+--------------------+

|       CS 140       |

| PROJECT 1: THREADS |

|   DESIGN DOCUMENT  |

+--------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Akshay Chopra <[achopra6@buffalo.edu](mailto:achopra6@buffalo.edu)>

Samraddha Dubey <[samraddh@buffalo.edu](mailto:samraddh@buffalo.edu)>

Sandeep Kumar <[skumar28@buffalo.edu](mailto:skumar28@buffalo.edu)>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

As we implemented Priority Scheduling, we did some optimization while scheduling a new thread.

No need to yield current running thread if its priority is highest and no other thread

in ready queue has higher or equal priority to this thread. As yielding it will eventually schedule this higher priority thread(current) again, with the extra overhead of context switch.

High priority thread can only be yielded if there is one of the below event

1. Higher priority thread is created

2. Sleeping thread with higher priority got wake up

3. Lock released which has a higher priority thread in its waiting list (same for semaphore)

4. Priority of a thread in ready queue changes and become higher then current thread (either by thread\_set\_priority or as a result of recalculation of priority by mlfq scheduler)

However, Enforce preemption if a same priority thread is present in ready queue to maintain a round robin scheduling between same priority threads.

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

    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

{

int orig\_priority; /\*Original Priority of thread\*/

struct lock \*acquiring\_lock; /\*Lock, this thread trying to acquire\*/

struct semaphore \*acquiring\_sema; /\*Semaphore, this thread trying to ac quire\*/

struct list acquired\_locks; /\*List of acquired locks by this thread\*/

int recent\_cpu; /\* recent cpu of thread \*/

int nice\_value; /\*Nice value of thread\*/

**int64\_t sleepwait; /\*No of ticks thread needs to Sleep\*/**

};

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

In timer\_sleep() function, for a valid value of sleep ticks(saved in variable sleepwait), we will call the thread\_yieldnsleep function. In which we will disable the interrupt, push the thread element to the ‘sleep list’, block the thread and after that enable the interrupt.

When the interrupt occur at each timer\_tick, we will decrement the sleepwait value for the threads in the sleep list. If, after decrementing the value of the sleepwait changes to 0, then we will call the wakeupthread function, in which we will remove this thread from the 'sleep list’ and push it to the ‘ready list’ based on priority.

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

We Optimized scheduling of threads at each TIME\_SLICE.

No need to yield current running thread if its priority is highest and no other thread

in ready queue has higher or equal priority to this thread. As yielding it will eventually schedule this higher priority thread(current) again, with the extra overhead of context switch.

High priority thread can only be yielded if there is one of the below event

1. Higher priority thread is created

2. Sleeping thread with higher priority got wake up

3. Lock released which has a higher priority thread in its waiting list (same for semaphore)

4. Priority of a thread in ready queue changes and become higher then current thread (either by thread\_set\_priority or as a result of recalculation of priority by mlfq scheduler)

However, Enforce preemption if a same priority thread is present in ready queue to maintain a round robin scheduling between same priority threads.

This will significantly reduce the average time requires in timer interrupt handler.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer\_sleep() simultaneously?

Moving a thread from ready list to sleep list is done only after disabling interrupt so that in case multiple thread call timer\_sleep or interrupt handler arrival has no race condition.

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer\_sleep()?

We will disable the interrupt when we move from ready queue to sleep queue and enable it after that as both timer sleep and timer interrupt handler try accessing the same queues.

---- RATIONALE ----

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

>> another design you considered?

It does not have busy waiting, time spent in timer interrupt handler is minimized, no locks are used in the implementation and it has simple design.

Another design we were considering in which we are checking the sleepwait value of all the threads using allelem. Which will cause extra overhead of checking all threads even if there is no thread sleeping.

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

{

I **int orig\_priority; /\*Original Priority of thread\*/**

**struct lock \*acquiring\_lock; /\*Lock, this thread trying to acquire\*/**

**struct semaphore \*acquiring\_sema; /\*Semaphore, this thread trying to ac quire\*/**

**struct list acquired\_locks; /\*List of acquired locks by this thread\*/**

int recent\_cpu; /\* recent cpu of thread \*/

int nice\_value; /\*Nice value of thread\*/

int64\_t sleepwait; /\*No of ticks thread needs to Sleep\*/

};

struct lock

{

**struct list\_elem elem; /\*Element of lock , for the purpose of maintaining info of locks in a thread\*/**

**int holder\_priority; /\*Maintain holder's current priority related to this** lock\*/

};

>> B2: Explain the data structure used to track priority donation.

>> Use ASCII art to diagram a nested donation.  (Alternately, submit a

>> .png file.)

int orig\_priority; /\*Original Priority of thread\*/

struct lock \*acquiring\_lock; /\*Lock, this thread trying to acquire\*/

struct semaphore \*acquiring\_sema; /\*Semaphore, this thread trying to acquire\*/

struct list acquired\_locks; /\*List of acquired locks by this thread\*/

Thread which acquired a lock and having a donated priority, will moved back to its original priority once it releases the lock.

int holder\_priority; /\*Maintain holder's current priority related to this lock\*/

holder\_priority belongs to a lock, either the priority of holding thread or the donated priority.

H(20)--->M(18)--->L(15)--->A(10)

|

|

|

H(20)--->M(20)--->L(20)--->A(20)

                                                  |

                                                  | execution done

                                                  |

H(20)<---M(18)<-----L(15)<---A(10)

H get chance to run which have higher priority.

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

The waiting queue will be sorted according to the priority of threads. So when a highest priority thread comes to the waiting queue, it will go to the head of the queue. So at the time of wakeup, it will be moved to ready queue first.

>> B4: Describe the sequence of events when a call to lock\_acquire()

>> causes a priority donation.  How is nested donation handled?

If lock\_acquire() get called and a lock hold by a lower priority thread then the thread's priority will be passed to the lower priority thread. This process will be repeated if there any further lower priority thread holding the lock(can be nested upto 8 steps). Then lowest priority thread released the lock, it attains back it’s original priority and 2nd last lowest priority thread will be scheduled.

>> B5: Describe the sequence of events when lock\_release() is called

>> on a lock that a higher-priority thread is waiting for.

1. Check if there is an thread in waiting list
2. Higher priority thread at head will be removed from waiting queue.
3. It will be added to the ready queue according to its priority
4. Lock holding thread attains back its original priority or the donated priority if it acquired another lock.
5. Lock holding thread will call thread\_yield.

---- 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?

If we set priority of a thread when it is holding a lock and a higher priority thread has donated its priority. Then it may lower its priority and cause priority inersion.

As per our implementation, if a thread contains a donated priority then its priority will only be updated after it releases the lock.

---- RATIONALE ----

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

>> another design you considered?

This design has only one queue to manage priority scheduling which has low space complexity.

  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.

Added two extra field in thread structure for recent cpu and nice value.

struct thread

{

I int orig\_priority; /\*Original Priority of thread\*/

struct lock \*acquiring\_lock; /\*Lock, this thread trying to acquire\*/

struct semaphore \*acquiring\_sema; /\*Semaphore, this thread trying to ac quire\*/

struct list acquired\_locks; /\*List of acquired locks by this thread\*/

**int recent\_cpu; /\* recent cpu of thread \*/**

**int nice\_value; /\*Nice value of thread\*/**

int64\_t sleepwait; /\*No of ticks thread needs to Sleep\*/

};

static int load\_avg = 0; // To calculate system load avg

static struct list mlfq[64]; // 64 multilevel feedback queues

void update\_load\_avg(void); // to update load avg of system

void update\_recent\_cpu(void ); // to update recent cpu of threads

void calculate\_threads\_prority(void); // to calculate threads priority

int no\_of\_ready\_threads(void); // to calculate no of ready threads

---- 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:

priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2).

PRI\_MAX = 63, recent\_cpu will increase every tick for running thread and for all thread at TIMER\_FREQ

timer recent\_cpu priority thread

ticks A B C A B C 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 8 4 0 61 60 59 B

16 12 4 0 60 60 59 A

20 12 8 0 60 59 59 B

24 16 8 0 59 59 59 A

28 16 8 4 59 59 58 C

32 16 12 4 59 58 58 B

36 20 12 4 58 58 58 A

>> 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?

After 8 clock tick completion both the thread A and B have same priority so scheduler should pick thread B as per round robin scheduling. And the tread which are running at every 4rth clock tick they are decreasing their priority so Thread C which have lower priority get chance to run.

>> C4: How is the way you divided the cost of scheduling between code

>> inside and outside interrupt context likely to affect performance?

Inside interrupt instead of calling all methods (update\_load\_avg, calculate\_threads\_prority, update\_recent\_cpu) We are disabling interrupt only when it needed.

And recent cpu for every thread is calculating at each tick for running thread so we do not need to disable interrupt for that.

---- 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?

We have improved the design a bit in round robin scheduler part where as per current implementation at every TIME\_SLICE it is yielding the thread and scheduling the highest priority thread but if the yielding thread is the highest priority one then there is no need of extra overhead of context switching need to schedule same thread again.

Advantages: This design will give fair chance to run all the thread and prevent any starvation.

Disadvantage: Their might be extra overhead of calculation and our calculation based on heuristic approach and does not guarantee to be always correct.

And their might be too much context switching but we tried to improve that design in our implementation.

>> 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 load avg calculation, to preserve the proper value of load we are calculating 10000 times more. And similarly to calculate proper recent cpu we are preserving 100 times of recent cpu value. This gives us high accuracy.

We created a separate file to handle fixed-point arithmetic operation

This file gives us the required calculation as required.

Creating this separate file (abstraction) is increasing code readability and separation of mathematical calculation part.

  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?