+--+

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

---- DATA STRUCTURES ----

>> Modified `struct thread` :- added a donation priority(`int donation_priority`) to save donated priorities, added a list to track all locks(`struct list locks`) acquired by thread(to handle multiple locks test case), also added a lock object(`struct lock* locker`) to save the lock which is blocking the thread(and causing priority inversion/deadblock).

```
        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. */
                int donation_priority;              /* Donated Priority implemented
to avoid priority inversion */
                struct list_elem allelem;           /* List element for all threads
list. */

                /* Shared between thread.c and synch.c. */
                struct list_elem elem;              /* List element. */
                struct list locks;                  /* List for tracking all locks
*/
                struct lock* locker;                /* Lock blocking thread */

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

```
                /* Owned by thread.c. */
                unsigned magic;                     /* Detects stack overflow. */
          };
```

>> Modified `struct lock` :- added an integer variable(`int max_prio`) to save the highest priority thread which (is going to)uses this lock, added a list element(`struct list_elem elem`) (which was adopted from semaphore->waiters) in order to save lock to a list inside `struct thread`.

```
        /* Lock. */
        struct lock
          {
                int max_prio;               /* To save the highest priority
currently meddling with this lock. */

                struct thread *holder;      /* Thread holding lock (for debugging).
*/
                struct semaphore semaphore; /* Binary semaphore controlling access.
*/
                struct list_elem elem;      /* List elem to save lock onto thread
which has acquired it. */
          };
```

---- ALGORITHMS ----

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

>> We implemented this by choosing the highest priority thread(`list_max ()` from list.h) from list of threads with a compare function that compares priorities/donation priorities between two threads. The compare function code is as follows :-

```
        /* Compare func to get the highest priority thread*/
        bool
        get_highprio_thread (const struct list_elem* thread_1_elem, const struct
list_elem* thread_2_elem, void * aux UNUSED)
        {
                struct thread* thread_1 = list_entry (thread_1_elem, struct thread,
elem);
                struct thread* thread_2 = list_entry (thread_2_elem, struct thread,
elem);
                int thread_1_max_prio = thread_ask_priority (thread_1);
                int thread_2_max_prio = thread_ask_priority (thread_2);

                return thread_1_max_prio < thread_2_max_prio;
        }
```

>> Below is an implementation where we utilised `list_max()` and the compare function to get the highest priority thread and remove it(or wake up) for function

`sema_up()` :-

```
    struct list_elem* highprio_thread_elem = list_max (&sema->waiters,
get_highprio_thread, NULL);
    struct thread* highprio_thread = list_entry (highprio_thread_elem, struct
thread, elem);
    list_remove (highprio_thread_elem);
```

>> The function to retrieve the priority of a thread is also changed, since the addition of new donation priority attribute. We also implemented a new (overloaded) function to retrieve the priority of the thread passed as argument :-

```
        /* Returns the current thread's priority(or donation priority whichever is
greater). */
        int
        thread_get_priority (void)
        {
        struct thread* curr_thread = thread_current();
        int thread_priority = (curr_thread->donation_priority >
curr_thread->priority) ?
                                    curr_thread->donation_priority :
curr_thread->priority;
        return thread_priority;
        }

        /* Returns the given thread's priority(or donation priority whichever is
greater). */
        int
        thread_ask_priority (struct thread* t)
        {
          int thread_priority = (t->donation_priority > t->priority) ?
                                                            t->donation_priority :
t->priority;
          return thread_priority;
        }
```

>> For locks, no additional sorting was required, as locks use binary semaphores. For conditions, we implemented a new compare function similar to `get_highprio_thread()`. The new function was required since conditions store `semaphore_elem` objects that have threads stored inside the `waiters` list of their `semaphore` attribute.
>> The compare function compares two highest priority threads inside semaphores of `semapore_elem` objects. The compare function code for this is as follows :-

```
        /* Compare func to get the semaphore elem with the semaphore holding the
highest prio thread */
        static bool
        sort_condwaiters (const struct list_elem *elem_1, const struct list_elem
*elem_2, void *aux UNUSED)
```

```
        {
          struct semaphore_elem *sema_elem_1 = list_entry (elem_1, struct
semaphore_elem, elem);
          struct semaphore_elem *sema_elem_2 = list_entry (elem_2, struct
semaphore_elem, elem);

          ASSERT (&sema_elem_1->semaphore != NULL);
          struct list_elem* thread_1_elem = list_max
(&sema_elem_1->semaphore.waiters, get_highprio_thread, NULL);
          struct thread* thread_1 = list_entry(thread_1_elem, struct thread, elem);

          ASSERT (&sema_elem_2->semaphore != NULL);
          struct list_elem* thread_2_elem = list_max
(&sema_elem_2->semaphore.waiters, get_highprio_thread, NULL);
          struct thread* thread_2 = list_entry(thread_2_elem, struct thread, elem);

          ASSERT (thread_1 != NULL);
          ASSERT (thread_2 != NULL);
          return thread_1->priority < thread_2->priority;
        }


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

>> 1. When a thread tries to acquire a lock, we need to check whether the lock is
already blocked by another thread. We can utilise `lock_try_acquire()` for this.
>> 2. If the lock cannot be acquire, it could mean a case of priority inversion,
for which we begin priority donation.
>> 3. For priority donation, we iterate through all acquired(and blocked) locks of
dependent threads(nested donation) and try to update the donation priorities of
threads in any case where a lower priority thread is blocking a lock for a higher
priority thread.
>> 4. While updating thread donation priorites, we also save the new higher
priority to the lock. This is done in order to check for priority inversion.
>> The algorithm code for this implementation is as follows :-

        static void prio_donate(struct lock* lock, int priority)
        {
          while (lock != NULL)
          {
          int max_prio = (lock->max_prio > priority) ? lock->max_prio : priority;
          int donation_priority = lock->holder->donation_priority > priority ?
lock->holder->donation_priority:priority;

          lock->max_prio = max_prio;
          lock->holder->donation_priority = donation_priority;

          lock = lock->holder->locker;
          }
```

```
        }
```

>> 5. After updating all priorities, we acknowledge the lock blocking the thread by saving the lock to it.
>> 6. We then, call `sema_down()` to initiate synchronization. Which also yields CPU, calling for the scheduler(which is going to wake up appropriate process).
>> 7. We then change the holder of the lock to current running thread(whichever thread acquired lock after priority donation).
>> 8. We also save the newly acquired lock to list of acquiring thread, and change blocking lock to NULL(priority donation was successfull).

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

>> 1. When a lock is released, the donation priority needs to be updated of the thread.
>> 2. For this, we iterate through all locks acquired by thread(multiple use-case) and first remove the lock which was just released.
>> 3. We then update the donation priority of the thread to the highest priority of the thread that wants to acquire any of its locks(priority donation).

---- 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 ()` updates the priority of the calling thread, and could result in a race condition, if a priority of the calling thread is updated, where there is atleast one thread in the running queue which has higher priority. This race condition(attributing to shared running queue) causes issue to priority scheduling. To avoid this, we use `thread_yield ()` so that the scheduler is called again to deal with the changed priorities.

---- RATIONALE ----

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

>> The priority scheduler algorithm is straightforward with using `list_max()`.
>> Priority donation involves saving thread priorities effectively while also saving the blocking lock and the maximum priority of meddling threads to lock.
>> A design we considered first was updating priorities periodically inside `thread_tick ()`. We realised later that priority donation is a condition and does not occur all the time, and thus chose this design instead.
>> Another issue we had with a design was when working on multiple and nested donation use-cases. Retrieving the priority of thread blocking the lock became troublesome and almost impossible(with iterating). Then we made the current design choice to save the information inside the lock itself `int max_prio`, which proved

to be really helpful.

+---
---+

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