CSE 521: OPERATING SYSTEMS

PROJECT 1: THREADS

DESIGN DOCUMENT

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.

Structure: 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. */

// Phase 2 Addition

// Declaring a variable in the struct thread

// int64_t : typedef signed long int64_t from the stdint.h library

// thread_sleep_ticks: variable to hold ticks from OS boot time, until a thread unblocks

from sleep

int64_t thread_sleep_ticks;

/* Shared between thread.c and synch.c. */

struct list_elem elem; /* List element. */

```
   int actual_priority;            /*Actual priority before donation*/
struct list *waiting_locks;    struct list locks;    int64_t
end_ticks;    struct list_elem wait_elem;

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

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

and

struct list sleep_list;

thread_sleep_ticks: variable to hold ticks from OS boot time, until a thread unblocks from sleep.

Sleep_list: list to store the sleeping threads

compare_thread_sleep(): bool compare_thread_sleep(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED);

---- ALGORITHMS ----

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

In a call to timer_sleep(), a thread's 'thread_sleep_ticks' value is updated by adding the timer_ticks() and ticks, which gives us the ticks at which the thread is supposed to wake up.

After the wakeup tick for the thread is calculated interrupts are disabled, and we insert the thread into the sleep queue in an ordered way into 'sleep_list'. After this the thread is blocked and interrupts are enabled.

In timer_interrupt(),we first increment the number of ticks, we iterate through every thread in the sleep_list and check if the ticks is greater less than thread_sleep_ticks. If the ticks is greater for a thread, i.e, it completed its sleep duration then we pop it from the sleep_list and unblock the thread. If its not greater then we break from the loop as we do not check for further as the sleep list is ordered in the increasing order of thread_sleep_ticks.

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

The thread in the sleep_list are sorted in the increasing order of the thread_sleep_ticks value, so in timer_interrupt() it does have to check if ticks greater than thread_sleep_ticks

for every thread. Once it finds ticks less than thread_sleep_ticks it just breaks from the loop as other threads will also have ticks less than thread_sleep_ticks because it is sorted. This is how we minimize the time spent in interrupt handler.

---- SYNCHRONIZATION ----

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

The function timer_sleep() disables the interrupts before adding a thread to the sleep_list, this is how it prevents race condition when multiple threads call timer_sleep().

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

Since timer_sleep() disables all interrupts before adding thread to the sleep_list, timer_interrupt will not cause any race conditions.

---- RATIONALE ----

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

In this phase we decided to use an ordered list instead of an unordered list to store sleeping threads. If we had made use of an unordered list then timer_interrupt() would have had to iterate through every sleeping threads. It is more eaicient to maintain an ordered list, with ordered insertion and then having iterate through unordered list every time timer_interrupt() is called. This is why we went ahead with the ordered list approach.

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. Structure:

```
struct thread
 {
   // Phase 3 Addition: Priority Donation
   // defining actual_priority which stores the priority which will be used for priority donation purposes
   int actual_priority;

   // list of locks: contains the list of all locks which a threads holds
struct list locks;

   // pointer to identify the lock for which the thread is waiting
struct lock *wait_lock;
   // End Phase 3 Addition: Priority Donation

   // Phase 2 Addition
   // Declaring a variable in the struct thread
   // int64_t : typedef signed long int64_t from the stdint.h library
   // thread_sleep_ticks: variable to hold ticks from OS boot time, until a thread unblocks from sleep
int64_t thread_sleep_ticks;

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

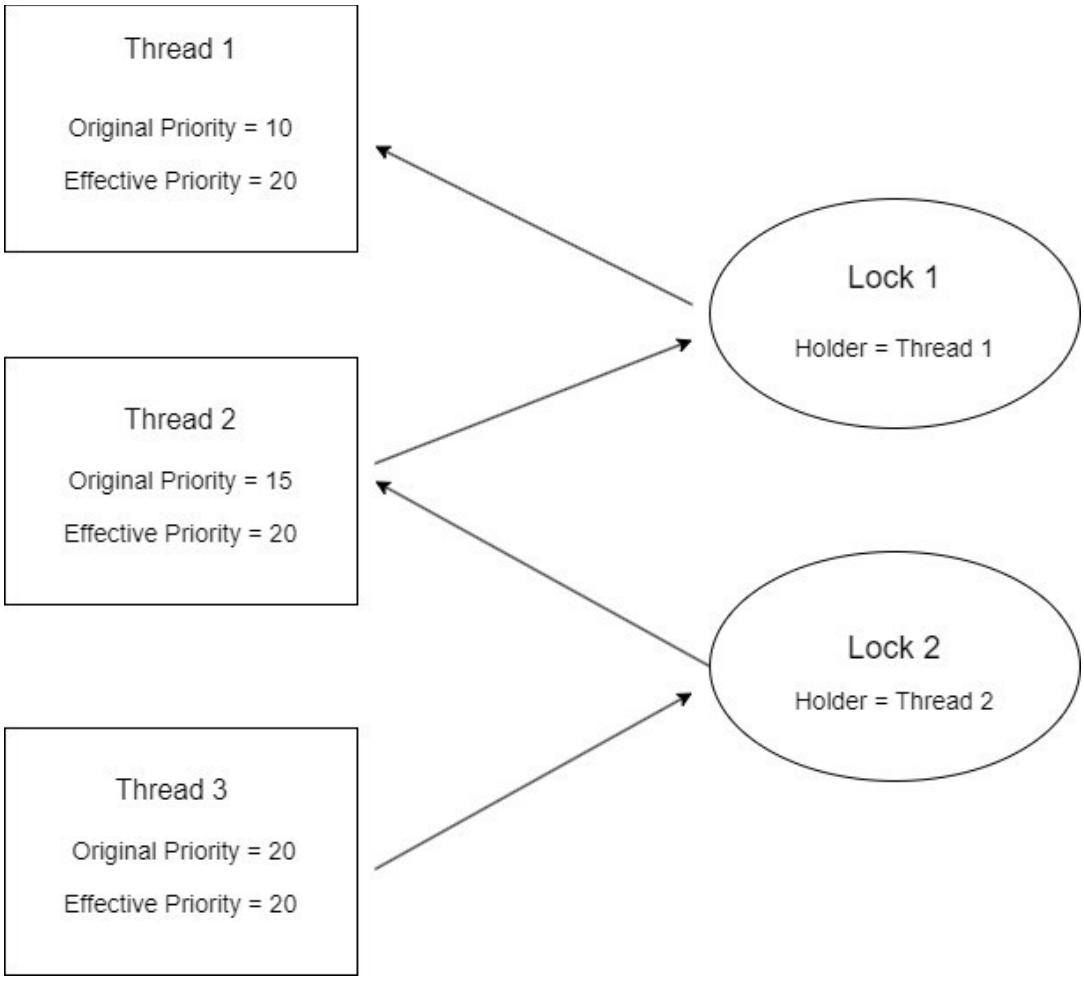
```
#ifdef USERPROG
  /* Owned by userprog/process.c. */
  uint32_t *pagedir;              /* Page directory. */
#endif

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

```
Structure:  struct lock
 {
  struct thread *holder;     /* Thread holding lock (for debugging). */    struct
semaphore semaphore; /* Binary semaphore controlling access. */
  // Phase 3 Addition: Priority Donation
  // priority of the thread which acquired the lock
int priority;
  // creating a list element for struct list locks
struct list_elem lock_element;
  // End Phase 3 Addition: Priority Donation
 };
```

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

In the below diagram, Thread 2 holds Lock2 and Thread1 holds Lock1, Since Thread3 is waiting on Lock2 and has a higher priority and hence it donates its priority to Thread2 making its eaective priority 20. Thread2 in turn is waiting on Lock 1 and has higher priority than Thread1 and hence donates its priority to Thread1.



The struct list locks in struct thread holds the list of locks that are acquired by the thread. The wait_lock points to the lock that the thread is waiting on. If a thread has to wait on another to aquire the lock and the latter has a smaller priority, then the former thread donates it's priority to the latter. The lock varible in struct lock has highest priority of the threads trying to acquire that lock.

---- ALGORITHMS ----

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

Threads are inserted into the waiting list in the decreasing order of their priority. So whenever we retrieve a thread from the waiting list we ensure that the thread with the highest priority is popped first.

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

When a call is made to lock_acquire(), if the lock is not held by any other thread then it acquires the lock. If the lock is held by another thread with a priority lower than the current thread then the current thread calls priority_donation() to donate it's priority to the latter thread. After donating its priority the current thread goes into waiting state, the wait_lock will point to the lock. If the second thread is in turn waiting on another lock held by another thread of lower priority then it will again donate its priority. This is how nested donation is handled.

>> **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 on a lock that a higher priority is waiting for, the lock is firstly freed and the lock holder value is set to NULL. If MLFQS is not enabled, the threads waiting for that lock are sorted and the highest priority thread is given priority to aquire the lock. The priority value of the lock is updated using the thread that acquired the lock and the lock element is removed from the lock. The semaphore value is then updated and if required priority donation is carried out. This whole process is carried out while disabling interrupts. This sequence of events prevents priority inversion.

---- 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?

Consider thread A holds a lock, if two threads B and C (with higher priority) tries to acquire the lock held by A then both threads will try to donate their priority to A using thread_set_priority() which could potentially result in race condition. We avoid this in the function by disabling the interrupts and re enable it once the priority is changed. This makes sure that setting priority is an atomic operation which avoids any potential race conditions.

We can use locks to avoid race condition, however, lock_acquire() itself disables interrupts to perform its operation in an atomic way. Hence, using locks will just add overhead, which is why we went with the approach of disabling interrupts for thread_set_priority().

---- RATIONALE ----

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

Assigning a lock to a thread with the highest priority after checking for the availability of the lock by assigning a lock priority and then updating the priority value of the lock to the value of the thread's priority sounded like a good implementation method for both acquiring the lock and priority donation. This approach solved two problems in a single implementation method. In the other design we thought of, we had to consider these two processes separately which would have led to performance overhead and extra code. Hence, we chose to implement this design. The current design is also simple considering the other design we thought of.

ADVANCED SCHEDULER ==========

---- DATA STRUCTURES ----

C1: Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, `typedef', orenumeration. Identify the purpose of each in 25 words or less.

Changed Structure 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. */

   // Phase 3 Addition: Priority Donation
```

```
   // defining actual_priority which stores the priority which will be used for priority donation purposes
   int actual_priority;

   // list of locks: contains the list of all locks which a threads holds
struct list locks;

   // pointer to identify the lock for which the thread is waiting
struct lock *wait_lock;
   // End Phase 3 Addition: Priority Donation

   // Phase 3 Addition: MLFQS
   // declare an integer 'nice' that holds the niceness value of a thread. Value ranges between 20 and -20
   int nice;

   // recent cpu to measure how much CPU time each process has received recently
   // fixed_point_t declared in "threads/fixedpoint.h" header file
fixed_point_t recent_cpu;
   // End Phase 3 Addition: MLFQS

   // Phase 2 Addition
   // Declaring a variable in the struct thread
   // int64_t : typedef signed long int64_t from the stdint.h library
   // thread_sleep_ticks: variable to hold ticks from OS boot time, until a thread unblocks from sleep
int64_t thread_sleep_ticks;

   /* 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. */
 };
```

Variables included:

1. nice: holds the niceness value of a thread. Value ranges between 20 and -20
2. recent_cpu: measures how much CPU time each process has received recently
3. load_avg: The system load average that estimates the average number of threads ready to run over the past minute

Functions declared

1. get_recent_cpu() : void get_recent_cpu(struct thread *ready_list_thread, void *aux UNUSED);
2. get_load_avg() : void get_load_avg(struct thread *ready_list_thread);
3. mlfqs_thread_priority(): void mlfqs_thread_priority(struct thread *ready_list_thread, void *aux UNUSED);

---- 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			Priority			Next thread to run
	A	B	C	A	B	C	

0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

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?

No, there were no ambiguities in the scheduler specification. The implementation of the function compare_thread_priority() made sure that when two threads with same priority are encountered, the function will return result in such a way that the thread in the ready list will preempt the running thread which holds the same priority, this makes sure that if a running thread and a thread in ready queue have same priorities, then the thread in ready queue will be given a chance to run since the running thread already used some CPU time. This makes the implementation eaicient as it enables a thread with no CPU time to run first before a thread which has recent_cpu time.

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

To reduce the performace overhead, I made the calculations for the variables 'recent_cpu', 'load_avg' in an external interrupt context in a function thread_tick() which runs for every tick. The boolean value 'thread_mlfs' was used to make sure that these calculations were performed only when the --mlfqs flag was passed to the OS during boot.

Moreover the calculations for 'recent_cpu' and 'load_avg' were performed for every multiple of second i.e 100 ticks. This divided and reduced the cost of scheduling between the code inside and outside interrupt context. The performance overhead for our design was less because we made these calculations in an external interrupt context.

---- 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 main advantages of my design were that

a. The calculations for recent_cpu and load_avg were performed for only a multiple of a second i.e. for every 100 ticks which reduced the performance overhead

b. The MLFQS code was executed only when the MLFQS flag was passed to the OS during boot

c. Calculations were performed in an external interrupt context

The disadvantages of my design were

a. I used linked lists for the implementation. Implementation of min and max heaps can be done which are faster than lists. Heapify operations can reduce the sorting overhead.

b. Lot of sub functions were called during calculations for priority, recent_cpu and load_avg from the fixed_point header file

I could have improved the functions and written new fixed_point functions.

Also another issue was overflow due to multiplying load avg by recent CPU directly. I could have built on this to handle the overflow.

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?

The fixed point header file was already provided to us which used an abstraction layer for the fixed point arithmetic

calculations. Pintos does not support floating-point arithmetic in the kernel, hence the file was needed to be defined

to simulate fixed point calculations. This also reduced the overhead on the kernel by including it separately.

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.

Q) 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?

Ans) The assignment on the whole was a wonderful experience. We loved delving into the concepts of operating systems and how they are implemented. Working with synchronization, thread concepts was a great practical learning experience.

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

Ans) Yes, while implementing priority donation, we had to consider all situations. For priority scheduling we had to think about a condition when equal priority threads called the thread compare function. We handled all these cases, and implemented the same in our design.

Q) 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?

Ans) All the hints were provided to us through TA's project class and the recitation videos. The pintos documentation was also very helpful while building and implementing the code.

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

Ans) None, the TA's did a wonderful job assisting students and their doubts.

Q) Any other comments?

The project was challenging, but equally rewarding as it made us implement theory into practical concepts.