through the sleeping queue in order and if the total ticks of a thread is bigger than current ticks break from the for loop.

SYNCHRONIZATION:

A4. As interrupts are disabled since a thread enters timer_sleep then after it finishes the interrupts are enabled .

A5. As interrupts are disabled since a thread enters timer_sleep then after it finishes the interrupts are enabled . since pintos works with a single processor.

RATIONALE

A6. As it is easy to implement and works fast.

We think there is no better design to consider than using the sorted list.

PRIORITY SCHEDULING

DATA STRUCTURES:

B1.

In thread.h :

```
#define DONATION_DEPTH 8 /*As stated in the pintos official
document,Maximum depth of nested donation = 8 */
```

```
struct thread{

struct list locks ;  /* locks are held by this thread */

struct lock* waiting_lock ; /* lock the thread is waiting on to acquire */

int original_priority  ; /*   original priority of this thread (always
updated with thread_set_priority()) */}
```

In synch.h:

```
struct lock

 {

  int priority ;   /*   maximum priority for the threads waiting on this
lock  or the thread holds this lock */

  struct list_elem elem; /* List element. */};
```

B2: As shown above we in thread struct We add original_priority, waiting_lock and list of locks, also in lock struct we add priority of the lock and list_elem to track priority donation.

Nested Donation Example:(Just explain the effect in priority)

D ======== >> lock 1 ===== >> A ====== >> lock 2 ===== >> B ===== >> lock 3 ===== >> C

Assume P: priority of the thread

1. Thread A (P: 50) holds lock 1.
2. Thread B (P: 60) holds lock 2.
3. Thread C (P: 40) holds lock 3.
4. Thread A waits on lock 2.
5. No priority donation is required as B-> P > A->P.
6. Thread B waits on lock 3.
7. Thread B denotes its P to Thread C (C->P = 60).
8. Thread D (Priority 70) waits on lock 1.
9. Thread D denotes its P to Thread A (A->P = 70).
10. Thread A denotes its P to Thread B (B->P = 70).
11. Thread B denotes its P to Thread C (C->P = 70).
12. Thread C releases lock 3 and its P returns to its original (40).
13. Thread B holds lock 3.
14. Thread B releases lock 3 and its priority is still the same.
15. Thread B releases lock 2 and its P returns to its original (60).
16. Thread A holds lock 2.

17.Thread A releases lock 2 and its priority is still the same.

17.Thread A releases lock 1 and its its P returns to its original (50)

18.Thread E holds lock 1.

ALGORITHMS:

B3. We keep the ready_list in order by replacing list_push_back() which was used in thread_unblock() and thread_yield() by list_insert_ordered().

We also consider the cases when the thread is created, the thread is unblocked ,the priority of a thread in the ready_list changes and the priority of the running thread changes to a smaller one by thread_set_priority().In this cases , We should test to preempt the running thread or not by comparing the priority of the front of the ready_list with the running thread priority.

By this way , the scheduler always picks the thread which has the highest priority.

Also Semaphore-> waiters are kept in order according to thread priority by using list_insert_ordered().

Also Thread->list of locks is kept in order according to lock priority by using list_insert_ordered().

B4.void lock_acquire (struct lock *lock) // declaration

1. T = thread_current().
2. Check lock->holder != NULL & !thread_mlfqs (Advanced Scheduler is off): True : do that -

```
T->waiting_lock = lock ;
hold = lock;
/* Nested Priority Donation for Priority Scheduler */
for( d = 0 ; hold!=NULL && d<DONATION_DEPTH &&
t->priority > hold->priority; d++){
hold->priority = T->priority ; /*update the lock priority*/
donate(hold->holder); /* donate the current thread priority to the
lock holder and reorder the holder if it is in the ready_list */
hold = hold->holder->waiting_lock ; /* Check if there is a chain */
}
```

3.Wait on the lock->semaphore using sema_down function.

4. After the thread is waken do that :

```
T = thread_current()
Disable interrupts using intr_disable()
if (!thread_mlfqs) /*  if Advanced Scheduler is off  */{
T->waiting_lock = NULL;
lock->priority = T->priority;/*    Update lock priority  */
add_lock (lock);} /* Add the lock to the thread list of hold locks*/
lock->holder = t;/*    update the lock holder */
```

intr_set_level (old_level); /* set interrupt as the old_level */

B5. void lock_release (struct lock *lock) // declaration

1. Disable interrupts using intr_disable()
2. Check if The Advanced Scheduler is off, remove the lock and update the thread priority either to the original priority or the highest priority in its lock list.
3. lock->holder = NULL.
4. Signal lock->semaphore using sema_up function.
5. intr_set_level (old_level); /* set interrupt as the old_level */

SYNCHRONIZATION

The thread waiting on a lock may donate its priority to the lock->holder and at the same time the lock->holder may change its priority using thread_set_priority().

Solution: disable interrupts which we did in the denote() function. I can't use a lock as in our implementation we didn't provide an interface to make a lock shareable between 2 threads. If we implement it or pintos base code provides it , then we can use a lock to avoid this race.

RATIONALE

It is simple and easy to implement.

I think our solution is good and other solutions like store priority in semaphore struct instead of storing it in lock struct may be more complex.

ADVANCED SCHEDULER

DATA STRUCTURES:

C1.

In thread.h

```
int nice;                   /* Niceness value to calculate priority */

real recent_cpu;    /* Recent CPU value time it takes using cpu. */
```

nice attribute in each thread which represent its nice value , real attribute in each thread which represent its recent cpu value “percent of its cpu usage”

In thread.c

```
real load_avg;
```

Load_avg attribute is the same in all threads which represent Load avg value estimated from previous knowledge.

ALGORITHMS

C2.

	Recent cpu			priority			
Ticks	A	B	C	A	B	C	Thread to run
0	0	0	0	63	61	59	A

4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	A
28	16	8	0	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

C3.

Yes, there will be threads of equal priority.

Solution: work in FIFO principle , where the first thread entering the list will be executed first.

C4.

If the CPU spends too much time on recent_cpu , load_avg and priority arithmetic, it takes most of the time that the thread does before it enforces precautions. Then that thread can't get enough uptime as expected and it will last longer. This will blame itself for taking up more cpu time, raising load_avg and recent_cpu , thus lowering its priority. This may disturb scheduling decision making. Thus, if the scheduling cost increases within the interrupt context, it will result in lower performance.

C5.

Our design did not implement the 64 queues. We used only one queue - the ready-made queue that Pintos originally owned. But like what we did for priority scheduler tasks , we keep the ready list as a priority-oriented descending order at every start - that is, when we insert a thread into the ready list, we insert it in order. The time complexity is O(n). Every fourth tick, it is necessary to recalculate the priority of all threads in all_list. Next, we need to sort the ready-made list, which will take time (O(n lgn). Since we need to do this task every 4 ticks, it will make the thread's running ticks shorter than expected. If n becomes larger, thread switching may occur in Often. If we use 64 ready-made thread queues, we can put 64 queues into an array with an index equal to its priority value. When the thread is first inserted, it only needs to index the queue by the priority of that thread. This will take O time (1) Only. After calculating the priority of all threads every fourth tick, it takes O(n) time to re-enter ready threads. But our implementation is better than this situation - no ready list is requested. As pintos did originally, for every new thread that is not Forbidden, just push back to ready_list. When you need to find the next thread to run, you must rearrange a ready-made list. Sorting takes O(n lgn) time. Needs to repeat this sort as often as we need to call preempt , and after calculating the priorities of all threads Letters every fourth tick.

C6.

In BSD's tab guide, recent_cpu and load_avg are real numbers, but pintos has disabled floating numbers. Instead of using floating numbers, we can use fixed point numbers. So we use fixed point numbers to represent recent_cpu and load_avg.

1. Convert a value to a fixed point value.

Left integer offset for 16 bits. (The fraction bits are all 0)

2. Add 2 fixed point values or replace 2 from fixed point value.

Simply do A + B and A-B

3. Add the fixed point value A and the int value B (the sub-summary is the same).

First convert the int value to the fixed point value, then add them together.

4. Multiply the value of the fixed point A by the value of int B. (Division is the same) reproduce immediately

5. Multiply two fixed point values (equal to division) to avoid redundancy, you first convert A to int_64. After multiplication, go right 16 bits to set the number of bits of the fraction to be 16.

6. Get the correct part of the fixed point value Fixed point right offset for 16-bit

7. Get an integer rounded to the fixed point value If A > = 0, add 0.5 to the value and shift it. If A < 0, subtract A by 0.5 and offset. The last result is rounded to the nearest integer.

We used #define the macro in the new fixed point of the address created within the thread. we You haven't implemented them as inline functions in thread.c because

i. It's simple. Each parameter appears only once in the calculation, which The macro #define error can be avoided, which calculates the same parameter (expression) several times.

ii. It is faster than the built-in functions.