+--+
                                                                Design Document for
Project 1: Phase 01

+--+

+---------- GROUPS ------------+

* Anirudh Nallana <anallana@buffalo.edu>
* Priyanka Garg <pgarg2@buffalo.edu>
* Aishwarya Chand <achand3@buffalo.edu>

Task 1: Efficient Alarm Clock

+------- DATA STRUCTURES -------+

New Data Structures:

In timer.h

```
/* Phase 1 Changes */
/* Timer sleeper */
struct timer_sleeper
  {
    struct semaphore timer_sema; /* Counting semaphore for timer */
    int amount_ticks;            /* To save the info of start(start timer
tick)+ticks(sleep time of alarm clock) of interrupter */
    struct list_elem elem;       /* List elem to use with a queue list */
  };

static struct list waiting_queue; /* Waiting Queue for Timer sleeper */
static struct lock queue_lock;    /* Lock Synchronization for waiting_queue */
```

In timer.c

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

  /* Phase 1 Changes :- */
```

```
  struct timer_sleeper cur;
  sema_init (&cur.timer_sema, 0);
  cur.amount_ticks = start+ticks;

  /* Synchronization for `waiting_queue` using `queue_lock` */
  lock_acquire (&queue_lock);
  list_push_back (&waiting_queue, &cur.elem);
  lock_release (&queue_lock);

  sema_down (&cur.timer_sema);
}


/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;

  /* Phase 1 Changes :- */
  /* Iterate through the `waiting_queue` to wake up threads using `sema_up()` */
  struct timer_sleeper* cur;
  struct list_elem* next = list_begin (&waiting_queue);
  while (next != list_end (&waiting_queue))
  {
    cur = list_entry (next, struct timer_sleeper, elem);
    if (cur->amount_ticks <= ticks)
      {
        sema_up (&cur->timer_sema);
        next = list_remove (next);  /* Remove list_elem after waking up thread */
      }
    else
        next = list_next (next);
  }

  thread_tick ();
}
```

+------- Algorithm --------+

---A2---

* timer_sleep():

 >> Instead of busy waiting, the current thread blocks on a semaphore timer_sema.

 >> First, we calculate the wake-up time by adding the current timer_ticks() to the ticks value provided an this calculated wake-up time is stored in the amount_ticks field of a newly created timer_sleeper structure.

>> A timer_sleeper structure is created, storing the thread's wake-up time and a semaphore to block the thread.The semaphore is also initialized within this structure.

>> The thread is added to the waiting_queue and then blocks itself by calling sema_down().

>> The timer interrupt handler periodically checks the waiting_queue and wakes up threads whose sleep time has expired by calling sema_up() and removing them from the queue.

---A3:---

>> The timer interrupt handler iterates through the waiting_queue and only performs minimal operationslike checking if the current time (ticks) has surpassed the thread’s wake-up time.
>> Threads that are ready to wake up are immediately removed from the waiting_queue using list_remove().

+---- Synchronization ------+

>> In our code in order to ensure safe access to shared resources like the sleeping_list, we need to enforce synchronization because the sleep_list is accessed by multiple threads concurrently.

>> Since Pintos' lists are not inherently thread-safe, we use a lock, sleepling_list_lock, to manage access.

>> When inserting elements from the sleeping_list the lock is acquired to prevent race conditions between multiple threads.

>> After the operation, the lock is released, allowing other threads to access the list.

>> Additionally, each sleeping thread blocks on a unique semaphore (sema_down()) until it is time to wake up, ensuring that no unnecessary CPU cycles are wasted by busy waiting.

+---------- Rationale ---------+
>> The advantage of our implementation is that it is simple and effective.

>> A disadvantage we can consider may occur for large test cases with multiple number of threads, which may cause unnecessarily iterating the entire waiting_queue structure.

>> This can be mitigated if the waiting_queue is sorted, but we chose to not implement this to avoid unwanted issues such as causing irregularities in timer ticks intervals.

>> Additionally, the time complexity is O(n)(where n is the length of the waiting_queue), which didn't call for this optimization.

+--
--+

Task 2: PRIORITY SCHEDULING

>>We plan to implement a priority scheduling algorithm inside next_thread_to_run(), and the priority donation by modifying the thread struct to add additional priority which holds the donation priority but are yet to deliberate and decide on it.

---- DATA STRUCTURES ----

>> We plan to add an additional property inside the struct thread which holds donation priority value. This value can then be utilized to perform priority scheduling with preemption.

---- ALGORITHMS ----

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

>>To make sure the highest-priority thread wakes up first when waiting for a lock, semaphore, or condition variable, priority donation is used. In this process, if a high-priority thread is waiting for a lock held by another thread, the waiting thread temporarily donates its priority to the thread holding the lock. This boosts the priority of the second thread so it can release the lock faster. Once the lock is released, the second thread’s priority returns to normal, and the high-priority thread continues executing.

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

>> When lock_acquire() is called, a thread attempts to get a lock that is already held by another thread. If the waiting thread has a higher priority than the one holding the lock, it donates its priority to the lock-holding thread. This boosts the priority of the other thread, allowing it to release the lock more quickly so the higher-priority thread can proceed. In the case of nested donation, if the second thread is also waiting on a lock held by a third thread, the first thread’s priority is passed along to the third thread, ensuring that the third thread runs

faster with the highest priority.

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

>> When lock_release() is called, the thread gives up the lock, and its priority goes back to its original, lower value. After the lock is released, if no other locks are in the way, the higher-priority thread waiting for the lock can start running.

---- 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?
>> thread_set_priority() is a critical method to change a thread's priority. We have not yet planned to implement any synchronization here, but instead thought of implementing changing of priorities of threads inside the external interrupt context :- thread_tick()

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?

>> We came up with 2 approaches but insertion and removal of highest priority thread is more efficient in this approach.

+---
---+

Task 3: ADVANCED SCHEDULER

>>At this point, we just had discussions on implementing the MLFQ Scheduler.
>>We plan to create a new header file for this new scheduler and implement a flag for it in execution.
>> We plan to utlize FCFS scheduling(using struct list list_push_back and list_pop_front) for the levels inside MLFQS

---- DATA STRUCTURES ----

>> A list of lists for MLF queue,
>> Possibly change the struct thread again to add additional properties to handle priorities effectively
>> A boolen flag for mlfqs scheduling

---- 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:

timer ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority 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	A
12	12	0	0	60	61	59	B
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	A
28	20	8	0	58	59	59	C
32	20	8	4	58	59	58	B
36	20	12	4	58	58	58	B

>> 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?

>>The following ambiguities can be faced :
>> Deciding how often should recent_cpu and priority should be updated .
>> There could also be ambiquities on how do ties get broken between threads with the same priority.

>> We plan to resolve these by -
>> For example if two threads have the same priority we can use round-robin scheduling(but we are planning to stick to FCFS for now and optimize scheduling then on).

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

>>recent_cpu and priority calculations are done iside timer interrupt and these calculation are triggered every few timer ticks .
>> Instead of updating the priority for each of all the threads , we plan that the sytem updates only priority of the current thread every 4 ticks.
>>So in this case ,all threads will get their priority updated every 1 second
>> By only updating the current thread's priority every 4 ticks all threads priorities , less time is spent inside the timer interrupt.
>> This improves systems performance by reducing work during each interrupt.

---- RATIONALE ----

---- 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?

Our plan to implement a MLFQ Scheduler is pretty straightforward with multiple queues/levels
with the same priority(which makes insertion and deletion less tricky, since all priorities in
a level are equal)

An idea of improvment is to not necessarily check all the processes every few ticks and update priorities, but to only update priority when a process is running.

>> 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?

We have not discussed introducing an abstraction layer as of yet and plan to follow the same arithematic taught in the lectures. But otherwise, an abstraction layer would definitely
prove to be a better and flexible choice