CS 130: Project 1 - Threads Design Document

Group Information

Please provide the details for your group below, including your group's name and the names and email addresses of all members.

- **Group Name**: *[printf("team")]*
- **Member 1**: Zirui Wang <wangzr2024@shanghaitech.edu.cn>
- **Member 2**: Xing Wu <wuxing2024@shanghaitech.edu.cn>

Preliminaries

> If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.

None.

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.

Pairing Heap

A data structure that maintains the heap property through constant-time merge operations.

```
/* Heap element. */
struct heap_elem
{
  struct heap_elem *child;   /* Child element. */
  struct heap_elem *sibling; /* Next sibling element. */
};

/* Compares the value of two heap elements A and B, given
   auxiliary data AUX.  Returns true if A is less than B, or
   false if A is greater than or equal to B. */
typedef bool heap_less_func (const struct heap_elem *lhs,
                             const struct heap_elem *rhs, void *aux);

/* Heap. */
struct heap
{
  size_t size;           /* current number of elements in the heap. */
  struct heap_elem *top; /* Heap top, which is the greatest element. */
```

```
  heap_less_func *less;  /* Comparison function. */
};
```

New Struct Member to `struct thread`

```
struct thread
{
  /* ... */

  /* Owned by devices/timer.c. */
  int64_t wakeup_tick;       /* Time when the thread stops sleeping. */
  struct heap_elem heapelem; /* Heap element for sleeping queue. */

  /* ... */
}
```

New Global Variable to `devices/timer.c`

```
/* A priority queue which contains sleeping threads. */
static struct heap sleep_que;
```

Algorithms

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

`timer_sleep()` adds the current thread to the `sleep_que`, which is a priority queue based on the pairing heap, and blocks the current thread.

`timer_interrupt()` updates the static variable `ticks`, and called function `thread_tick()` declared in `threads/thread.c` to update statistics (time spent idle, in kernel threads, or in user programs), then check whether a sleeping thread needs to be woken up. If true, wake those thread(s) up.

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

When a thread calls `timer_sleep()`, it is blocked immediately and is added to the sleeping queue, wating for an external wakeup. Then we use a data structure called Pairing Heap, whose time complexity to query the minimum item is $\mathcal{O}(1)$. Therefore, we can check all the sleeping threads efficiently.

Synchronization

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

Before operations, we called `intr_disable()` to disable interrupts. And restored the interrupts to the previous state when operations are done. Therefore, at one time there is only one thread accessing the sleep queue.

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

Same as A4, we disabled interrupts.

Rationale

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

- We have considered using a List to implement the sleeping queue. However, the Pairing Heap has advantage of time complexity.
- Compared with creating an array with fixed size or allocating memory dynamically, we use an approach similar to the list declared in `lib/kernel/list.h`, that is, embedding heap nodes directly within the thread struct.
    - Fixed-size Arrays: The reason we don't create a fixed-size array is that we suppose the maximum number of threads might be large after we finished demand paging in project 3.
    - Dynamic Allocation: We have two reasons not to allocate memory dynamically. The first is memory safety, and the second is to eliminate allocation/deallocation costs.

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.

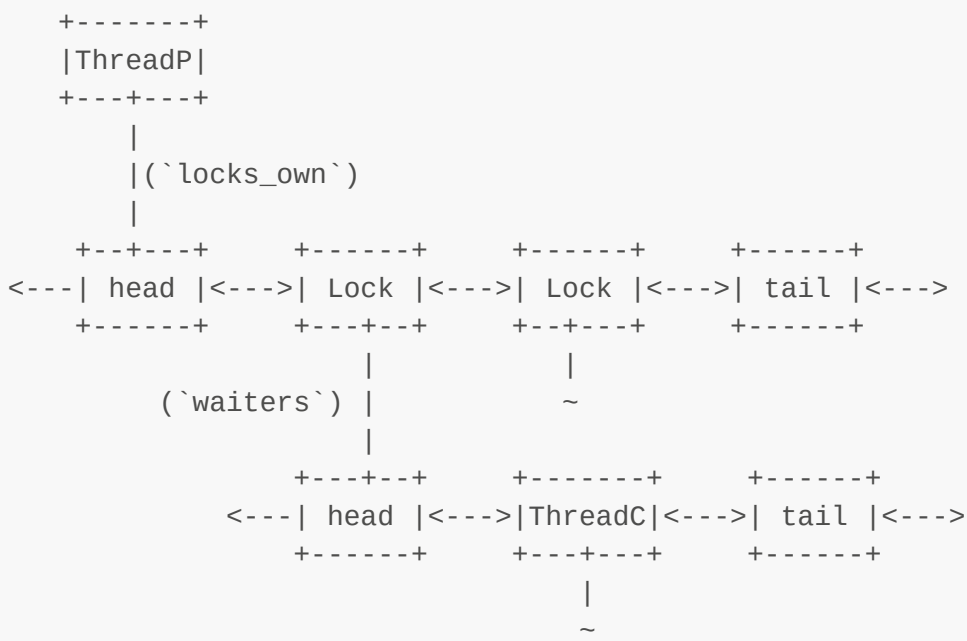
New Struct Member to `struct thread`

```
struct thread
{
  /* ... */

  /* Shared between thread.c and synch.c. */
   int extra_priority;    /* Extra priority by priority donation. */
   struct thread *parent; /* The thread that holds a lock waiting for. */
   struct list locks;     /* Locks this thread own. */

  /* ... */
}
```

New Struct Member to `struct lock`

```
struct lock
{
  /* ... */

  struct list_elem elem;      /* List elements used by struct thread. */
}
```

> **B2:** Explain the data structure used to track priority donation. Describe how priority donation works in a nested scenario using a detailed textual explanation.



A thread can only be blocked directly by one lock. If we define the holder of that lock as the parent of this thread, then the relationship of threads becomes several trees, whose roots are running or ready to run threads.

- When a thread is blocked, we climb up the tree until we reached the root, donating its priority along the way. Each jump climbs two levels up because we record the parent thread instead of lock.
- When a lock is released, we iterate its direct children to find the new donated priority. There is actually no need to always search the whole tree.

Algorithms

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

We iterate through the waiting list to find the thread with highest priority. The priority of each thread is maintained carefully.

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

When a thread is blocked, we climb up the tree until we reached the root, donating its priority along the way. Each jump climbs two levels up because we record the parent thread instead of lock.

> **B5:** Describe the sequence of events when `lock_release()` is called on a lock that a higher-priority thread is waiting for.

First we have to reset the parent-child relationship connected by that lock.

Then we iterate its direct children to find the new donated priority. There is actually no need to always search the whole tree because we have maintained a variable `extra_priority` which contains the maximum priority in a subtree rooted at that thread.

Finally, we called `sema_up()` to wake up the correct thread.

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?

It's a possible implementation of `thread_set_priority()`

```
void
thread_set_priority (int new_priority)
{
  struct thread *cur = thread_current ();
  cur->priority = new_priority;
  struct thread *max_waiting = thread_list_max (&ready_list);
  if (thread_less (cur, max_waiting))
    thread_yield ();
}
```

If a thread of higher priority is inserted into the ready list between the execution of `struct thread *max_waiting = thread_list_max (&ready_list)` and `if (thread_less (cur, max_waiting))`, that thread may not be scheduled correctly. Our implementation disabled interrupts to avoid this.

A lock is not suitable here. Suppose we used a lock `ready_list_lock` to ensure atomicity of accessing `ready_list`. When a thread gets preempted after acquiring that lock, then he will never be able to release that lock because the current thread is unable to make the preempted thread out of ready list. (A possible situation of deadlock.)

Rationale

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

We have considered making the ready threads in a priority queue based on heap. However, the priority of a thread may change after entering queue, thus breaking the structure of heap. We do have a solution to this problem. However, that leads to more memory usage, and we think memory is more valuable than time in pintos.

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.

Fixed Point

```
/* Fixed-Point Arithmetic. (17.14 format) */
typedef int32_t fp32_t;
```

New Struct Member to `struct thread`

```
struct thread
{
  /* ... */

  /* Owned by thread.c. */
  int nice;                   /* Nice value. */
  fp32_t recent_cpu;          /* Recent CPU. */

  /* ... */
}
```

New Global Variable in `threads/thread.c`

```
/* System load average, for mlfq. */
static fp32_t load_avg;
```

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:

In pintos, `src/devices/timer.h` has defined `TIMER_FREQ` to be 100. Therfore, we assume that it is greater than 36.

Timer Ticks	recent_cpu A	recent_cpu B	recent_cpu C	Priority A	Priority B	Priority 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	A
12	12	0	0	60	61	59	B
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A

Timer Ticks	recent_cpu A	recent_cpu B	recent_cpu C	Priority A	Priority B	Priority C	Thread to Run
24	16	8	0	59	59	59	A
28	20	8	0	58	59	59	C (FIFO)
32	20	8	4	58	59	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?

When two threads have the same priority, the next thread to run is uncertain. We use First-in-First-out rule to manage order. (Assuming that the threads are created in the order of A->B->C) This rule match our scheduler, since we select the one with maximum priority and appears the earliest in the list.

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

Incrementing `recent_cpu` by 1 needs to be done every tick, and recalculate `priority` needs to be done every fourth tick. Those two are put outside of interrupt context for efficiency. Updating `load_avg` and iterating through all threads for `recent_cpu` only happens once per second, thus we put them inside the 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?

We did not implement 64 queues. Instead, we reused the `ready_list` from Task 2. This approach reduces memory usage and maintains consistency with Task 2's implementation. However, it resulted in a time complexity of $\mathcal{O}(n)$ for finding the highest-priority thread, whereas using 64 queues would achieve $\mathcal{O}(1)$ time complexity.

Similar to Task 2, we considered using a priority queue based on a heap. However, frequent priority changes could break the heap structure, and the solution to maintain heap integrity might be too complex for pintos.

If we have extra time, we might choose to write more tests to check our design.

> **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 (i.e., 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?

We create a file `fixed.h` and wrap common fixed-point arithmetic operations in macros. It makes the code easier to read and avoids the cost of function calls. In addition, such lib makes it possible to expand 32-bit fixed point to 64-bit in the future.

Survey Questions

> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects? Any other comments?

None.