CS302 Project1 Final Report

personal information

- **SID : 11810419**
- **Name： 王焕辰**

Task 1: Efficient Alarm Clock

Data structures and functions

- First, in the thread.h file, add a variable whose type is int64_t in the struct thread:

  ```
  int64_t  t_for_b;
  ```

  which has a use of recording the time of thread has been blocked.

- Second, fix some function to solve the Task as follow:
  - ```
    void timer_sleep();
    ```

    Don't us the loop to do the busy_wait rather than block the thread directly when invoke the **timer_sleep()**, and then uses the OS's own clock interrupts (executed once per tick) to check on the state of the thread, reducing ticks_blocked by 1 each time, and waking the thread if it drops to 0. Then call
  - ```
    void timer_interrupt()  ;
    ```

    In this function add the **thread_foreach** (check_block, NULL) to operate the all thread in the **all_list** .
  - ```
    void check_blocked();
    ```

  which make the blocked thread's tick minus 1 and make them unblock if their ticks equal to 0.

- Besides, there is a operation to ensure the atomicity of the action, which can let the interrupt flag be 0 during do this action:

  ```
  enum intr_level old_level = intr_disable ();
  ...
  intr_set_level (old_level);
  ```

- After finish above step, the test alarm-priority still FAIL, which will be fix in the **Task 2**.

Algorithms

- the Algorithm part will be especially talked in TASK2 and TASK3.
  ```

Synchronization

- The synchronization part will be especially talked in TASK 2.

Rationale

- Before I modified the code, the original code rationale is :**timer_sleep**() let the thread continually go to the ready_list if its STATUS is RUNNING. the **thread_yield**() just make current thread go to the ready_list and do the **schedule**() again. **schedule**() just switch the next thread to run.
- To avoid the busy waiting, should use a awake method to forbid the thread change between ready_list and running list, which realization function has been claim in the **Data structure and function part**.

TASK2 : Priority Scheduler

Data structure and function

- First , to deal with the test: alarm-priority, will use
  - the **ready_list** and **all_list**, which is a **struct list** and has been declared in the original pintos code in the thread.c file

    ```
    static struct list ready_list;
    static struct list all_list;
    ```

    to the **ready_list** is the List of processes in THREAD_READY state, that is , processes that are ready to run but not actually running.

    to the **all_list** is the List of all processes. Processes are added to this list when they are first scheduled and removed when they exit.
  - the member variable **priority** in the struct **thread**

    ```
    int priority;
    ```

  - Then, just maintain the **read_list** is apriority queue during the whole process of scheduling. Thus, use the function already exist in the list.c: **list_insert_ordered()** and create a new cmp function for sort

    ```
    void cmp_thread();
    ```

    is OK. The detail of it will be talked in the **Algorithm** part.
- Second, to pass the test priority-preempt and priority-change which is the basis of the synchronicity and the thread priority donation. just judge the priority, and let the lower priority thread to execute the **thread_yield()**
- Third, to solve the priority donation problem. add some variables and data structures as follows . Also fix and add some functions:
  - ```
    int original_priority;
    struct list locks;
    struct lock *lw;
    ```
    ```

They are the original priority before the donation, a List record which lock this thread is holding and the lock that the thread is waiting for.

  - ```
    struct list_elem lock_elem;
    int max_priority;
    ```

    They are the List element for the priority donation and the max priority in the threads acquiring the lock.

  - ```
    void lock_acquire();
    ```

    which do the operate of semaphore POST operation to acquire the lock. I fix it in the way that before do the POST operation, do the donation recursively until be woken and be the holder of the lock.

    ```
    void thread_set_priority();
    ```

    to this function, need to let the current thread get the new priority or keep it In the case of donation.

    ```
    void donation_update();
    ```

    which update the priority with the donation.

    ```
    void cmp_lock();
    ```

    which used as a cmp function to do the list sort in the donation process.

- Last, to the synchronicity , fix the function **cond_signal() ， sema_up() and sema_down()** and add a function:

  ```
  void cmp_cond_sema();
  ```

  which used as a cmp function to do the list sort in the semaphore priority queue.

  ```
  void cond_signal();
  ```

  which should sort the semaphore and do the **sema_up()** to wake thread.

  ```
  void sema_up();
  void sema_down();
  ```

  are the WAIT and POST operation of the semaphore to realize the synchronization with a priority also.

Algorithm

To the algorithm.

- basic priority : When setting a thread priority, immediately reconsider the execution order of all threads and rearrange the execution order. then pass the alarm priority test.
- **priority donation**: To solve the priority inversion problem. should let the holder of the lock's thread priority = higher priority thread which need this lock.

- To the simple case: When a thread acquires a lock, it increases the priority of the thread that owns the lock if it has a lower priority than itself, and then changes the priority of the thread that originally owned the lock back to its original priority after the thread releases the lock.
- To the multiple case: When a lock is released, the owner of the lock is changed to the second priority that the thread was donated to, and if there are no other donors, the original priority is restored.
- To the priority embedding problem: the key is that the lock owned by the medium is low and low. If the medium is obtained after the high in this serial, the priority enhancement has a chain effect, that is, the medium is promoted, and it is bound by the lock at this time. Low thread should be promoted along with it.

Synchronization

This part we focus on the semaphore with some functions.

In the pintos, the V operation **sema_up()** function: just pop the front of the waiter and add to the **ready_list**, and let this thread unblocked. but this can be synchronize but without priority. Thus, should select the max_priority thread in waiter.

Corresponding, **sema_down()**, just push the thread in to the waiter, without other operation, which need use the insert_ordered to add the thread into the waiter. not only realizes the synchronization but maintain the priority.

Additionally, the condition part is use for do some action when receive some condition. To the **cond_siganl()** which is an interrupt handler cannot acquire a lock, so it does not make sense to try to signal a condition variable within an interrupt handler. To improve it and pass test, make the condition waiter also has the priority is OK.

Rationale

To the priority donation, the key is let the priority low but hold the resource thread release the resource faster. Thus, let the higher priority donation the priority to it.

Task 3 : Multi-level Feedback Queue Scheduler

Data Structure and function

- First, use the fixed_point type to solve the problem of no float in the pintos to calculate the priority in the multi-level feedback queue scheduler.
- add some functions and variables, according to the document with the formula.
  - add to the thread:

    ```
    int nice;
    fixed_t recent_cpu;
    ```

    the nice is f(t) can be a constant or some other values .
  - add some functions to calculate the Priority, recent_cpu and load_avg:

    ```
    void check_mlfqs_priority();
    void update_load_avg_and_recent_cpu ();
    ```

to realize: **priority = PRIMAX - (recnet_cpu/4) - nice *2**

recent_cpu = 2*load_avg / (2 *load_avg +1) * recent_cpu + nice.

load_avg = 59/60 * load_avg + 1/60 * ready_threads

the ready_threads is the size of the ready_list right now.

- complete the function of :

```
void thread_set_nice (int new_nice);
int thread_get_nice();
fixed_t thread_get_load_avg();
fixed_t thread_get_recent_cpu();
```

Rationale

Use the multiple level feedback queue scheduler may solve the starvation problem. Because it allows processes to move between queues. If a process uses too much CPU time, it will be moved to a lower priority queue. If a process waits too long in a lower priority queue, it will be moved to a higher priority queue to prevent starvation .

Design Document Additional Questions

- (This question uses the MLFQS scheduler.) Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a recent_cpu value of 0. Fill in the table below showing the scheduling decision and the recent_cpu and priority values for each thread after each given number of timer ticks. We can use R(A) and P(A) to denote the recent_cpu and priority values of thread A, for brevity.

timer_ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(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	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

- Did any ambiguities in the scheduler specification make values in the table (in the previous question) uncertain? If so, what rule did you use to resolve them?

  Yes. Because if there are two or more thread has the same priority, may let the schedule is not certain which will run next.

I think it can just use the recent_cpu to do the compare again, since recent_cpu represent the value of the amount of CPU time a thread has received "recently." Thus, if its recent_cpu is lower, to avoid the starvation appearing, choose the lower recent_cpu to run is OK.

- Answer questions about pintos source code
  - a) Tell us about how pintos start the first thread in its thread system (only consider the thread part).

    just invoke the thread_start(). in the thread/thread.c file.
  - b) Consider priority scheduling, how does pintos keep running a ready thread with highest priority after its time tick reaching **TIME_SLICE**?

    just invoke the **thread_yield()**, which use the schedule() function and use the insert_ordered to ensure the front of the ready_list with max priority.
  - What will pintos do when switching from one thread to the other? By calling what functions and doing what?

    The switch happen in the schedule(), the answer is the process (过程) of it.
    - invoke the running_thread() and next_thread_to_run() to get the current thread running and decide the next thread to run.
    - Then use the switch_thread() to to make current thread to be the previous thread and next thread to run if current and next is not same .
    - In the switch_thread(), the assembly language is use for saving the current thread state and restores the thread state saved before the new thread.
- How does pintos implement floating point number operation.

  In the document of the pintos, use the fixed_point type to represent the float point as follows:

  First, use the fixed_point, which make the fractional part in the right 16 bits by do the left shift to the integer.

  The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table, x and y are fixed-point numbers, n is an integer, fixed-point numbers are in signed p.q format where p + q = 31, and f is 1 << q`:

Convert n to fixed point:	**n*f**
Convert x to integer (rounding toward zero):	x / f
Convert x to integer (rounding to nearest):	(x + f / 2) / f if x >= 0 (x - f / 2) / f if x <= 0.
Add x and y	x + y
Subtract y from x	x - y
Add x and n:	x + n * f
Subtract n from x:	x - n * f
Multiply x by y:	((int64_t) x) * y / f
Multiply x by n:	x * n
Divide x by y:	((int64_t) x) * f / y
Divide x by n:	x / n

- What do priority-donation test cases(priority-donate-chain and priority-donate-nest) do and illustrate the running process?
  - priority-donate-chain
    - which is a chained priority donation, essentially testing the correctness of the multi priority donation logic.
    - The test created 15 threads, eight of which chained waiting for a lock to make a chained priority donation;Seven other threads with different priorities are interspersed.
  - priority-donate-nest
    - The key point is that the lock held by medium priority is blocked by lower priority. In this case, the lock held by higher priority can obtain the lock of medium priority. If the lock is blocked, the priority promotion has a cascading effect, that is, the lower thread bound by the lock should be promoted along with it.
    - use the test as example:

      The main thread has priority of PRI_DEFAULT and owns lock a .create thread medium with priority PRI_DEFAULT+1, which hold the lock b and need to acquire the lock a. Then the main thread creates thread HIGH with priority of PRI_DEFAULT + 2 thread high needs to acquire lock b.

Reference

https://mislove.org/teaching/cs5600/fall10/pintos/pintos_7.html

CS302-OS-Project1_2021.pdf (the content is nearly same with above.)