Design Document for Project 1: Threads

Name: 卓尔 (Zhuo Er)
SID: 11611026

Task 1: Efficient Alarm Clock

1. Data structures and functions

Modified Structs

`int64_t ticks_sleep;`

- added to `struct thread` as a member variable
- keep track of the remaining ticks for a thread to sleep

Modified Functions

`void timer_sleep (int64_t ticks)`

- block the current thread
- update the `ticks_sleep` of the current thread to be `ticks`

`tid_t thread_create (const char *name, int priority, thread_func *function, void *aux)`

- initialize `ticks_sleep` to be 0

`static void timer_interrupt (struct intr_frame * args UNUSED)`

- For each thread, apply `check_blocked_thread`

Added Functions

`void check_blocked_thread (struct thread *t, void *aux UNUSED)`

- added to `thread.c`
- check and decrease the value of `ticks_sleep` for each blocked thread
- unblock the blocked thread if `ticks_sleep` equals 0

2. Algorithms

Main Idea

Block the current thread and record the ticks for sleeping whenever we want a thread to sleep. Every `timer_interrupt` (invoked per tick), the `ticks_sleep` is checked and decreased for each blocked thread. Once there's no remaining ticks for sleeping, we wake up the thread by unblocking it.

3. Synchronization

We can disable interrupts before accessing `ticks` and updating `ticks_sleep` in order to keep the states synchronized. We can also disable interrupts when we call `thread_unblock` to unblock a thread and push it back to the ready list.

4. Rationale

I have considered two approaches for updating the value of `ticks_sleep`. One is to iterate and check each blocked thread (this requires `O(n)` time), the other is to maintain a sorted list of the blocked threads and only check the first block thread(s). Although a sorted list could make the checking process more efficient, I prefer the former approach based on the following reasons.

1. A sorted list needs additional space.
2. Inserting a sleeping thread into a sorted list also requires `O(n)` time.
3. Maintaining a sorted list needs more coding.

Therefore, I prefer my current choice.

Task 2: Priority Scheduler

1. Data structures and functions

Modified Structs

`int base_priority;`

- added to `struct thread` as a member variable
- keep track of the base priority of a thread
- subject to `thread_set_priority`

`struct list locks;`

- added to `struct thread` as a member variable
- keep track of all the locks held by threads

`struct lock *lock_waiting;`

- added to `struct thread` as a member variable
- keep track of the lock the thread is currently waiting for

`struct list_elem elem;`

- added to `struct lock` as a member variable
- keep track of the threads waiting for the lock

Modified Functions

`static void init_thread (struct thread *t, const char *name, int priority)`

- initialize `base_prioriy` to be `priority`
- initialize `locks` to be a list
- initialize `lock_waiting` to be `NULL`

`static struct thread * next_thread_to_run (void)`

- iterate through the `ready_list`
- return the thread with highest priority based on `thread_get_priority`

`void lock_acquire (struct lock *lock)`

- recursively call the lock `holder` and donate the current thread's priority to the `holder` if the current thread has a higher priority
- update the lock `holder` to the current thread after a successful acquire
- update the locks held by the current thread in `locks` after a successful acquire
- call `thread_yield()` for each change in `priority`

`void lock_release (struct lock *lock)`

- update the `priority` of the thread to `max(base_priority, max(the max priority of each lock held by the thread))`
- call `thread_yield()` for each change in `priority`

`void thread_set_priority (int new_priority)`

- always set the `base_priority`
- set `priority` to `new_priority` if `new_priority` is greater than `priority`
- Otherwise, set `priority` of the thread to `max(base_priority, max(the max priority of each lock held by the thread))`
- call `thread_yield()` for each change in `priority`

`void cond_signal (struct condition *cond, struct lock *lock UNUSED)`

- sort the `waiters` before `sema_up`

2. Algorithms

Choosing the next thread to run

In order to take the `priority` into account, we iterate through the ready list instead of directly pop the front thread. We return the thread with the highest `priority` for the next run. The `priority` here can be obtained by calling `thread_get_priority`. The entire process is executed in `next_thread_to_run`.

Acquiring Locks

When acquiring locks held by a `holder` with lower priority, the current thread should donate its priority to the `holder`. This process goes recursively until the `lock_waiting` is `NULL` or the `holder` has a higher priority. Update the holder of the lock and the locks held by the thread after a succesful acquire.

Releasing a Lock

When releasing a lock, we call `lock_release` and remove the lock from the list of locks held by the thread. Update the `priority` of the thread to `max(base_priority, max(the max priority of each lock held by the thread))`. Call `thread_yield()` for each change in `priority`.

Computing the effective priority

The effective priority is computed each time `lock_acquire`, `lock_release` or `thread_set_priority` is invoked. Donation, `base_priority` and `new_priority` are all considered in the computations.

Priority scheduling for semaphores and locks

Before `sema_up` in `cond_signal`, we first sort the `waiters` list by priority. Therefore, `sema_up` will unblock the front thread in the list which should be of highest priority. When acquiring locks held by a `holder` with lower priority, the current thread should donate its priority to the `holder`.

Changing thread's priority

Changing thread's priority is done by calling `thread_set_priority`. The `priority` will be changed to `new_priority` if the `new_priority` is the greater one. Otherwise, the `base_priority` is changed either to `new_priority` or `max(the max priority of each lock held by the thread)`, depending on which one is greater (This process takes donation into consideration).

3. Synchronization

We can disable interrupts when calling `thread_set_priority`, `lock_acquire` or other operations that interact with the `ready_list` in order to prevent access from other threads and thus maintain synchronization.

4. Rationale

I have considered two approaches for taking priority into accounts when scheduling. One is to sort the ready list before returning the next thread using `next_thread_to_run` , the other is to maintain the ready list as a priority queue and call `list_insert_ordered` to insert the thread respecting the order. I prefer the first approach based on the following reasons.

1. `list_insert_ordered` is called with `O(n)` time for each `thread_unblock`, `init_thread` and `thread_yield`, so this implementation could be inefficient if these operations are frequently invoked, e.g. when priorities of threads are changing frequently.
2. Maintaining a priority queue needs more coding and could be inconvenient to extend.

Therefore, I prefer my current choice.

Task 3: Advanced Scheduler

1. Data structures and functions

Global Variable

`fixed_t load_avg;`

- added to `thread.c`
- a moving average of the number of threads ready to run, recalculated once per second

Modified Structs

`int nice;`

- added to `struct thread`
- Every thread has a `nice` value between -20 and 20 directly under its control

`fixed_t recent_cpu;`

- added to `struct thread`
- `recent_cpu` measures the amount of CPU time a thread has received "recently"

Modified Functions

`static void init_thread (struct thread *t, const char *name, int priority)`

- initialize `nice` and `recent_cpu` to be `0` and `FP_CONST (0)`, respectively

`void thread_start (void)`

- initialize `load_avg` to be `FP_CONST (0)`

`void thread_set_nice (int nice)`

- set the current thread's `nice` value

`int thread_get_nice (void)`

- get the current thread's `nice` value

`int thread_get_load_avg (void)`

- get 100 times the system load average value

`int thread_get_recent_cpu (void)`

- get 100 times the current thread's `recent_cpu` value

Added Functions

`static void timer_interrupt (struct intr_frame *args UNUSED)`

- increase `recent_cpu` by 1 on every tick
- update `load_avg` and `recent_cpu` once per second using the provided formula
- update `priority` at every fourth tick using the provided formula

`void thread_mlfqs_increase_recent_cpu_by_one (void)`

- increase `recent_cpu` by 1

`void thread_mlfqs_update_load_avg_and_recent_cpu (void)`

- update `load_avg` and `recent_cpu` using the provided formula

`void thread_mlfqs_update_priority (struct thread *t)`

- update `priority` using the provided formula

2. Algorithms

Main Idea

The main idea is to implement the following formulas based on the floating point number operations provided in the `fixed-point.h`.

1. `priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)` - updated every fourth tick
2. `recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice` - updated every second
3. `load_avg = (59/60)*load_avg + (1/60)*ready_threads` - updated every second

3. Synchronization

I disable interrupts when accessing and computing threads' `priority` values to keep synchronized.

4. Rationale

Since we have the support of `fixed-point real numbers`, we can direct compute the required values based on the provided formulas. The advantage of this design is its concision and simplicity.

Additional Questions

1.

timer_ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread_to_run	load_avg
0	0.00	0.00	0.00	63	61	59	A	0.00
4	4.00	0.00	0.00	63	61	59	A	0.00
8	8.00	0.00	0.00	62	61	59	A	0.00
12	12.00	0.00	0.00	61	61	59	A	0.00
16	12.00	4.00	0.00	60	61	59	B	0.00
20	0.00	1.00	2.00	60	60	59	A	0.05
24	4.00	1.00	2.00	63	60	58	A	0.05
28	8.00	1.00	2.00	62	60	58	A	0.05
32	12.00	1.00	2.00	61	60	58	A	0.05
36	16.00	1.00	2.00	60	60	58	A	0.05

2.

Yes, there are ambiguities that can make the above values uncertain.

My rules are as follow:

- the initial value of `load_avg` is `0`
- the value of `ready_threads` is `3`
- the value of `TIMER_FREQ` is `20`
- the calculation order is `priority` -> `recent_cpu` -> `load_avg`