CSE521 PROJECT 1

Bhavya Dayananda Kattapuni UBIT 50111831
bhavyada@buffalo.edu

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.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
    int64_t sleep_until;                                /* for project 1 */
    int orig_prio;                      /* Original Priority */
    struct lockPrio lock_prio_struct[MAX_LOCK_NEST];  /* record priority donation with limit 8 */
    struct lock * lock_blocked;                 /* This thread is blocked on this lock */
    int recent_cpu;                                          /* recent cpu calculater for this thread- fixed
point */
    int nice;                                                /* nice value for thread- fixed point */
  };
```

In Timer.c

Added a list

static struct list timer_sleeping_threads;
-- list of all the threads which are sleeping

In thread.h

This variable is added to struct thread.

int64_t sleep_until;
--stores the time until which the thread should sleep.

---- ALGORITHMS ----

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

In timer_sleep
1) Initially the interrupt is disabled to avoid race condition.
2) Time upto which the thread should sleep is calculated i.e the initial time plus the total time.
3)The thread is put into the list of sleeping threads with the time to wke up assigned to it.
4) The thread is inserted in the sorted order such that the thread which has the earliest time to be woken is at the beginning of the list.
5) Block the thread.
6) Again set the interrupt level to the previous level.

In timer interrupt, the 'sleep_until' value of first thread in sleeping list is compared to the current time and woken it up if it is ready. This is repeated if there are multiple threads that have to be woken at same time.

A3. What steps are taken to minimize the amount of time spent in the timer interrupt handler?
The function less_fxn_sleeping_time has been defined. This is used from timer_sleep, while inserting into timer_sleeping_threads, to keep it as a sorted list of sleeping threads according to the ascending order of their wake up time. Due to this, the handler need not iterate through the entire sleep list at every interrupt, and simply pop the first thread.

---- SYNCHRONISATION ----

A4. How are race conditions avoided when multiple threads call timer_sleep() simultaneously?
Interrupts are disabled to prevent race conditions.

A5. How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep?
In timer_sleep the interrupts are turned off using intr_disable(). The sleeping list and the ready list are modified or read by the interrupt handler. So it is important that the interrupts are disabled at this time to prevent any unstable condition when the timer interrupt occurs.

---- RATIONALE ----

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

The design uses the existing list structure and is easy to handle. The list of ready threads is sorted such that the scheduler can easily pop out the first thread to run according to the condition instead of searching through the entire list.

PRIORITY SCHEDULING

---- DATA STRUCTURES ----

B1: 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.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
```

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
    int64_t sleep_until;                                /* for project 1 */
    int orig_prio;                      /* Original Priority */
    struct lockPrio lock_prio_struct[MAX_LOCK_NEST];  /* record priority donation with limit 8 */
    struct lock * lock_blocked;                 /* This thread is blocked on this lock */
    int recent_cpu;                                          /* recent cpu calculater for this thread- fixed
point */
    int nice;                                                /* nice value for thread- fixed point */
  };

struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
    int id;
  };
```

Added these members to the thread structure

int orig_prio;
--This is declared to keep track of the original priority

struct lock_prio_struct[MAX_LOCK_NEST];
-- this is used to record nested priority donation with limit 8.Used to indicate the maximum locks that a single thread can hold at any given time.

*struct lock * lock_blocked;*
-- this is a pointer to the thread which is blocked on this lock. This is necessary for nested donation.

synch.c

Added this data structure when initialising lock.

static int LockID = 0;
--start lockID numbering from 0. For finding locks in lock_prio_struct of threads

B2: Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, submit a .png file.)
This uses:

```
struct lockPrio
{
        int lockID;     //unique id of a lock
        int prio;       //highest priority donation obtained for that lock
};
```

and array :
struct lock_prio lock_prio_struct[MAX_LOCK_NEST];

X,Y,Z are locks
A,B,C, D are threads having priority in ascending order

Diagram to show nested donation.

```
   A --> X
 B-->A-->X
C-->B-->A-->X
Now C has donated its priority to A to block lock X.
```

Suppose A is holding both X and Y and B is holding Z. Another thread D of higher priority D wants Z.

```
 A --> X AND Y
 B --> Z
D-->B--> Z
Since D wants to acquire Z which is held by B, where B is waiting on A, so D will donate its prior-
ity to A.
```

---- ALGORITHMS ----

B3: How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?

Created a function which would return the thread having the greater priority.

```
bool greater_prio_fxn (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
   struct thread *ta = list_entry (a, struct thread, elem);
   struct thread *tb = list_entry (b, struct thread, elem);

   return (ta->priority > tb->priority);
}
```

Once the higher priority thread is returned, it is used to sort the list in the descending order of their priorites using this function.

list_insert_ordered (&ready_list, &t->elem, greater_prio_fxn, NULL);
and list_sort()
So when the thread is unblocked the first thread in the list would be the thread having the highest priority.

B4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?

1) Every time a thread (thread A) acquires a lock (say lock X)one of the MAX_LOCK_NEST elements of its lock_prio_struct is initialized to the lock id of the lock.
2) If a different thread (thread B) tries to acquire lock X and its prio is higher than thread A, the element of lock_prio_struct of thread A corresponding to X is updated with prio of thread B and B donates priority to A.
3) Now if thread C with even higher prio tries to acquire lock X, the element of lock_prio_struct of thread A corresponding to X is again updated to prio of C and C donates to A.
4) Let's say thread A also acquires another lock Y and thread B had acquired lock Z before being blocked on X. Another thread D with higher prio that A and B now tries to acquire Z. D donates to B and B should donate to A. D uses `lock_blocked->holder` to find B and then again A to complete the donate chain.

So once we have donated the priority we keep track of that lock holder thread by using the pointer.
thread_current()->lock_blocked = lock;

and if we follow it until we hit the pointer = NULL.
thread_current()->lock_blocked = NULL;

This way we can track all the threads that got a donation in that chain and change their priority if necessary.

B5: Describe the sequence of events when lock_release() is called on a lock that a higher-priority thread is waiting for.

1) The current thread looks through its lock_prio_struct array to find the lock id element it is releasing and resets that element.
2) It also find the next highest donated priority in the same array (if none, it gets its original priority) and sets it's own priority to that number.

3) The rest is the same as before- set lock->holder to NULL, up the semaphore and yield (since new priority could be lower than one of the ready threads)

---- SYNCHRONIZATION ----

B6: Describe a potential race in thread_set_priority() and explain how your implementation avoids it. Can you use a lock to avoid this race?
Locks can not be used in interrupts, since when interrupt is running, other interrupts are disabled and if the lock is already taken, blocking the interrupt, the system will never recover.
To avoid this, interrupts should be disabled in this critical section. This way, although a context switch can not happen, the current execution can not be blocked either.

---- RATIONALE ----

B7: Why did you choose this design? In what ways is it superior to another design you considered?
Since each entity (thread or lock) contained its own regulating structures, it was elegant and modularized. By maintaining sorted lists, the mechanism was made efficient. Also this method is logically simple and robust.

ADVANCED SCHEDULER

---- DATA STRUCTURES ----

C1: 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.

In thread.c

#DEFINE Q_(14)
#DEFINE F_(2<<Q)
Defined two constants Q and F which is necessary for conversion of floating to integer and vice versa.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
    int64_t sleep_until;                                /* for project 1 */
    int orig_prio;                      /* Original Priority */
    struct lockPrio lock_prio_struct[MAX_LOCK_NEST];  /* record priority donation with limit 8 */
    struct lock * lock_blocked;                 /* This thread is blocked on this lock */
    int recent_cpu;                                         /* recent cpu calculated for this thread- fixed
point */
    int nice;                                               /* nice value for thread- fixed point */
  };
```

These two data structures are added to the struct thread.

int recent_cpu;
- it holds the rcent cpu value

int nice;
- it holds the nice value

Declared a global variable
static int load_avg =0;
-a global variable which holds the systems load average. Initilised to zero.

---- ALGORITHMS ----

C2. 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:

Number of timer interrupts per second is:
#define TIMER_FREQ 100
This means the load average and recent cpu never get updated with the formula
Only the running thread receives +1 recent_cpu for every tick.

timer ticks	recent_cpu A	B	C	priority A	B	C	thread to run
0	0	1	2	63	60	58	A
4	4	1	2	62	60	58	A
8	8	1	2	61	60	58	A
12	12	1	2	60	60	58	B
16	12	5	2	60	59	58	A
20	16	5	2	59	59	58	B
24	16	9	2	59	58	58	A
28	20	9	2	58	58	58	C
32	20	9	6	58	58	57	B
36	20	13	6	58	57	57	A

Solving this
Intially we consider the load_avg =0

load_avg = (59/60)*recent_cpu + (1/60)*num_of_threads
load_avg =(59/60)*0 +(1/60)*3
load_avg = 1/20 = 0.05

Depending on the load_avg and recent_cpu values we calculate the priority for A, B and C.
Priority for thread = PRI_MAX - recent cpu/4 - 2*nice
Priority for A = 63 - (0/4) - (2*0) = 63
Priority for B = 63 - (4/4) - (2*1) = 60
Priority for C = 63 - (4/4) -(2*2) = 58

Since A has the highest priority A is the running thread. So after 4 timer ticks the recent_cpu value of A becomes 4. And similarly we calculate the rest of the values.

C3: 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?

From the above table we see that for ticks 12, 20, 28,32 we have two or more threads which have the same priority. So there was an ambiguity on which thread should be selected to run. Here we could solve it using the round robin algorithm. The thread gets pushed at the end of its corresponding queue and the next thread which is at the front is selected. A time quantum is given and all the threads get an access to the processor.

In the scheduler I have developed, I do not have multiple queues, instead I have a single queue which selects the threads having the highest priority. It is given that the thread which is running would be incresing its recent_cpu for every tick and hence the priority of the running thread gets reduced as we subtrct the recent_cpu from PRI_MAX. So the first ready thread in the queue gets access to the processor. And for every 4 ticks the priority changes and the next best thread is selected to run.

C4: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

I wrote most of the code inside the thread_tick funcion as the recent_cpu value and the priority would be calculated depending on the timer ticks. The function thread_ticks is called by the timer interrupt handler at each timer tick. Thus, this function runs in the interrupt context since it is called by the timer_interrupt function.
The calculations for recent cpu and load average could be moved out of the interrupt and called from a thread's context. Every context switch would happen if timer ticks worth more than 1 second have elapsed (timer ticks since last call will have to be kept track of). However, this will cause the load of the thread that runs these function to be more and it's recent cpu increases and hence its effective priority decreases.

---- RATIONALE ----

C5: 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?

The advantages of my design is that since I managed to create a single queue the implementation was much more easier considering the complexities of pointers and time quantum management which would be required for a multilevel queue.
The disadvantage in this method is that the other threads which are behind the first thread in the ready queue, even though having the same priority, will not get a chance to access the processor and this would lead to starvation.

Given more time I would have implemented the multilevel queue such that the round robin algorithm would be applied to give equal opportunity to all threads having the same priority.

C6: 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?

For this I did the math in the code itself, since it was used only 6 times. I defined Q_ and F_ initially and used it for conversion. Also some optimisation could be done in this method i.e. if we have to do multiplication followed by division, the floating point changes could be cancelled out and we could treat it like a simple arithmetic instead of fixed point arithmetic.

SURVEY QUESTIONS
==========

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

>> Any other comments?