DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IIT GUWAHATI

CS 341
Operating System Lab

"THREADS- DESIGN DOCUMENTATION"

BY

Group : 38

Kunal Jain 130101042
Mrinal Tak 130101049
Nikhil Teja 130101005

UNDER THE GUIDANCE OF
Prof. G. Sajith

1 ALARM CLOCK:

We are using the DESIGN DOC Questions provided to Stanford Students.

1.1 DATA STRUCTURES

Q.1) Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', or enumeration. Identify the purpose of each in 25 words or less.

— thread.h —

```
struct thread;       /* Added the following fields */
int64_t wake;       /* Absolute time to wakeup thread if thread is asleep */
```

— thread.c —

```
static struct list blocked_list; /* List of processes currently sleeping */
```

1.2 ALGORITHMS

Q.2)Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.

thread_sleep is called with a parameter of the wake up time for the current thread. thread_sleep then disables interrupts, sets the calling thread's wakeup time, inserts the thread into the sleep_list and, finally, blocks.

In schedule(), the sleep_list is traversed and threads eligible to be woken up are removed from the sleep_list and unblocked. Note that the traversal is broken upon finding a thread that is not ready to be awoken, since sleep_list is sorted.

void timer_sleep(int64_t ticks)
In this function, we:
1) Check for valid ticks argument (i.e. ticks >0).
2) Calculate the tick value for the thread as described above. This is calculated by adding the global ticks (ticks since the OS booted) to the ticks argument.
3) Add the current thread to the sleep list. Note that it is added in sorted order such that the front thread element ends its sleep the soonest.
4) Block the thread

Here we called add_blocked(thread_current()); This function add_blocked() adds the current thread to blocked list in sorted order.

static void timer_interrupt(struct intr_frame *args UNUSED)
In this handler, we:
1) Get the beginning front list thread.
2) If the thread's ticks value ¡= the global ticks, the thread is removed from the sleep list and unblocked.
3) Repeat steps 1-2 until the sleep list is empty or the thread's tick value ¿ the global ticks.
4) Test to see if the current thread is still highest priority since other threads may have been unblocked.

Q.3: What steps are taken to minimize the amount of time spent in the timer interrupt handler?

The only timer interrupt handle which has new work is when schedule is called, upon which sleep_list is iterated to potentially wake up sleeping threads. Since sleep_list is ordered, the traversal can exit early as soon as it encounters a thread that isn't ready to be awoken. Thus, we perform only the work that is absolutely necessary in the timer interrupt handler.

By keeping the sleep list in sorted order, this minimizes the time in the interrupt handler. Thus, the handler does not have to iterate through the entire sleep list at every interrupt.

1.3 SYNCHRONIZATION

Q.4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

thread_sleep disables interrupts while manipulating sleep_list, since sleep_list is a kernel structure. The other parts of timer_sleep are not racy (getting number of ticks is already protected by disabled interrupts).

Q.5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?

Again, interrupts are disabled while thread_sleep is manipulating the kernel's sleep_list. This is the only place in which timer interrupts could cause a problem.

1.4 RATIONALE

Q.6: Why did you choose this design? In what ways is it superior to another design you considered?

We considered keeping the sleep list unsorted. While this would make adding sleeping threads more efficient, more time would be spent in the interrupt handler. This could cause a problem when there are a large number of sleeping threads. Thus, we felt the current design is superior based on that evaluation.

First, creating a new list allowed us to keep everything clean as well as efficient. Rather than treating sleeping threads as generic blocked threads, we put them in a separate list so that we can traverse only the threads that are asleep. Furthermore, since we keep the list sorted, minimal work is done inside schedule(), making everything quite fast.

One could imagine implementing the functionality without using an additional list. This, however, would cause the kernel to have to examine more threads than necessary when checking for wakeups. Additionally, it's not quite as intuitive how one would approach this if the list were to be kept sorted (since threads that aren't sleeping would also be in the all_threads list). Therefore, we chose to go with the simple and efficient design of adding an extra list.

2 PRIORITY SCHEDULING

2.1 DATA STRUCTURES

Q.1) Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', or enumeration. Identify the purpose of each in 25 words or less.

— thread.h —

```
int orig_prior;                  /* Base priority, before considering priority donations. */
struct list priority_donations;     /* Sorted list (high-to-low) of all priorities donated to this
                                                                    thread. */
struct lock *waiting;            /* Lock that the thread is waiting to acquire. NULL if the thread is
                                        not waiting on a lock. */
struct list_elem prior;      /* List element for donating a
                                priority to other threads. */
```

— synch.c —

```
struct semaphore_elem
int priority;                      /* Priority of this waiting thread */
```

Q.2: Explain the data structure used to track priority donation.

We chose to embed a sorted linked-list of priority donations into each thread. This means that, if thread A needs to donate a priority to thread B, A will add its priorelem to the priority_donations list of B. That way, B keeps track of ALL donations given to it at any point in time. Additionally, when a thread donates, it calls thread_donate_priority() on the receiving thread (that is, thread A in our example would call thread_donate_priority(B)). This call allows the thread that received the donation to update its effective priority, as well as to update the priority of any outstanding donation that it has made to another thread.

As an example, suppose that thread B is waiting on thread C's lock, in addition to A waiting on B. To make the example more interesting, suppose that yet another thread D is waiting on a *different* lock that thread C also holds.

In this complex case of priority donation, the donation lists would look like this for the four threads:

priority_donations (thread A)
(empty list)

priority_donations (thread B)
A.priorelem

priority_donations (thread C)
B.priorelem
D.priorelem

priority_donations (thread D)
(empty list)
Now, we point out that, when thread A donated priority to thread B, it called thread_donate_priority(B), which would have caused B to recompute its effective priority (taking into consideration the donation from A), *as well as* to update any existing donation that B had made. In this way, the priority from A would be transferred all the way to C *through* thread B's priorelem (which would hold the effective priority of B, taking into account thread A's donation). This update allows chained donation to work correctly.

Finally, we remark that, since we always keep a complete list of donations, thread C will still have the correct priority if, for example, it releases the lock upon which B was waiting, but not the lock upon which D was waiting. This would cause thread C to invoke thread_recall_donation(B), so that B.priorelem would be removed from C.priority_donations, but D.priorelem would remain (which is the correct behavior, since C still holds a lock on which D is waiting). Thus, the effective priority of C will be recomputed, taking into consideration only the base priority of C and the priorelem of D.

In thread_waiting function

We check untill the waiting list on that particular lock is not NULL. When a thread releases a lock, it iterates over the list of threads that were waiting on the lock, and recalls donations on each of the threads, instructing them to each remove their priorelem.

This allows a thread to give up any priority that it received as a result of holding the lock that it just released, while keeping intact any donations that it has received as a result of other locks that it still holds. In other words, clearing the list of priority donations in lock_release () is not sufficient.

```
void thread_waiting(struct thread *t)
{
        enum intr_level old_level;
        old_level = intr_disable ();
        while(t->waiting!=NULL)
        {
                if(t!=thread_current())
                {
                        list_remove(&(t->priorelem));
                        (t->priorelem).next = (t->priorelem).prev = NULL;
                }
                update_prior(t);
                thread_donate_priority(t);
                t = t->waiting->holder;
        }
        intr_set_level( old_level );

}
```

.

In the thread_donate_priority function

We update the priority of the thread in recursive manner by donating the priorelem to the thread. We add it in the list in sorted order

```
void thread_donate_priority(struct thread *t)
{
        list_insert_ordered(&(t->waiting->holder->priority_donations), &t->priorelem , priority_comp , NULL);
        update_prior(t->waiting->holder);
}
```

In update_prior function:

We update the priority of a thread only when the new prior is greater than the original priority. Among the new priorities , we select the one with the max priority and denote it as max_donated_priorty

```
void update_prior(struct thread *t)
{
        enum intr_level old_level;
        old_level = intr_disable ();
        int max_donated_priorty = max_donated_prior(t);
        if(t->orig_prior > max_donated_priorty)
                t->priority = t->orig_prior;
        else
                t->priority = max_donated_priorty ;
        update_ready_list(t);
        intr_set_level (old_level);
        check_priority_yield();
}
```

2.2 ALGORITHMS

Q.3: How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?

For locks and semaphores, we simply modified sema_up to use the list_min function to return the highest-priority thread in the list of waiting threads. Note that we used list_min rather than list_max because our comparison function uses ¿ rather than ¡ (this choice was made so that the threads in the ready list are ordered from highest-priority to lowest-priority). Our rational for using list_min rather than keeping a sorted list was that priorities may change while the thread is waiting on a semaphore/lock (by priority donation), so any order we try to enforce could be voided by donations unless we spend extra effort to re-order the list when donations occur. This seemed more complex than necessary, so we chose to simply pull out the maximal element when we need it, making no assumptions about where it lives in the list.

For condition variables, we added a new field to struct semaphore_elem to store the priority of the thread to which the semaphore_elem belongs. Then, when a thread calls cond_wait, we changed the list insertion to use list_insert_ordered, such that the list will remain sorted from high-priority threads to low-priority threads. Thus, when we want to wake up the highest- priority thread, we need only grab the first element of the list. Note that this works, unlike in the situation described above, because condition variables do not invoke priority donation.

Q.4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?

Suppose thread A is trying to acquire lock L1, which is held by thread B, which is trying to acquire lock L2, which is held by thread C. This is a classic example of nested donation. The call stack during the call to lock_acquire would look something like this:

— Thread A —
lock_acquire (L1)
thread_donate_priority (A)
thread_update_priority (A)
thread_donate_priority (B)
— Thread B —
thread_update_priority (B)
thread_donate_priority (C)
— Thread C —
thread_update_priority (C)

.
When A tries to acquire L1, A recognizes that L1 is held by B. A sets its internal waiting_on pointer to L1, and then calls thread_donate_priority on itself. Thread_donate_priority recomputes A's effective priority (strictly speaking, this is not necessary, but the code reuse that comes with designing the function this way is worth the small amount of extra work). The function then recognizes that A is waiting on L1, looks into L1 and sees that B holds L1, and then makes a donation to B. That is, thread_donate_priority inserts the priorelem of A into the ordered donation list of B. Finally, thread_ donate_priority calls itself recursively on B.

The same sequence of events then happens with B: B updates its effective priority (taking into consideration the new priorelem that it rebharatceived from A), recognizes that it is waiting on L2, which is held by C, then updates the priorelem that it initially gave to C. It does so by calling thread_recall_donation (B). Note that A does not do this because A is the running thread, and, as such, knows that it had not already given a donation. B then re-inserts its priorelem into C's donation list. This step of removing and re-inserting is crucial to preserve the sorted property of donation lists. B then calls thread_donate_priority recursively on C.

Finally, C updates its effective priority, then, recognizing that it is not waiting on a lock, returns and unwinds the stack.

Q.5: Describe the sequence of events when lock_release() is called on a lock that a higher-priority thread is waiting for.

As described in Q.2), a thread invoking lock_release will iterate over the the lock's waiter list and, for each thread in the list, invoke thread_recall_donation on the thread. This function will, in turn, cause the given thread to remove its priorelem from the releasing thread's list of priority donations. Performing this on each element of the wait list causes the releasing thread to lose all priority donations associate with the lock that it is releasing (as it should).

2.3 SYNCHRONIZATION

Q.6: Describe a potential race in thread_set_priority() and explain how your implementation avoids it. Can you use a lock to avoid this race?

thread_set_priority must determine whether or not to yield, which requires determining the current highest-priority thread in the ready list. Doing so requires accessing the ready list, which is a shared structure. If we were to access the list without synchronization, it is possible that interleaving could occur in a detrimental way. We have detailed this possible race in the function max_priority below:

```
void update_prior(struct thread *t)
{
        enum intr_level old_level;
        old_level = intr_disable ();
        int max_donated_priorty = max_donated_prior(t);
        if(t->orig_prior > max_donated_priorty)
                t->priority = t->orig_prior;
        else
                t->priority = max_donated_priorty ;
        update_ready_list(t);
        intr_set_level (old_level);
        check_priority_yield();
}
```

Our solution to avoiding the race was, as stated in the comments, to disable interrupts during the call to thread_max_priority, which determines the maxmimum thread priority. It would *not* have been a good idea to synchronize this access using locks - that would require that all access to the ready list be protected by a lock rather than disabled interrupts. We prefer to follow the convention already established and use disabled interrupts for synchronizing access to kernel structures, rather than uprooting much of the existing code to make the lock synchronization work.

Q.7: Why did you choose this design? In what ways is it superior to another design you considered?

Our design is both simple and powerful. The use of a donation list allows all threads to keep track of, in theory, an infinite number of donations from other threads. Furthermore, our method can (again, in theory) handle an infinite-depth donation chain. Of course, we are limited by stack space, since thread_donate_priority is called recursively. The power in our design comes from the simple observation that any given thread may only ever be waiting on one thread at a time. This observation allowed us to justify embedding a priorelem in the thread structure, which allows the list donation scheme to work.

We consider the memory requirements for our design, assuming 32-bit arch:
1 x struct list = 16 bytes
1 x struct list_elem = 8 bytes
1 x struct lock* = 4 bytes
= 28 bytes total

Thus, we use 28 bytes per thread for our priority donation scheme. In terms of running time, we require O(1) time for determining the effective priority of any given thread (since the list of donations is ordered), O(n) time for donating a priority, where n is the number of existing donations for the receiving thread, and O(1) time for recalling a donation. In all, the time requirements of this method are *very* cheap! Note that the most common operation, determining a thread's effective priority, is very fast.

Now, we might have considered storing a fixed-size array of donations as an alternative. That is, we could have defined struct thread* donations[8], for example, and used the array to store donations on a per-thread basis. Such a scheme would be inferior to our design in every way. First, it would require 32 bytes per thread minimum (and probably 36, since it is still convenient to have a lock* to the waiting lock), and that's just if we wanted to support a maximum of 8 donations! Second, it limits the total number of donations allowable for any single thread. Finally, although the running time could be made comparable to that of our design, it would require that we write procedures for manipulating a sorted, array-based list. Our design leverages the power of the existing list code to implement priority donation in a way that is clean, fast, simple, and powerful.

3 PRIORITY SCHEDULING

3.1 DATA STRUCTURES

Q.1: Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', or enumeration. Identify the purpose of each in 25 words or less.

— thread.h —

```
struct thread
int recent_cpu;             /* Estimation of clock ticks recently
used by this thread */
int nice;                   /* Niceness value for BSD scheduling */
```

— thread.c —

```
int load_avg;                 /* load average for BSD scheduling */
```

3.2 ALGORITHMS

Q.2: 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 priority and recent_cpu values for each thread after each given number of timer ticks:

timer ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority C	thread to run
00	00	0	0	63	61	59	A
04	04	0	0	62	61	59	A
08	08	0	0	61	61	59	B
12	08	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

Q.3: Did any ambiguities in the scheduler specification make values in the table uncertain? If so, what rule did you use to resolve them? Does this match the behavior of your scheduler?

Ambiguities in the scheduler specifications made some timer ticks have multiple threads share the same highest priority, making the thread to run technically ambiguous. To resolve this, the thread with the highest priority that has not run in the most number of ticks is selected as the next thread to run. This behavior matches with the behavior of our scheduler, since our scheduler inserts the current thread back into the ready list in an ordered way such that it is moved to the back of any elements with the same priority. This produces a fair, round-robin policy so that no thread will dominate the system even when multiple threads maintain the same priority.

Q.4: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

The new additional steps of updating recent_cpu and load_avg values for BSD scheduling are done during a thread_tick interrupt. Of course, we only touch all of the threads every second, so very little overhead has been added to the timer interrupt overall. The most expensive part of the whole operation is recomputing the thread priorities in order to select a new one. However, we only do so during the call to schedule (as opposed to every tick), and we do so by using the built-in list sort routine. The sort is fast, and, moreover, may take advantage of the fact that the list is already partially-ordered.

Overall, we are doing no more work than necessary, so we expect performance to be good. We note that, initially, we implemented a scheme whereby threads were re-inserted into the list immediately upon incurring a priority update, preserving the ordering of the list on an element-by-element basis. This method, however, was wasteful and caused some of our tests to fail due to missed ticks. This is when we streamlined the process by deferring the sorting of the ready list until after all priorities had been updated, which turned out to be far more efficient.

3.3 RATIONALE

Q.5: Briefly critique your design, pointing out advantages and disadvantages in your design choices. If you were to have extra time to work on this part of the project, how might you choose to refine or improve your design?

We think that we've come up with a very clean design which leverages lists and minimum additional data elements and structures. We explicitly avoided potentially sparse arrays and indices for features such as priority donation and thus do not limit ourselves to a preset depth (within memory bounds) of donation. Additionally, our lists allow us to leverage the existing list structure which is already used in Pintos and to simplify many operations such as ordered insertion, mapping functions across many elements, sorting, and removal.

Although we are quite pleased with our design, if we had more time, we would like to figure out how to profile Pintos. Profiling would be the only real way to gain performance (or at least we think...hopefully we've grabbed the low-hanging fruit?). We'd enjoy figuring out if it would be possible to cut down on time spent in interrupt handlers (in fact, we'd be interested in just knowing what that number looks like right now).
Q.6: The assignment explains arithmetic for fixed-point math in detail, but it leaves it open to you to implement it. Why did you decide to implement it the way you did? If you created an abstraction layer for fixed-point math, that is, an abstract data type and/or a set of functions or macros to manipulate fixed-point numbers, why did you do so? If not, why not?

We implemented fixed-point with a typedef and a series of manipulate fixed-point numbers (fixed_point_create, fixed_point_multiply, etc.). We did this to make it clear to users or readers when they were using fixed_point values or operations, since it requires explicitly writing out the functions and type. This seems safer than re-writing the formulas each time. Additionally, it also creates an easy avenue to experiment with different fixed point splits such as using 15 vs 14 bits for the fractional bits and observing if there are any benefits in performance.

We chose to abstract the functions because it seems very, very easy to make a silly mistake in the math, which might end up causing miscalculations in priorities, which would make for a very hard-to-track bug (if, for example, a ¿¿ was interchanged with a ¡¡). As such, we decided to write the functions, then perform a series of unit tests on them using printf in Pintos to verify that they were working as we expected them to. This spared us the pain of debugging once the functions were put to real use.