Document of OS project 1

WARN:We gen this doc automatical by google translator, and we won't take responsbilty for that.

Table of Contents

Sub-division of the composition and Team members

4p team members from the four 08cs, the list below
Chen Song, C, <olivia200705@126.com>
Li Meng, L, <mayli.he@gmail.com>
Wang Sheng, W, <wstnap@gmail.com>
Xiaotian Jun, X, <xiaotj1990327@gmail.com>
Codenamed the first letter of each name

Version Control

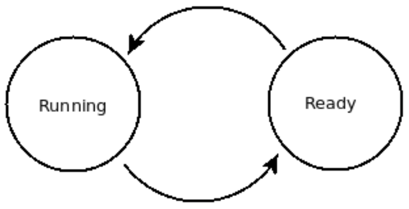
As several people contribute code, so I decided to use version control in the project.
Code hosted on googlecode, using SVN control, you can find project information in the following pages:
http://pintosof4p.googlecode.com
Additional References
The course of the job is complete compressed package in addition to the information, also referred to the
pintos official documents: http://www.scs.stanford.edu/10wics140/pintos
minix3 Source: http://www.minix3.org/source.html

Sleep and wake

This section focuses on the idea by the LXW, W to achieve, L to make the test proposed amendments.

3.1 Requirements Analysis

The original method to hang a thread is implemented in a while() in timer _sleep(). When the waking time is not coming, the thread will call thread_ yield to loop, if the time is coming, we push the thread into ready_list. It actually has only two status:Running and Ready. There is no real sleep and wake_up mechanism. As shown:

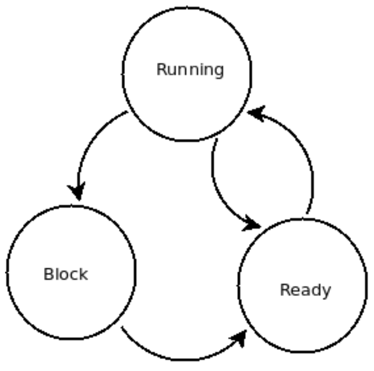


As the while () loop, continuously for thread_yield () operation. By viewing the source code thread_yield () function Note:

/ * Yields the CPU. The current thread is not put to sleep and
may be scheduled again immediately at the scheduler's whim. * /

It also verified the above mentioned thread and did not really sleep, and the thread may be scheduled once again, this result is to generate a busy wait.

We need to achieve the goal, in timer_sleep () function is thread into the Block state manipulation. System running for some time, sleep time, and then wake up the thread from the Block into the Ready state state. After transformation, the mechanism shown:



3.1 Data Structure

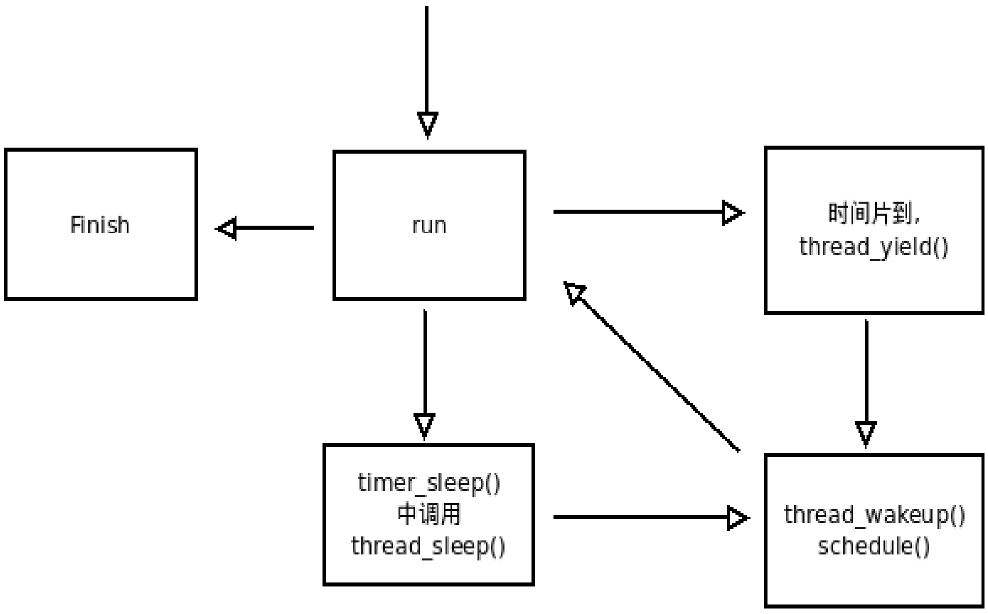
Although the thread.h file enum thread_statues {} has provided thread_block state, and existing ready_list the list, but in the source code, there is not a known block_list the list structure.

So the data structure is added:

    static struct list block_list;

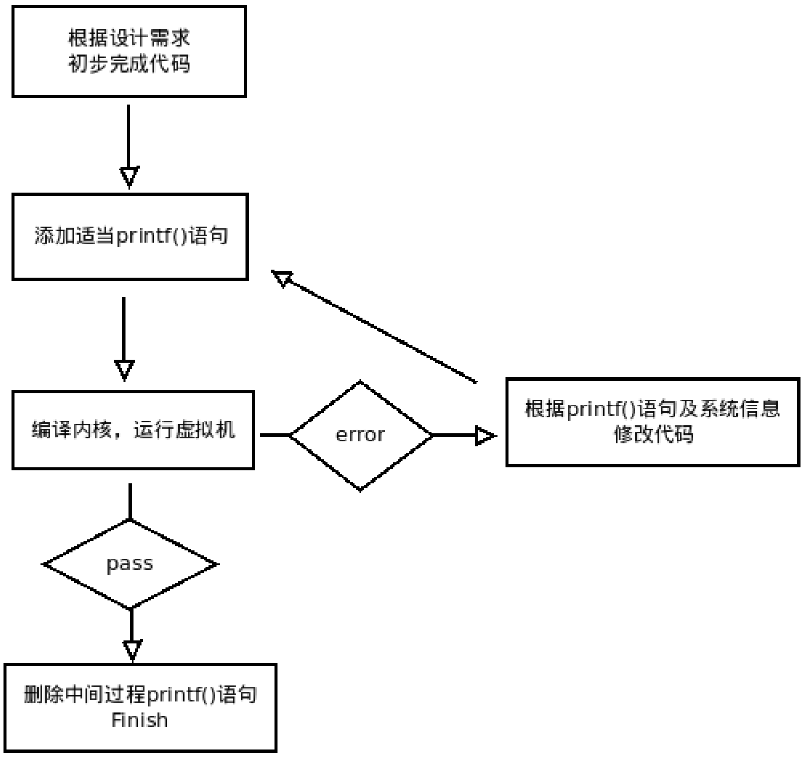
is used to store block state in the thread.

With block_list can be used to store block state in a thread. And sleep has a time limit, when the sleep time, the operation wake up should be carried out. However, struct thread {} structure,does not record the variables related to sleep time.
Therefore, in the transformation of struct thread {},we add:
int64_t wakeup_ticks
Implementation of wake-up operation to record the time, the current time is greater than the wakeup_ticks a thread when the thread should be waken up.
3.2 Algorithm Design
Existing timer_sleep () function, the while () {} loop to directly call the thread_yield (), to avoid busy waiting, need to make the thread to sleep, the following question is, how to wake up thread,further more when to wake up .
Carefully analysis we could know:
Sleep should be implemented in timer_sleep () , and the wake-up time at first glance there seems to more than one choice
1. After each tick, check block_list, if the current time is more than wakeup_ticks ,we wake it up.
2. Wake up a thread when there is a running thread releases the cpu.
If each tick have to wake up after the check block_list, there may state that all the threads in the Block are not the end of the sleep situation, even if some thread is end of sleep, were awakened, if not for scheduling, thread after wake-up Running into the state have no chance, so the efficiency is too low. So we choose to check before making the thread scheduling block_list, the wake-up operation.
In the specific implementation, in thread.c
thread_sleep () function turn thread into the Block state
thread_wakeup () function checks block_list, then wake-up
Summarized as follows:

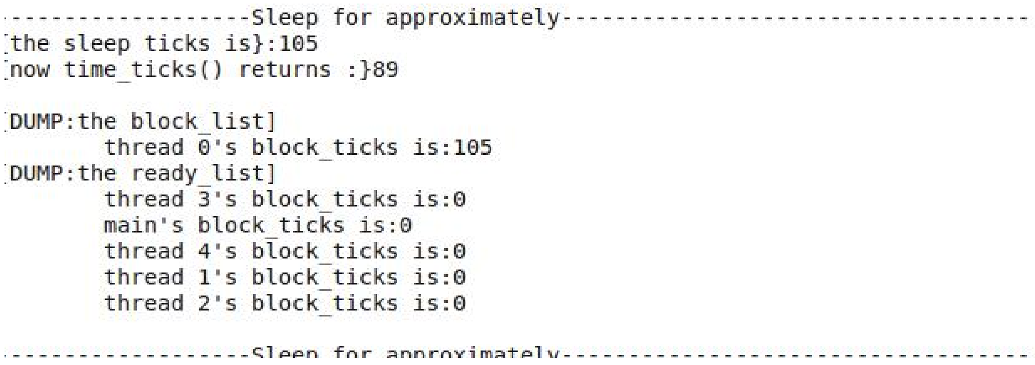


3.2 Debugging Steps

In the code execution process, when the system is running error, lack the necessary message, thus causing inconvenience for debugging. In order to debug the code but also to understand the implementation process, during the commissioning phase we add a lot of printf () statement. Specific steps of the process as shown:



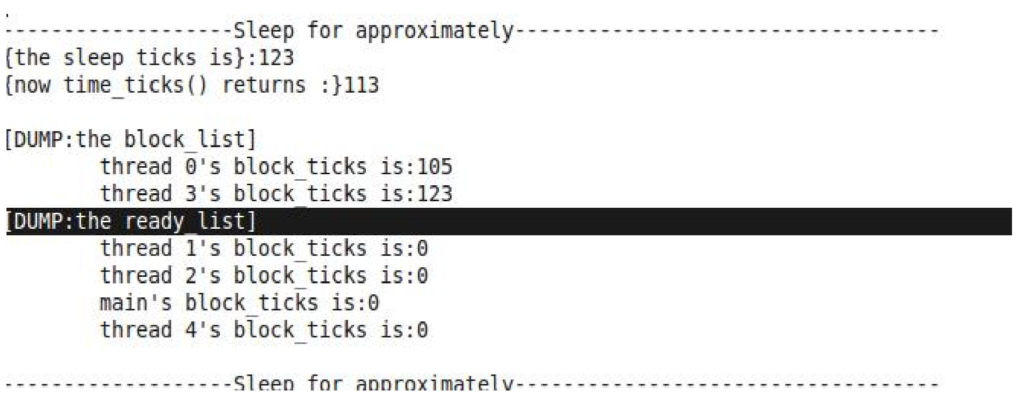
Debugging process printf () output as follows:



```
------------------Sleep for approximately----------------------------------
{the sleep ticks is}:105
{now time_ticks() returns :}89

[DUMP:the block_list]
        thread 0's block_ticks is:105
[DUMP:the ready_list]
        thread 3's block_ticks is:0
        main's block_ticks is:0
        thread 4's block_ticks is:0
        thread 1's block_ticks is:0
        thread 2's block_ticks is:0

------------------Sleep for approximately----------------------------------
```



```
------------------Sleep for approximately---------------------------------
{the sleep ticks is}:123
{now time_ticks() returns :}113

[DUMP:the block_list]
        thread 0's block_ticks is:105
        thread 3's block_ticks is:123
[DUMP:the ready_list]
        thread 1's block_ticks is:0
        thread 2's block_ticks is:0
        main's block_ticks is:0
        thread 4's block_ticks is:0

------------------Sleep for approximately---------------------------------
```

Debugging problems:
1. After thread_sleep () implement the blocking of the thread, continue to check block_list, and then calls schedule () for thread scheduling. But we misses the point that not only in the thread_sleep () we call schedule () for scheduling. In thread_yield () is called the schedule () function, and we did not thread_yield () function will check block_list, to wake up. Solution to the problem is to add the wake-up code thread_yield () to check block_list.
2. On the understanding idle_thread
   / * The idle thread is initially put on the ready list by
   thread_start (). It will be scheduled once initially, at which
   point it initializes idle_thread, "up" s the semaphore passed
   to it to enable thread_start () to continue, and immediately
  blocks. After that, the idle thread never appears in the
   ready list. It is returned by next_thread_to_run () as a
   special case when the ready list is empty. * /
idle thread is the first to enter a ready_list, after the idle thread does not appear in the ready_list. At first, treat it like a normal thread. In the thread.c file thread_tick (), if the current thread is idle thread, the action just for idle_ticks, we make a mistake to rewrite it as: idle_ticks> wakeup_ticks, do wake up. In fact idle thread in thread_start () did not enter after the ready list, the natural sleep and wake up so there will not be operating.

3.3 The optimization process

From the original correct code to the final edition, There are two optimization
1. Thread_block_ticks initially used thread_block_start were recorded with the state entered the block of time, and the thread in the block of time, when the time to run to the greater than or equal thread_block_start thread_block_ticks that should be awakened. Therefore this problem can be simplified for the use of wakeup_tick directly addressed one less variable.
2. At first thread in each state from the Running state when entering Block, directly into the queue with the implementation of operations into block_list the last position, and when the wake-up time, every time they must traverse block_list, so the time cost to a large operation a lot. If block_list in the press wakeup_ticks () to sort on, then wake up when the operation is not required to traverse each time, when the check to a thread wakeup_tick does not meet the wake-up conditions, not continue to follow-up to Block State check the thread, so to reduce the time cost for the purpose.

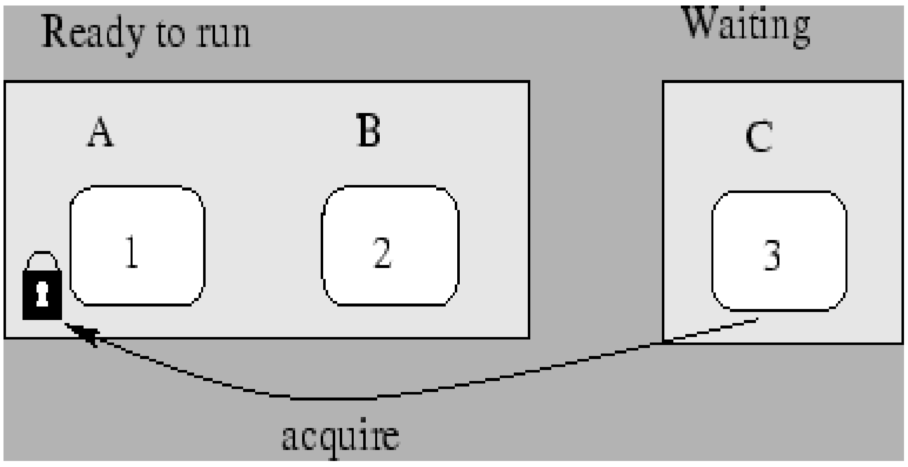
Priority Scheduling

This section focuses is on the idea by the L, L to achieve, X test and suggest modifications.

Requirement

Priority scheduling requirements to implement a priority-based scheduling strategy. Including the resolution of priority when the donate and other donate-related issues.

Priority flip described below:



Thread A, B, C respectively, with 1,2,3-priority (the higher the number the greater the description of priority), thread A, B is currently waiting in the ready queue scheduling, thread A thread has a mutex lock resources. By this time, high-priority thread C also want to visit this exclusive resource, the thread C had to wait on this resource, can not enter the ready queue. When the scheduler starts scheduling, it is only from the A and B to choose, according to the principle priority scheduling, thread B will be the first run.

At this time a problem arises, namely, C would have priority thread than the thread of high B, but thread B is the first run, resulting in flip-priority issues.

Data structure

Synch.h Add / Modify

```
/ * Lock itself on the basis of the original lock added a id, the value used to
distinguish the various locks, similar to the tid * /
struct lock
  {
    / * L: add an lid to identifer the lock * /
    int lid;
    struct thread * holder; / * Thread holding lock (for debugging). * /
    struct semaphore semaphore; / * Binary semaphore controlling access. * /
  };

/ * An element of the chain table, the list holds all the locks may lead to
donate * /
struct lock_elem
  {
    / * L: lock id * /
    int lid;
    struct lock * lock; / * This lock * /
    struct list_elem elem; / * List element * /
  };

/ * Lock the list of variables * /
struct list lock_list;
```

threah.h Add / Modify

```
struct thread
  {
    ... ...
    int priority; / * Priority. * /
    int priority_old; / * L: old priority stores the pri before donated * /
    struct list_elem allelem; / * List element for all threads list. * /
    ... ...
    / * Owned by thread.c. * /
    unsigned magic; / * Detects stack overflow. * /
  };
```

Algorithm

The above data structure depends on the following function implementation, where their code is not given, but will give them back the process, the specific details, refer to Source:

thread.c
/ * Auxiliary functions for the two worker threads in order of priority list, the operation * /
list_less_func priority_higher;
list_less_func priority_lower;
/ * Maybe has a helper function to aid in synch in lock_list and cond of the sort order of the priority list * /
list_less_func lock_lower;
list_less_func cond_lower;
/ * This function is maintained in priority order a thread into the ready_list, alarm-priority-related * /
void thread_unblock (struct thread * t);
void thread_yield (void);
/ * This function only changes priority_old, priority-change related * /
void thread_set_priority (int new_priority);
/ * This function checks whether the lock has a higher priority threads waiting * /
bool donate_check (void);
/ * This function is the waiters of the highest priority to the lock holder, that donate, priority-donate-* relevant * /
void donate_do (void);
/ * Will be called each time when scheduling the above two functions * /
static void schedule (void);
synch.c
/ * Get the lock function, and if you can donate the situation, the priority granted, and then blocked into the waiting list * /
lock_acquire (struct lock * lock);
/ * Release the lock function, and recovery priorities, this will not be granted again * /
lock_release (struct lock * lock);
/ * If you can not down, put the thread to wait by priority into the list of high priority in the list end * /
sema_down (struct semaphore * sema);
/ * Semaphore rising, and unblock those who wait for the biggest priority thread * /
sema_up (struct semaphore * sema);
/ * Cond wake-up, wake up here, waiting for the list of highest priority, priority-condvar related * /
cond_signal (struct condition * cond, struct lock * lock UNUSED);
Solution Description

Description:
Here only a relatively simple data structure (variable lock_list) to achieve the various operational priorities, the implementation of the method is relatively simple idea, one way to solve all the test case, realize it is not complicated,

but also the more obvious defects it does not solve the test_case methods one by one and high efficiency. In particular, there are many locks in place but no priority awarded when a certain amount of computing power is consumed. But this method is still robust and effective. Achieved by this method is that it is indeed a very good reason for compliance with the KISS principle.

Overall implementation description:
Relevant requirements of all priority divided into three groups:
Sleep wake of priority: alarm-priority
Granted priority: priority-donate-*
Priority scheduling of the fairness and reasonableness: priority-[sema | fifo | condvar]

Implementation of the first group, only need to control can be an orderly queue ready_list. Wake_up and thread_yield used here into the queue for control of the queue by next_thread_to_run control.

The realization of the second group need to solve a set of lock system. Including the functional requirements are granted priority, priority to recover, was awarded the priority of the thread priority of trying to change their situation, complex cases priority multiple / chain-type grant and withdraw.
Implementation of this group to the core of a lock table data structure / algorithm to provide. Application for any user will be locked down here, when another thread to apply the lock, the detection of its priorities, to determine whether to grant, and then into the waiters queue is blocked. And the scheduling priority is related, so donate when scheduling inspection and granted, there occurred the priority granted to all the locks in the end produces a high priority to grant work, which ensures the correctness of chain-type grants, while In the case of multiple implicit here is solved. donate-check and donate-do is the solution of the core operation of the priority granted by the Zhelia functions are also implemented.

The realization of the third group, only need to sema / ready_list / cond operations in strict priority and FIFO high to achieve the not complicated, not repeat them here, please refer to the following specific functions sema_up / thread_unblock / cond_signal .

The main function of the work flow
A typical lock_acquire () and sema_down () process

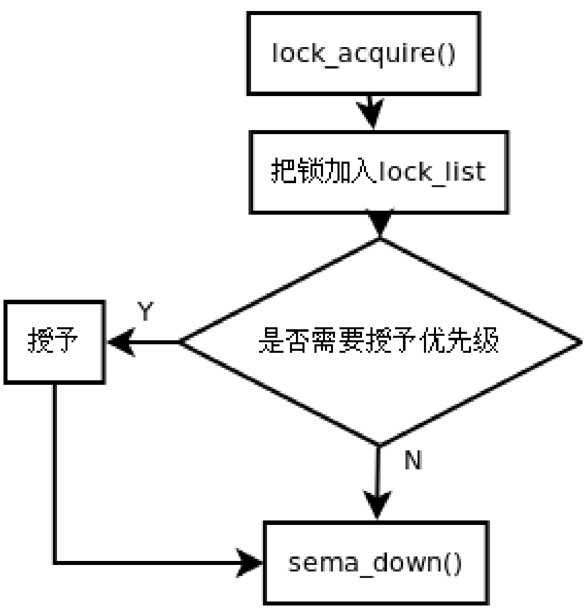


Figure1: lock_qcuire()

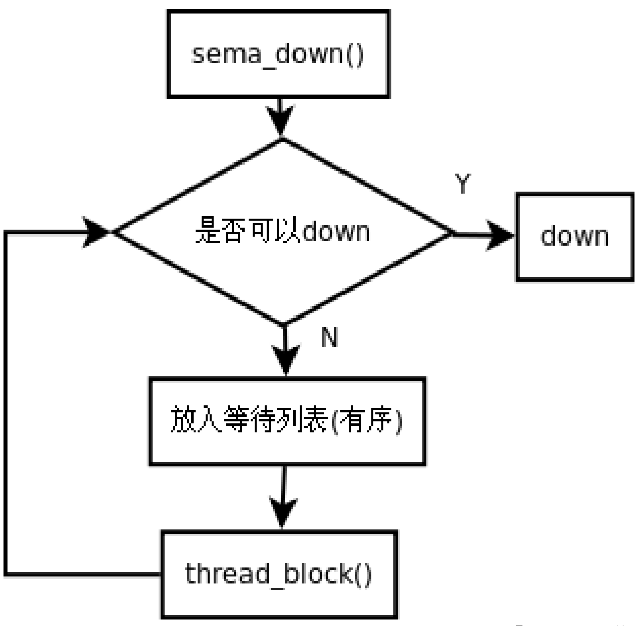
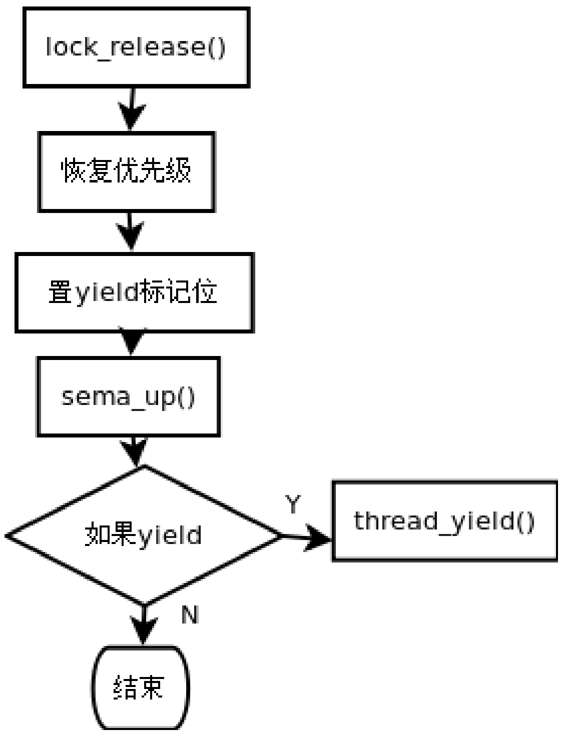


Figure2: sema_down()

A typical lock_release () and sema_up () process



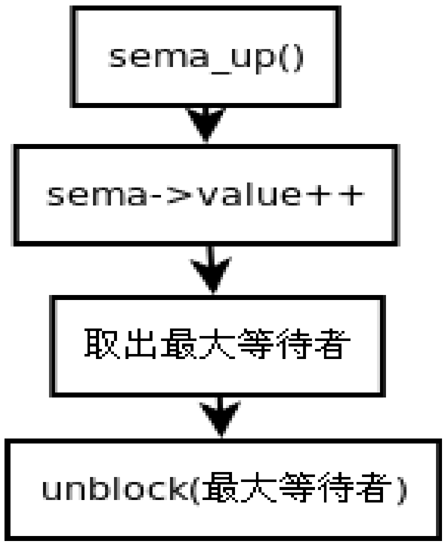


Figure3: lock_release()

Figure4: sema_up()

Handling the two functions donate

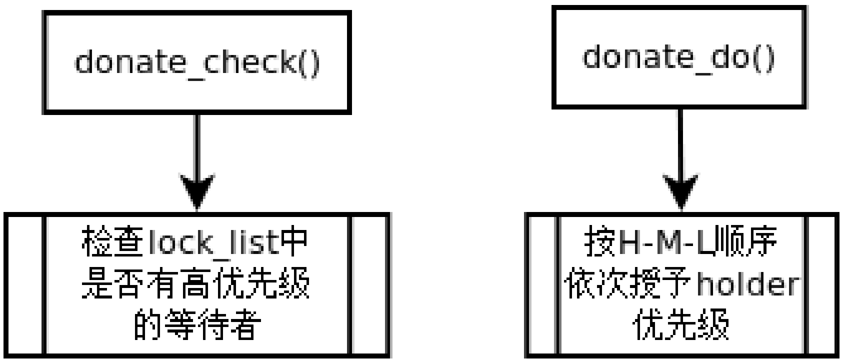

donate_check()
检查lock_list中是否有高优先级的等待者
donate_do()
按H-M-L顺序依次授予holder优先级

Figure5

A typical testcase: priority_donate_nest
The following table shows the first lock from the main access to the program the end of the process

	Main	***medium***	***high***
1	Get lock a		
2	Create thread medium		
3		Lock acquire b	
4		Try acquire a blocked，main->32	
5	yield()		
6	Create thread high		
7			Try acquire b blocked，medium->33，main->33
8	yield()		
9	Release a，back to 31		
10		Release a，yield，release b，back to 32	
11			Get b，release b，quit
12		quit	
13	quit		

Need to clear a few questions:

Waiting for a lock to ensure the highest priority thread wakes up first?
We use an ordered list of waiters to save all the blocked thread and always wake up in the wake when the highest priority.

When lock_acquire is called what?
If the lock is locked into the waiters list, the thread will be blocked, there are related in thread_block obstruction after treatment.

When such a high priority when the lock is lock_release what?
The operation is to unlock the place, and then waiters list of the highest priority thread_unblock, and determine whether to allow cpu to higher thread_yield thread.

Another solution:
Another way to optimize the solution is only lock_acquire and lock_release priority when granting and recovery operations, it needs to provide a thread structure in the chain table, abandoned a lock table shared data structure, a certain degree of efficiency. And when a thread crashes, all threads waiting for the lock will be lost.
However, the disadvantages of doing so is to make the process control system with two sets, one thread of the schedule, the second is synch the lock system, I think that in a system if there are two control system, should not be a good basis for the design, Therefore, the solution is actually used in the schedule to achieve almost all operations, and as little as possible to achieve in synch in them, so schedule a core of the system, but also very much consistent with KISS principle.
Debugging process
As more entries in this group test, where only the typical description of the test sample

priority-change
priority-preempt
This set of tests is not much problem in debugging, the first cause that must be added priority_old thinking.

priority-fifo
priority-sema
priority-condvar
This set of tests were awakened in the order reflected: 1 strict priority 2 rotation, in the debugging and there is no serious problem.

priority-donate-one
priority-donate-multiple
priority-donate-multiple2
priority-donate-nest
priority-donate-chain
priority-donate-sema
priority-donate-lower
Donate this group of tests are related to the test, which is a typical nest, multiple2, the idea of the method is relatively clear, so the actual coding is not too much technical content, but there are several negligence multiple2 strange phenomenon occurred. Changes in X with the help of the code, and do some fine-tuning the details.

Title of completion of the
pass tests / threads / priority-change
pass tests / threads / priority-donate-one
pass tests / threads / priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests / threads / priority-donate-nest
pass tests / threads / priority-donate-sema
pass tests / threads / priority-donate-lower
pass tests / threads / priority-fifo
pass tests / threads / priority-preempt
pass tests / threads / priority-sema
pass tests / threads / priority-condvar
pass tests / threads / priority-donate-chain
* Note: In default mode, sometimes there will be no "Powering off" message in error, we are convinced that this is due to Bochs problem in this mode, make check test in this situation may occur when, in the "pintos-v --q run priority-* "the output and verify the same test script.

Advance Scheduler

I. Needs analysis

Common thread scheduling method is based on time slice cycle priority scheduling, which can be achieved simply and convenient. This method reflects the strict obedience in priority. This scheduling strategy has certain advantages, however, in some cases, there are some disadvantages. The thread scheduling strategy can easily result in hunger when there are threads with higher priority and required long running time slices , the rest of the threads can only regain the CPU after it complete, this scheduling policy does not take the system throughput into account, and does not reduce the average waiting time for the thread. The disadvantages will be amplified when run in a interactive system with a large volume of users, causing users to wait a long time will reduce the user's evaluation of the system. Therefore, we need a new scheduling policy that considers the thread priority basis, in accordance with the operation of the thread to dynamically modify the thread priority, so the implementation of multi-level feedback queue scheduler came into being.

II. Design

1. Multiple ready queue:

We need 64 queue,each represents a priority, CPU always search the first nonempty queue from the priority 0f 63 to 0, then under the FIFO principle, the exit from the queue out the thread, sub- to the time slice to run, as shown.

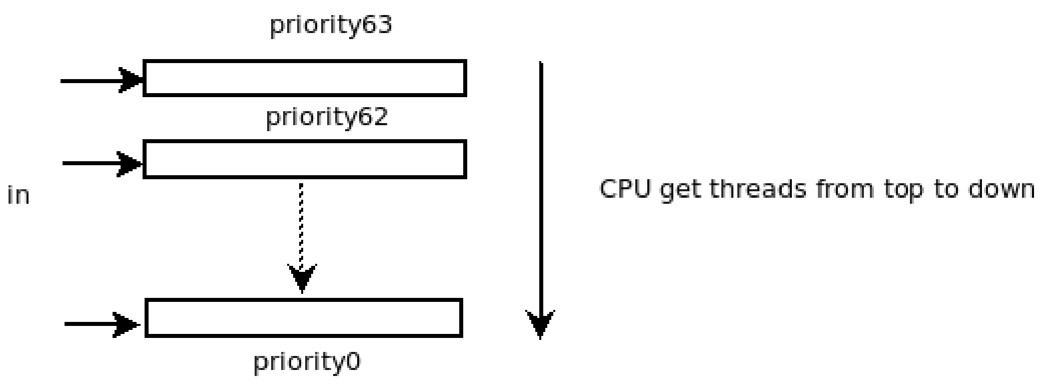


Figure 3-1

2. Records thread operation:
Corresponding data structure is designed to record a recent thread CPU-position of the thread to other threads friendliness, the recent operation of CPU load, etc., as basis for adjusting the priority of the thread.

3. By recording the information to modify the thread running thread priority
Modify the thread priority aim is to prevent the thread excessive hunger, due to lower accounts for the recent multi-thread iCPU priority, an appropriate increase in the near future less CPU-priority thread to achieve a dynamic equilibrium.

III. Implementation Detail

3.1 Data Structure

1.nice:
Basic information: thread structure member variable, record the friendliness this thread to other threads. Popular description is that the willingness of the thread by setting a higher nice value to low their priority, for the other threads to run more time slices.
Data Type: int
Value range: -20 ~ 20, the value is declared to reduce the priority.
Default: 0
Involved in the function:
 int thread_get_nice (void)
 void thread_set_nice (int new_nice)
 void renew_priority (struct thread * t)

2.recent_cpu:
Basic information: thread structure member variable, the recent record of the CPU-thread case, more recent CPU-priority should be taken into account to lower the priority of the thread.
Data types: fixed-point real number
Range of values :-(2 ^ 31-1) / 2 ^ 14 ~ (2 ^ 31-1) / 2 ^ 14
Default: 0
Function involved
int thread_get_recent_cpu (void)
void renew_recent_cpu (struct thread * t)

3.load_avg

Basic information: static global variable, the average load for the running system, estimated the average the number of threads loaded in the system over the past one minute
Data types: fixed-point real number
Range of values :-(2 ^ 31-1) / 2 ^ 14 ~ (2 ^ 31-1) / 2 ^ 14
Default: 0
Involved in the function:
int thread_get_load_avg (void)
void renew_load_avg (void)
void renew_recent_cpu (struct thread * t)

4.advanced_ready_list [64]
Basic information: global variables, multilevel feedback queue, the subscript corresponding to the priority of the thread;
Data Type: struct list
Description: The implementation of the feedback process scheduling queue, I think that only a single ready queue can perform the same function, I made reference to the project's official documentation pintos (http://www.scs.stanford.edu/10wics140/pintos) , described above, as long as the performance of the same operation, you can use a single queue for processing, I write in the program during the course of a single queue and have tried multiple queues,the comparison of the performance on the single-queue and multi-queue implementation will be noted later I this file, finally in the submission edition I reserve the single queue implementation.

3.2 Algorithm Design and function implementation
Overall algorithm demo:

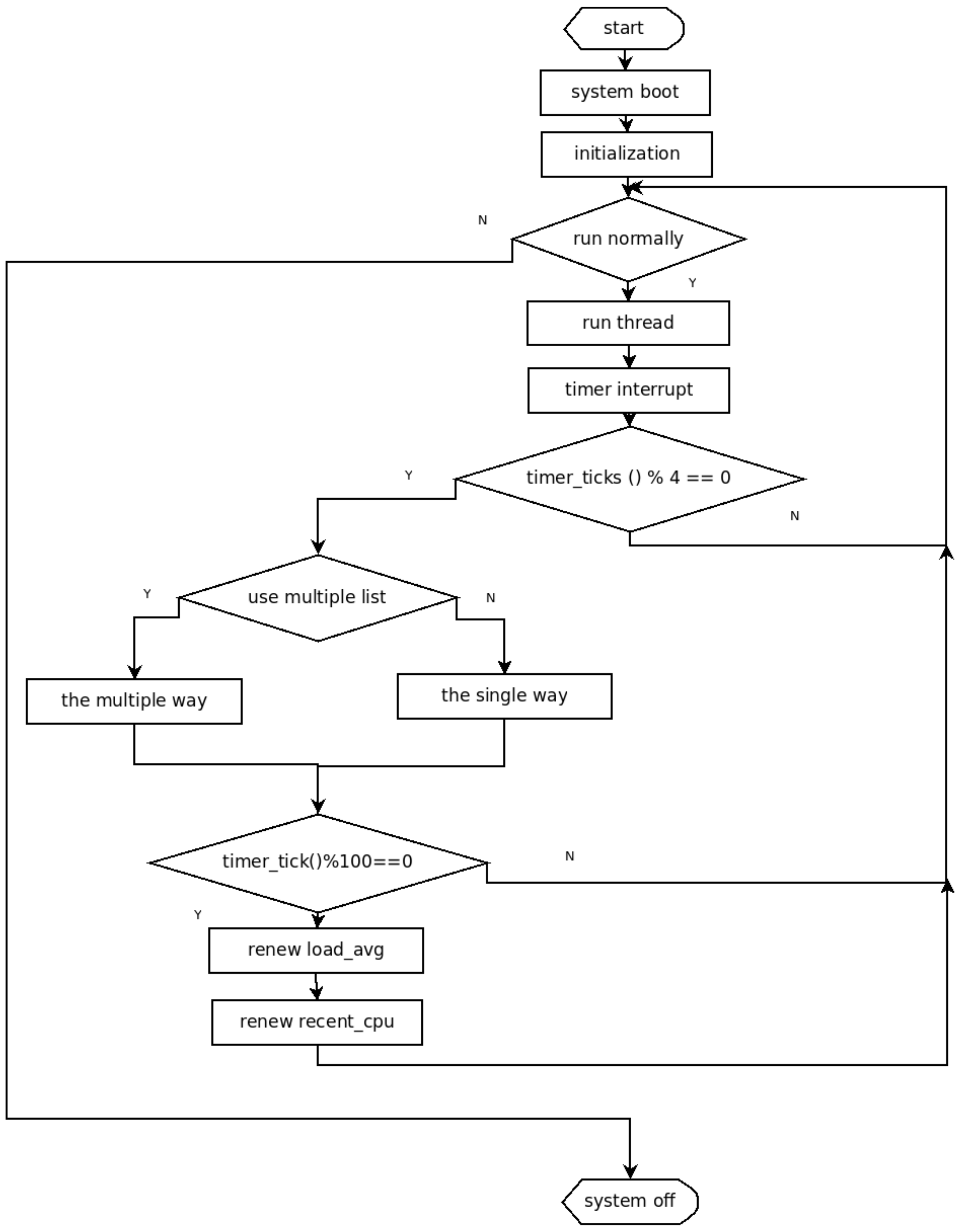


figure 3-2

Specific programming implementation:
1.int64_t get_ready_threads ()
Function: get the current running state and the number of threads ready state.

Method Description: call list_size (& ready_list), get the current number of threads in a ready state, together with a thread currently running and returns the results.
Call between: The renew_load_avg () function call.
Interrupt Status: interrupt should be called off.

2.void renew_load_avg (void)
Function: Update load_avg value.
Method Description: The update function load_avg = (59/60) * load_avg + (1 / 60) * ready_threads,
The meaning of style is to use the original value multiplied by the coefficient and the current value can be retained on the results of a second and updated by the current ready_threads.
Call the relationship: thread_tick () function is called.
Call conditions: timer_ticks ()% TIMER_FREQ == 0
Interrupt Status: interrupt should be called off.

3.void renew_recent_cpu (struct thread * t)
Function: Update thread t's recent_cpu.
Method Description: The update function
recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice
The formula is an iteration, using the exponentially weighted moving average method, iteratively updates ,this can avoid the recent memory cpu running thread table space overhead and query cpu time of the recent cost of running the thread table is an optimization algorithm.
Call between: The thread_all_renew (void) call.
Interrupt Status: interrupt conditions should be called off.

4.thread_all_renew (void)
Function: update all threads' recent_cpu, including running, ready, blocked thread.
Brief Introduction: traversal all_list, searching all the threads, call renew_recent_cpu complete the update.
Call the relationship: thread_tick () called inside.
Call conditions: timer_ticks ()% 4 == 0
Interrupt Status: interrupt conditions should be called off.

5.void renew_priority (struct thread * t)
Function: thread t updates the priority value.
Method Description: The update function
priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
By detecting the recent usage of the thread to change the cpu priority, if the more recent use of cpu, the next stage of the previous stage should be allowed to run with fewer threads cpu, the specific operation that is less recent_cpu multiplied by the coefficient, while the thread itself has a nice value, nice value is higher that the thread is willing to let the cpu to other threads running, the specific approach remains a priority with the nice value to change the value.
Call the relationship: renew_all_priority function is called.
Interrupt Status: interrupt conditions should be called off.

6.void renew_all_priority ()

Function: update all threads priority, including running, ready, blocked thread.
Brief Introduction: traversal all_list, searching all the threads, call priority to complete the update. The next operation, depending on whether use of multi-priority queue will produce different, if you use multiple priority queues, just updated the thread into the corresponding priority queue, if you use a single queue, you need to sort ready _list, in order to ensure the header is the highest priority threads.
Call the relationship: thread_tick () called inside.
Call conditions: timer_ticks ()% 4 == 0
Interrupt Status: interrupt should be called off.

7.thread_tick ()
Function realization: This is a function to add the code corresponds to the chart of the clock the whole part after the break, when the clock interrupt comes, to judge the need for priorities, recent_cpu and load _avg updates.

8.fixed-point.h
Function implementation: fixed-point real number computation macro definitions. Since floating-point type does not implemented in pintos, and load_avg, recent_cpu not int, so we need to use integers to represent real numbers, the approach is a integer int_32's low fourteen bits are supposed after the decimal point by default, due to relatively simple conversion , the use of macros can better accomplish this task, which involves computing a real number between addition and subtraction, multiplication and division, between the real and integer conversion, integer addition and subtraction and multiplication and division between the real number to meet the needs of parameter calculation.

3.3 Debugging process
1.mlfqs-load-1
Target: Test program to test the renew_load_avg () function is operating properly.
Testing process: a direct pass.

2.mlfqs-load-60
Target: Test program to test the renew_load_avg () function is operating properly.
Test: pass, but the results are different with standard output.
Analysis of the reasons: the system starts correctly timing began, when the test thread has not been created, but the main thread is already running, so testing is not threads created load_avg 0.

3.mlfqs-load-avg
Target: testing complex computing environment is correct to renew load_avg
Test: FAIL many times before pass.
Analysis of the reasons: 1. Behind priority calculation is wrong, causing the program did not run according to the original expectations;
2.bochs run in the default condition, and did not run strictly in accordance with exact time, leading to re-open after the break to run the current thread has not been interrupted again and again.
Solution: correct the error, optimize redundant code.

4.mlfqs-fair-2/mlfqs-fair-20
Target: the same priority threads to be distributed to a period of time should be roughly the same time slices;
Test: pass

5.mlfqs-nice-2/mlfqs-nice-10
Target: Test the accuracy of re-calculation of priority.
Test: pass

6.mlfqs-block
Target: recent_cpu blocked thread is correctly updated with the prioriy
Test Results: pass

3.4 Consideration and Optimization

1. Single and multi-priority queue priority queue implementation Comparison of feedback scheduling.
Single-queue and multi-listed implementation differs in the renew step.

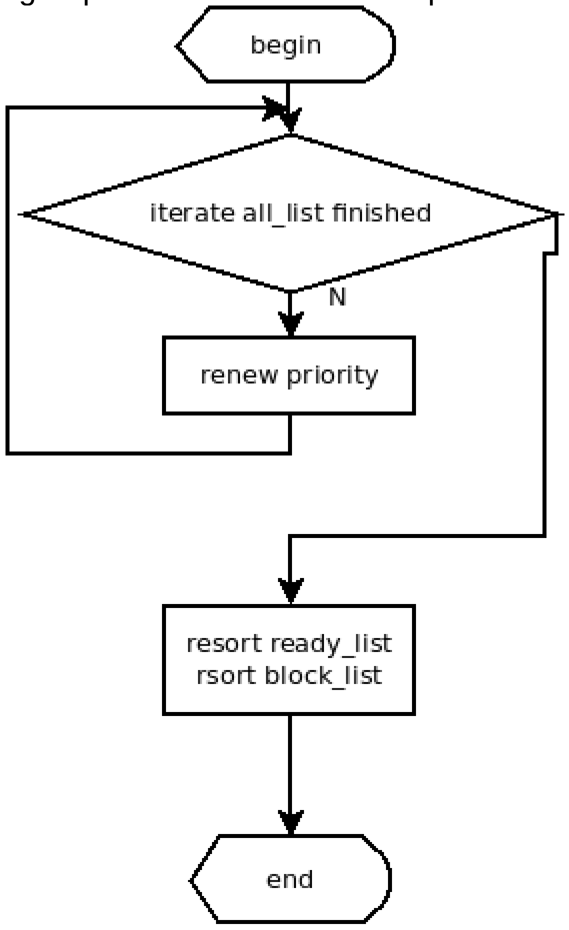


Figure 3-3(one)

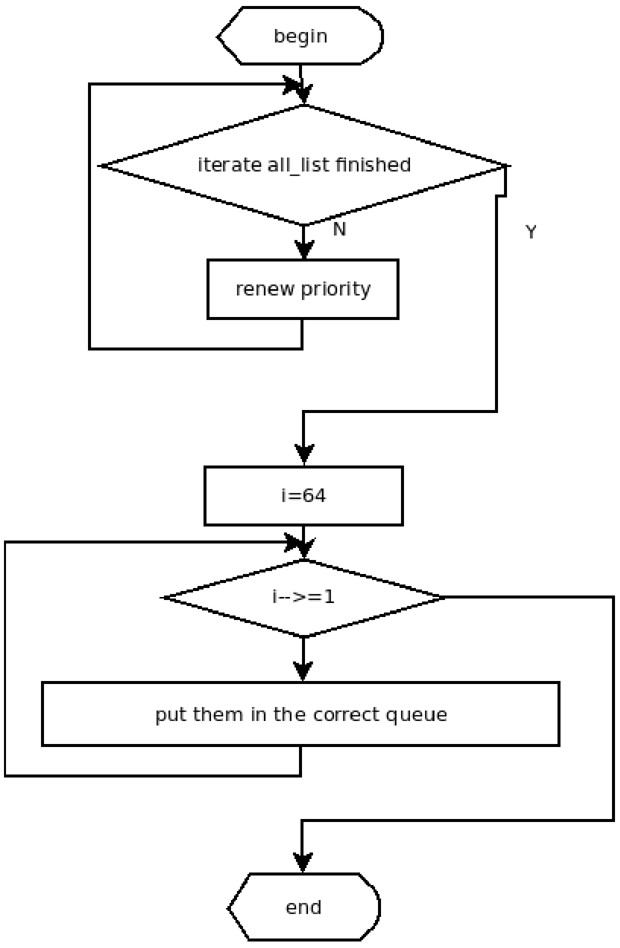


Figure 3-4(multiple)

Note: priority of the updates are carried out by traversing the all_list single queue implementation in the update finished, in order to ensure the queue remain monotonous, to ensure that each visit ready_list and block_list, the exports are the biggest priority thread, the complexity of this step operation is O (nlogn), and multi-queue implementation of all when you need to ready_list thread again, on the surface complexity is O (n), but considering that when the number of threads is less than 64, it remains to deal with 64

times its does not increase the number of average efficiency. In scheduling, we should remove the next running thread, as ready_list single queue sorted by priority, only the pop-up header, complexity is O (1), and more on the line to achieve, need priority 63 finding the queue, find the first queue is not empty out the first element of force, complexity is O (n), and inserted a single queue requires O (n), multiple queues requires O (1).
To sum up, in a small operating system (embedded operating systems, personal computer operating systems), the performance of two implementations should be little difference when running on the server when, ready_list in the thread may reach thousands of orders of magnitude, At this time, the multiple way is optimal.