CS39002
Operating Systems Laboratory
Spring Semester 2020-2021

ASSIGNMENT 4: Implementation of Mlfqs in PintOS
DESIGN DOCUMENT

GROUP 8		
Atharva Naik Roshan	**18CS10067**	**atharvanaik2018@gmal.com**
Radhika Patwari	**18CS10062**	**rsrkpatwari1234@gmail.com**

ADVANCED SCHEDULER - MLFQS
=========================

---- 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.

Answer:

a) Added a file to perform floating-point operations as `$HOME/pintos/src/threads/fixed_point.h`

```
#ifndef __THREAD_FIXED_POINT_H
#define __THREAD_FIXED_POINT_H

/* Basic definitions of fixed point. */
typedef int fixed_t;
/* 16 LSB used for fractional part. */
#define FP_SHIFT_AMOUNT 16

/* Some helpful macros. */
/* Convert a value to fixed-point value. */
#define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
/* Add two fixed-point value. */
#define FP_ADD(A,B) (A + B)
/* Add a fixed-point value A and an int value B. */
#define FP_ADD_MIX(A,B) (A + (B << FP_SHIFT_AMOUNT))
/* Substract two fixed-point value. */
#define FP_SUB(A,B) (A - B)
/* Substract an int value B from a fixed-point value A */
#define FP_SUB_MIX(A,B) (A - (B << FP_SHIFT_AMOUNT))
```

```
/* Multiply a fixed-point value A by an int value B. */
#define FP_MULT_MIX(A,B) (A * B)
/* Divide a fixed-point value A by an int value B. */
#define FP_DIV_MIX(A,B) (A / B)
/* Multiply two fixed-point value. */
#define FP_MULT(A,B) ((fixed_t)(((int64_t) A) * B >> FP_SHIFT_AMOUNT))
/* Divide two fixed-point value. */
#define FP_DIV(A,B) ((fixed_t)((((int64_t) A) << FP_SHIFT_AMOUNT) / B))
/* Get integer part of a fixed-point value. */
#define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
/* Get rounded integer of a fixed-point value. */
#define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >>
FP_SHIFT_AMOUNT) \
                              : ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))

#endif /* thread/fixed_point.h */
```

->Variables like `priority, nice` and `ready_threads` are integers. But `recent_cpu` and `load_avg` are real numbers and Pintos does not support floating-point arithmetic in the kernel as it complicates and slows down the kernel. Hence we define all the required mathematical operations in a separate file using macros.

b) Added header file, macro, variable and function definitions in `$HOME/pintos/src/threads/thread.h`

```
/* Assignment 4 : Part 2 : Code added */
#include "fixed_point.h"
/* Assignment 4 : Part 2 : Code ended */

/* Assignment 4 : Part 2 : Code added */
#define NICE_MIN -20          /* Minimum negative nice value : Max priority */
#define NICE_DEFAULT 0        /* default nice value (no effect on thread priority)*/
#define NICE_MAX 20           /* Maximum positive nice value : least priority */
/* Assignment 4 : Part 2 : Code ended */
```

-> The file fixed_point.h used for floating point operations is included in this file. To declare the nice values, we use macros and set the maximum and minimum values of nice values as `NICE_MAX` and `NICE_MIN` respectively.

c) Added header file, macro, variable and function definitions in `$HOME/pintos/src/threads/thread.h`

```
/* Assignment 4 : Part 2 : Code added */
int nice;                     /* Nice value of the thread */
int recent_cpu;               /* Recent CPU usage */
/* Assignment 4 : Part 2 : Code ended */

/* Assignment 4 : Part 2 : Code added */
void thread_test_preemption(void);

void thread_mlfqs_incr_recent_cpu(void);
void thread_mlfqs_calc_recent_cpu(struct thread *);
```

void thread_mlfqs_update_priority(struct thread *);
void thread_mlfqs_refresh(void);

bool compare_thread_priority(struct list_elem *a, struct list_elem *b);
/* Assignment 4 : Part 2 : Code ended */

-> `nice` is an integer value to determine how “nice” the thread should be to other threads ; `recent_cpu` estimates the amount of CPU time the thread has received "recently," with the rate of decay inversely proportional to the number of threads competing for the CPU.

-> Function definitions of Multilevel Feedback queue scheduling are declared globally in the `thread.h` file so make them accessible to `timer.c` file as well. The main body of all these functions is defined in `$HOME/pintos/src/thread/thread.c` file.

d) Added a function to preempt a lower priority running thread in `$HOME/pintos/src/thread/thread.c`

```
/* Test if current thread should be preempted. */
void
thread_test_preemption (void)
{
  enum intr_level old_level = intr_disable ();
  if (!list_empty (&ready_list) && thread_current ()->priority <
      list_entry (list_front (&ready_list), struct thread,
elem)->priority)
      thread_yield ();
  intr_set_level (old_level);
}
```

-> If the priority of the currently running thread is lower than priority of the highest priority thread of the ready queue, then we preempt this running thread and re-schedule. Interrupts are disabled to avoid race condition as this is a critical section.

e) Added a function to update recent_cpu of running thread in `$HOME/pintos/src/thread/thread.c`

```
/* Called every time a Timer Interrupt occurs */
/* Increase current thread's recent_cpu by 1 */
void
thread_mlfqs_incr_recent_cpu(void)
{
  ASSERT (thread_mlfqs);
  ASSERT (intr_context ());

  struct thread *t = thread_current ();
  if (t == idle_thread)
      return;
  t->recent_cpu = FP_ADD_MIX (t->recent_cpu, 1);
}
```

-> This function works only when mlfqs algorithm is used.Initial value of `recent_cpu` is 0 in the first thread created, or the parent's value in other new threads.Each time a timer interrupt occurs, `recent_cpu` is incremented by 1 for the running thread only, unless the idle thread is running.

f) Added a function to preempt a lower priority running thread in

`$HOME/pintos/src/thread/thread.c`

```
/* Updated once per second */
/* Calculate thread's recent_cpu */
void
thread_mlfqs_calc_recent_cpu(struct thread *t)
{
  ASSERT (thread_mlfqs);
  ASSERT (t != idle_thread);

  fixed_t coef = FP_DIV (FP_MULT_MIX (load_avg, 2),
                        FP_ADD_MIX (FP_MULT_MIX (load_avg, 2), 1));
  fixed_t term = FP_MULT (coef, t->recent_cpu);
  t->recent_cpu = FP_ADD_MIX (term, t->nice);
}
```

-> This function works only when mlfqs algorithm is used.When the system tick counter reaches a multiple of a second, that is, when `timer_ticks () % TIMER_FREQ == 0`, the value of `recent_cpu` is recalculated for every thread (whether running, ready, or blocked) using this function. The calculation is based on the formula:
`recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice`

g) Added a function to preempt a lower priority running thread in
`$HOME/pintos/src/thread/thread.c`

```
/* Updated once per second */
/* Update thread's priority */
void
thread_mlfqs_update_priority(struct thread *t)
{
  if (t == idle_thread)
      return;

  ASSERT (thread_mlfqs);
  ASSERT (t != idle_thread);

  fixed_t new_priority = FP_CONST (PRI_MAX);
  new_priority = FP_SUB (new_priority, FP_DIV_MIX (t->recent_cpu, 4));
  new_priority = FP_SUB_MIX (new_priority, 2 * t->nice);
  t->priority = FP_INT_PART (new_priority);
  if (t->priority < PRI_MIN)
      t->priority = PRI_MIN;
  else if (t->priority > PRI_MAX)
      t->priority = PRI_MAX;
}
```

-> This function works only when mlfqs algorithm is used.Thread priority is calculated initially at thread initialization. At every fourth clock tick and also after updating `recent_cpu` value, for every thread, priority is recomputed using the formula
`priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)`

The calculated priority is always adjusted to lie in the valid range `PRI_MIN` to `PRI_MAX.`

This formula gives a thread that has received CPU time recently lower priority for being reassigned the CPU the next time the scheduler runs. This is key to preventing starvation: a thread that has not received any CPU time recently will have a `recent_cpu` of 0, which barring a high nice value should ensure that it receives CPU time soon.

h) Added a function to update system load average in `$HOME/pintos/src/thread/thread.c`

```
/* Called once per second */
/* Invoked once per second to refresh load_avg
   and recent_cpu of all threads. */
void
thread_mlfqs_refresh(void)
{
  ASSERT (thread_mlfqs);
  ASSERT (intr_context ());

  /* Calculate load_avg per second. */
  size_t ready_threads = list_size (&ready_list);
  if (thread_current () != idle_thread)
      ready_threads++;
  load_avg = FP_ADD (FP_DIV_MIX (FP_MULT_MIX (load_avg, 59), 60),
                    FP_DIV_MIX (FP_CONST (ready_threads), 60));

  /* recent_cpu is recalculated for every thread per second. */
  struct thread *t;
  struct list_elem *e = list_begin (&all_list);
  for (; e != list_end (&all_list); e = list_next (e))
      {
      t = list_entry(e, struct thread, allelem);
      if (t != idle_thread)
      {
      thread_mlfqs_calc_recent_cpu (t);
      thread_mlfqs_update_priority (t);
      }
      }
}
```

-> This function works only when mlfqs algorithm is used.When the system tick counter reaches a multiple of a second, that is, `when timer_ticks () % TIMER_FREQ == 0`,this function is called. It recomputes the system load average(system-wide property and not thread-specific). At system boot, it is initialized to 0. It is updated according to the following formula:

`load_avg = (59/60)*load_avg + (1/60)*ready_threads`

Where `ready_thread` is the number of threads that are either running or ready to run at time of update (not including the idle thread).

After computing `load_avg`,for every thread (whether running, ready, or blocked), `recent_cpu and priority` values are updated.

i) Added a function to maintain mlfqs structure using a single queue in `$HOME/pintos/src/thread/thread.c`

/* A comparator for comparing the thread's priority for the ordered lists */
bool
compare_thread_priority(struct list_elem *a, struct list_elem *b){
return list_entry(a, struct thread, elem)->priority > list_entry(b, struct thread, elem)->priority;
}

-> The threads in the ready queue are arranged in the decreasing order of their priority from

0 `(PRI_MIN)` to 63 `(PRI_MAX)` to ensure mlfqs structure so that threads of higher priority are always scheduled first. In case of threads of similar priority values, they are scheduled in round-robin technique such that a new thread is always inserted behind the already existing threads with similar priority.

j) Added header file in `$HOME/pintos/src/thread/thread.c`

#include "threads/fixed_point.h"

-> Including the file created for floating point operations in the thread.c for computing `priority, nice, ready_threads, recent_cpu` and `load_avg` values.

k) Added variables in `$HOME/pintos/src/thread/thread.c`

fixed_t load_avg;
load_avg = FP_CONST (0);

-> `load_avg` estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second. It is declared as a global variable in thread.c file.It is initialised to 0 in the `thread_start()` function.

l) Adding preemptive function in `$HOME/pintos/src/thread/thread.c`

/* Test preemtpion. */
thread_test_preemption ();

-> When a new thread is created in `thread_create()` , it may happen that the priority of this process is more than the priority of the running process. So `thread_test_preemption()` is called to preempt the lower priority thread and schedule the process with maximum priority to the CPU.

m) Comparing priorities while pushing into ready queue in `$HOME/pintos/src/thread/thread.c`

list_insert_ordered(&ready_list, &t->elem, compare_thread_priority, NULL);

-> In `thread_unblock()`, after unblocking the thread, it is pushed into the ready queue such that it is inserted after all threads having greater than or equal to priority than this thread. This ensures that threads with different priorities are scheduled using Multilevel Feedback Queue scheduling approach and threads with similar priority are scheduled using Round-Robin scheduling approach.
It is also used in `thread_yield()` to push current running thread into the ready queue.

n) Initialising threads in `$HOME/pintos/src/thread/thread.c`

```
/* Does basic initialization of T as a blocked thread named
   NAME. */
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);
```

```
  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;

  /* Assignment 4 : Part 2 : Code added */
  t->nice = 0;
  t->recent_cpu = FP_CONST (0);

  enum intr_level old_level = intr_disable ();
  /* Assignment 4 : Part 2 : Code ended */

  list_push_back (&all_list, &t->allelem);

  /* Assignment 4 : Part 2 : Code added */
  intr_set_level (old_level);
  /* Assignment 4 : Part 2 : Code ended */
}
```

-> During initialisation of a thread, we set `nice` and `recent_cpu` values to 0. We disable the interrupt while pushing this thread into the list of all threads.

o) Calling mlfqs functions in `$HOME/pintos/src/devices/timer.c`

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  /* Assignment 4 : Part 2 : Code added and removed */
  ticks++;
  thread_tick ();
  /* Actions for Multilevel feedback queue scheduler. */
  if (thread_mlfqs)
  {
     thread_mlfqs_incr_recent_cpu ();
     if (ticks % TIMER_FREQ == 0)
          thread_mlfqs_refresh ();
     else if (ticks % 4 == 0)
          thread_mlfqs_update_priority (thread_current ());

          bool preempt = false;

          /* Check and wake up sleeping threads. */
          struct thread *t;
          while(!list_empty(&sleeping_threads)) {

          t = list_entry(list_front(&sleeping_threads), struct thread, elem);
          if (timer_ticks() < t->abs_ticks)
          break;
```

```
        list_pop_front (&sleeping_threads);
        thread_unblock(t);
        preempt = true;
        }

        if (preempt)
        intr_yield_on_return ();
  }
  /* Assignment 4 : Part 2 : Code ended */
}
```

-> When mlfqs scheduling approach is used, at every second (taken as `ticks%TIMER_FREQ`) to ensure working of some test cases, the system `load_avg,recent_cpu` and `priority` value of all threads is updated.At every 4rt tick, priority of the current thread is updated. Then we wake up the sleeping threads in the while loop and set `preempt` as true to ensure preemption of running threads in case thread with a higher priority is ready to execute.

p) Checking priority of the threads in priority scheduling in `$HOME/pintos/src/thread/thread.c`

```
void
thread_set_priority (int new_priority)
{
  /* No need to change priority in case of mlfqs */
  if(!thread_mlfqs){

        /* update only when running in the priority mode */
        thread_current ()->priority = new_priority;
        if(!list_empty(&ready_list)){
        struct thread *front_ele = list_entry(list_front(&ready_list), struct thread, elem);

        /* if the current thread has less priority than any ready thread, then preempt
the current thread and schedule */
        if(front_ele->priority > thread_get_priority())
        thread_yield();
        }
  }
}
```

-> The function is used to set priority when normal priority scheduling is used (without mlfqs). If the priority of the current running thread is lower than the priority of the thread in the ready queue, then `thread_yield()` is called to preempt the current thread and schedule the thread of highest priority in the CPU.

---- 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	B	C	priority A	B	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	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

Explanation:

0: We schedule A first as it has the highest priority (least nice value)
4: After 4 ticks priority of A decreases by recent_cpu_time/4.= 1, i.e. from 63 to 62, we still
, schedule A as it has the highest priority
8: After 4 more ticks priority of A decreases by recent_cpu_time/4.= 1, i.e. from 62 to 61. B
, and A now have the same priority, so we schedule B now (in a round robin way).
12: A gets scheduled as it has the highest priority
16: A and B are tied, so we break the tie in a round-robin way and schedule B
20: A has highest priority so it gets scheduled
24: A,B,C all have same priority so we schedule C next, in a round robin fashion
28: B gets scheduled when there is a tie between A and B (same reason as above)
32: A gets scheduled next as it has the highest priority
36: A,B,C all have same priority so we schedule C next, in a round robin fashion

>> 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?

-> If the running thread has the same priority as some thread in the ready queue, the scheduler will take the one in the ready queue and then in the next time slice will do the same as round-robin. Yes this match with the scheduler as the highest priority one is still the one in the running state but this is done to deliver a more responsive system.

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

-> Most of the calculations for `recent_cpu` and `priority` are done within timer interrupt every fixed number of ticks. Redundancy of calculations in `thread_tick ()` was managed to be cut down by updating `priority` only for the currently running thread every 4 ticks, and for all threads every 1 sec.

---- 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?

-> Implementation has some advantages in context of

- design simplicity, as it offers a relatively small thread struct size , only fixed sized integer variables for nice, `recent_cpu` and `abs_ticks` are used
- The usage of `list_insert_ordered()` in place of `list_push_back()` for inserting threads into the queue ensures that the threads are maintained in the order of priority (as in `ready_list`) or wake up time (as in `sleeping_threads`). This reduces the overhead of performing O(n log n) sortings on a ready queue every time a new thread is scheduled. Also in `sleeping_threads`, the front element of the queue is enough to signify if any thread needs to be woken up
- redundancies in calculations are avoided as much as possible by updating priority only for the currently running thread after every 4 ticks.

As for disadvantages, usage of linked lists greatly affects performance, as ordered insertions and modifications running in O(n) complexity are frequently used in code.

One suggestion includes usage of a priority queue (Min/Max heaps) that supports insertion and modification in O(log n) time with some workaround to ensure stability, and eliminating the need to sort completely. Finally, considerations to refine the design may include detection of overflow in fixed-point operations in `fixed_point.h`

>> 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?

-> `fixed-point.h` is implemented completely by using macros that support fixed-point arithmetic operations, as they are usually faster as they do not need a function call or an activation record, also it allows the compiler to optimize the equations. Real numbers throughout implementation of the scheduler like those used in `recent_cpu` and `load_avg` are simulated using fixed-point representation rather than floating-point arithmetic representation which is marginally slower. Pintos does not support floating-point arithmetic in the kernel, because it would complicate and slow the kernel. Hence `fixed_point.h` provides an abstraction layer for fixed-point math.