Jaswinder Kaur

ALARM CLOCK
===========

---- DATA STRUCTURES ----

>> A1: 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.

Static struct list sleeping_threads_list;   /* List of all currently sleeping threads and shows the
-> tick value where the thread is finished sleeping */

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.

Timer_sleep() suspends execution of the calling thread until time has advanced by at least x timer ticks. The argument in timer_sleep() is given in timer ticks. Those timer values are expressed as signed 64-bit number, timer values does 100 ticks per second. The system timer by default has set up to 100 ticks per seconds. Timer_sleep() function uses the while loop which calls to thread_yield() to wait for the specified number of ticks in timer_sleep(). Call to thread_yield() is the reason of the busy waiting in the timer.c because thread_yield() immediately restores the interrupt level upon return from scheduler() even though thread is sleeping and does not wake up yet. Also, interrupts are turned on inside the timer_sleep(). Which basically means that timer is set up to interrupt which would be calling for each elapsed tick.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?

We ran the test “pintos run alarm-priority” which shows we have bug inside priority scheduler, Threads are immediately being schedule even if it is not awake yet. The sleeping threads are using a lot CPU time which is wasteful of CPU time. Since time interrupt handler determines how long the current thread has been running on the CPU we need to take steps that make sure that timer interrupt handler wake up the sleeping threads on the appropriate time. So instead of let thread_yield() to immediately schedule the sleeping thread we will need to remove thread_yield() from our timer_sleep() function . we should actually let the thread blocked until the specified number of ticks are counted.Threads with earliest wake up needs to be unblock first. Also,keep track of all the remaining sleeping threads in order to find the next wake up time thread.since all the blocking/sleeping threads are not executing any cpu time thus they free up space for wake up threads.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?

Race conditions are avoided when multiple threads call timer_sleep() simultaneously due to interrupts being turned off which will not allow threads to be put to sleep at the same time.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?

Race conditions are avoided when a timer interrupt occurs during a call to timer_sleep() by turning off interrupts in the critical section of timer_sleep(). By doing so, timer interrupts will not occur during the call to timer_sleep resulting in no race conditions.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?

For data structures the linked list handels the frequent and random insertions very well compared to other data structures. Arrays need to be rebuilt with any element being inserted/ deleted in most places. Trees would be a good contender for insertion, but degrade the iteration runtime which is very important, as it runs at every timer interrupt. Hashmaps just didn’t make sense with the ordered nature required and lack of random access being required. Timer_interupt was used to wake the threads after the timer was up as it was the only solution found and it works with no glaring issues. The ordered list enables us to have to check/ iterate over only one extra element that does not need to be woken, to see if any need to be removed. So efficacy shouldn’t be to bad so long as we know which end of the list has the smaller values.

--- CODE --

```
typedef struct thread_store{
   struct *thread thread_pointer;
   int64_t tick_when_done;
}
```

Thread_store is a data type that holds a pointer to a sleeping thread, and the tick number when that thread should be woken. The sleep list will consist of these.

1) Insert

2) Disabling interrupts

3) Deleting

Code outline:

1. Timer_sleep():
   -Block interrupts and save priority (copied from other methods in the thread.c file)
      This: “old_level = intr_disable ();”
   -Take current time
   -Add to sleep list
      -This is our custom function
   -Call thread_block. (This will call the scheduler automatically and start executing other threads)
   -Restore priority level so the thread isn’t kneecapped for the rest of its life
      This:”intr_set_level (old_level);”

Deleting: Insert a function call after the call to thread_tick() in timer_interupt()

a. Function should iterate over the sleep list struct from smallest to larest to find any threads that need to be woken up.
b. If a thread that needs to be woken is found: remove element from list, and call thread_unblock(). That should be it.
c. Potential complications. Thread priorities might not get saved. If they don’t we will just throw them in another tuple in the sleep list and restore when we wake them.

Misc: initialize empty list in timer_init

- list_init(*sleeping_threads_list)

Insert list definition