+---+
| CS39002 |
| ASSIGNMENT 4: Implement wakeup thread and multilevel feedback queue (MLFQS) |
| DESIGN DOCUMENT |
+---+

---- GROUP 22 ----

(18CS30047) Somnath Jena <somnathjena2011@gmail.com>
(18CS10069) Siba Smarak Panigrahi <sibasmarak.p@gmail.com>

---- PRELIMINARIES ----

1. **Terminating qemu simulator**:
   After successful execution of pintos run alarm-multiple (or any similar run command) the qemu simulator did not terminate. We added an additional -q flag (**pintos -- -q run alarm-multiple**). It led to the termination of the qemu simulator after successful execution. Further, while executing the make commands, the inbuilt make file adds the above -q flag.
2. Additionally we had to change the **shutdown.c**, according to Timeout in tests when running pintos. This led to the prevention of the TIMEOUT after a certain (generally 61) seconds) issue on qemu simulator.

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.

1. **In pintos/src/threads/thread.h**
   MACROS:

```
#define F1714 16384                 /* f=2^14 for converting between int and float in 17.14 format */

/* Thread nice values */
#define NICE_DEFAULT 0              /* Default nice value. */
#define NICE_MIN -20                /* Lowest nice value. */
#define NICE_MAX 20                 /* Highest nice value. */
```

Modifications/Additions:

F1714: used to represent the value of 2^14 for converting between int and float. Since Pintos does not support floating point arithmetic in the kernel, floating point values are represented by designating the lower 14 bits as fractional bits.

NICE_DEFAULT: used to represent the default nice value of 0

NICE_MIN: used to represent the minimum nice value of -20

NICE_MAX: used to represent the maximum nice value of 20

STRUCTURES:

```
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 nice;                   /* Nice value. */
    int recent_cpu;             /* recent cpu time. */
    struct list_elem allelem;   /* List element for all threads list. */

    int64_t wake_tick;          /* tick when a sleeping thread will wake up */
    struct list_elem sleepelem; /* list element for all sleeping threads list */

    /* 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. */
 };
```

Modifications/Additions:

struct list_elem sleepelem: list element embedded into the struct thread which is part of the sleeping_threads list that maintains a list of all threads which are sleeping, i.e. in THREAD_BLOCKED state and waiting upto wake_tick ticks.

int64_t wake_tick: this variable stores the tick at which the thread will wake if it is sleeping, i.e. sum of start time of sleep and ticks to sleep.
int nice: this variable stores the nice value of the thread, that determines how "nice" the thread should be to other threads. Priority of a thread changes as per the nice value.
int priority: this variable stores the priority of the thread.

2. **In pintos/src/threads/thread.c**
GLOBAL/STATIC VARIABLES:

```
/* Wakeup thread, wakes up all the sleeping threads. */
static struct thread *wakeup_thread;

/* List of threads in THREAD_BLOCKED state, i.e. processes that
   sleeping and waiting to be ready once the ticks reach the wake_tick */
static struct list sleeping_threads;

static unsigned total_ticks;          /* # of timer ticks since OS booted. */
static unsigned min_waketick;         /* wake_tick value of the 1st thread in sleeping_threads list
                                         if no thread in list, then maximum*/

static int load_avg;                  /* system wide load average*/
```

Modifications/Additions:
static struct thread *wakeup_thread: a managerial thread that wakes up the sleeping threads at the right time, It receives highest priority and cannot be preempted while it is in THREAD_RUNNING state.
static struct list sleeping_threads: static variable of type struct list which consists of struct list_elem members embedded inside struct thread. It maintains the list of all threads currently sleeping and in THREAD_BLOCKED state.
static unsigned total_ticks: a variable similar to **ticks** in timer.c. It stores the number of timer ticks since OS booted and is updated every time thread_tick() is invoked.
static unsigned min_waketick: wake_tick value of the first thread in sleeping_threads list. If there is no thread in the list, then it is INT32_MAX. It is updated in thread_tick() as well as thread_wakeup().
static int load_avg: stores the system load average, which estimates the average number of threads ready to run over the past minute. It is system-wide, initialized to 0, and updated every second as an exponentially-weighted average.

---- 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			thread to run	ready_list
	A	B	C	A	B	C		
0	0	0	0	63	61	59	A	A->B->C
4	4	0	0	62	61	59	A	A->B->C
8	8	0	0	61	61	59	B	B->A->C
12	8	4	0	61	60	59	A	A->B->C
16	12	4	0	60	60	59	B	B->A->C
20	12	8	0	60	59	59	A	A->C->B
24	16	8	0	59	59	59	C	C->B->A
28	16	8	4	59	59	58	B	B->A->C
32	16	12	4	59	58	58	A	A->C->B
36	20	12	4	58	58	58	C	C->B->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?

There is ambiguity of scheduling which thread, when the priority of two or more threads are the same after a time slice which makes the values in the table uncertain.

We used **Round Robin Scheduling and FIFO concepts** to resolve the tie.

Yes, it matches the behaviour of our scheduler. The **priority_list_less_func()** works as the comparator and helps to sort the threads in ready_list and is also used as the basis function for **list_insert_ordered()** Our ready_list essentially simulates the 64 ready queues in a single list. This works because the time quantum for each ready queue in the given scheduler is the same. Now after each time slice we update priorities of all the threads(actually updating only the current thread's priority suffices). After that when **thread_yield()** is called it actually inserts the current thread in the ready list in order using the **list_insert_ordered()** function. Say current thread A has priority 50 after update, and priority of B in ready_list is also 50. The comparator function is such that it inserts the current thread in the ready_list before a thread only if current thread's priority is greater than that thread. So A cannot be inserted before B and hence is inserted after B. Thus A essentially gets added to the queue at the end and B will get a chance earlier than A next. This ensures fair scheduling.

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

If the CPU spends a lot of time in **timer_interrupt(),** such as calculating recent_cpu, load_avg and accordingly priority updation, and sorting of ready_list this would add to the time just before thread preemption. The running time for the current running thread is occupied in this computation and thus reduces its effective time spent on CPU. This affects the scheduling procedure. Thus increase in the cost of scheduling inside the interrupt context leads to decrease the performance.

---- 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?

Currently our design does not use 64 queues for multilevel feedback queue scheduling (MLFQS) explicitly. Present implementation uses a single ready_list for all the threads having their state as THREAD_READY. Once the priority is set for a new thread (during **init_thread()**) the thread is inserted into ready_list with the aid of **list_ordered_func()** (inserts into ready_list respecting the priority of other ready threads). In another scenario, when priority is updated for threads in ready_list after every TIME_SLICE (4 ticks in thread_tick) using **update_priority()**, the entire ready_list is sorted with respect to priorities.

ADVANTAGES:

One particular advantage is that a **single ready_list suffices** for the entire arrangement of the MLFQS. This is because the ready_list is sorted at critical times and Round Robin scheduling is used in ready queues at each priority level with the same time slice. Thus a lower priority thread can only be assigned to the CPU only when there is no higher priority thread in the ready_list (i.e. the lower priority is the higher priority in ready_list). This makes the implementation **simple** and **better understandable**. Accessing the next thread to schedule takes O(1) time.

DISADVANTAGES:

One major disadvantage is the sheer number of times the ready_list is needed to sort. This is an inefficient approach **when the number of threads increases to a large value** since sorting takes O(nlogn) time. The sorting of ready_list would consume a larger time and thus making the system in-efficient.

To refine the design, we could use **Max-Heap data-structure** to maintain the ready_list. The insertion time would be worst case O(logn). But the threads would be in priority respecting order. Thus it would reduce the time spent in the sorting of ready_list. Also, we could use an array of **64 separate lists** for the ready queue, but that was not necessary in our case.

>> 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?

MACROS:

We used a macro **F1714** to represent the factor of 2^14 (16384) to convert between integer and floating point representations of a number. One thing to note is that we did not define it as 1<<14, instead we defined it directly as 16384. Defining it as 1<<14 was causing errors in calculations that involve multiplication and division.

FUNCTIONS:

to_fp(): used to convert a number to its floating point representation by multiplying with F1714
mul_fp(): used to multiply two numbers in their floating point representations
div_fp(): used to divide two numbers in their floating point representations
to_int_round(): used to convert and round off a number in floating point representation into normal integer representation.

recent_cpu of each thread, **load_avg** of the system are actually floating point numbers. While recent_cpu is updated in **update_recent_cpu()** and incremented in **increment_recent_cpu()**, load_avg is updated in **update_load_avg()**. Similarly while updating priority and nice values recent_cpu and load_avg values need to be rounded off to the nearest integers. Since floating point math is used in so many places, and writing the conversion expressions at each place would have made the code cumbersome, we decided to implement the above four functions and call them as and when necessary.