Advanced Operating Systems
Project 1: Threads
Design Document

The questions in this design document should reflect the design of the code you wrote for the project. **Your grade will reflect both the quality of your answer in this document and the quality of the design implementation in your code.** You may receive partial credit for answering questions for parts of the project that you did not get to implement, but you must indicate in your answer that there is no corresponding implementation, or you will not receive any credit.

For each question, **you should include both the name of the file(s), function name(s), and the line numbers where the relevant code may be found** – both the code that answers the question directly and any function that you refer to in your answer.

These design documents will be completed and submitted as a group. Please use this document as a guide for design and discuss the questions and their potential answers prior to beginning implementation. **All text in *italics* is intended to be replaced with your answer**. When you have completed your design document, submit it to the Canvas assignment Project 1 (Threads): Design.

Team Information Fill out the following information for each member of your team.

Name: *Yunseok Kang*
EID: *yk9784*
Email: *yunseok.kang@utexas.edu*
Unique Number: *51275*

Name: *Dohyun Kwon*
EID: *dk29493*
Email: *gundo0109@utexas.edu*
Unique Number: *51275*

Name: *Stefanus Adrian*
EID: *sja2673*
Email: *stefanus.adrian@utexas.edu*
Unique Number: *51275*

Preliminaries Please provide any preliminary comments and cite consulted sources.

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

Your answer here.

Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Your answer here.

ALARM CLOCK (10 points)

The 10 total points are divided equally among the following sub-questions.

A1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary for your implementation of alarm clock. Identify the purpose of each in 25 words or less.

1. **Global variable in timer.c:**
   - `struct list sleep_list; (line 20)` - Manage sleeping threads to allow the system to easily track and wake threads when their specified sleep time has elapsed.
   - `bool wake_up_tick_less(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED); (line 84)` - Compare two threads based on their wakeUpTick.
2. **In 'struct thread' (in thread.h):**
   - `uint64_t wakeUpTick; (line 110)` - A member variable declared in `struct thread` - serves as a marker for the time at which a sleeping or blocked thread is scheduled to be woken up or made ready again.

A2. Briefly describe what happens when a thread calls timer_sleep(), including the steps necessary to wake a thread (hint: timer_interrupt).

1. The current system tick count is retrieved.
2. The calling thread calculates its wake-up tick based on the current tick count and the specified sleep duration.
3. The thread is then added to the `sleep_list` with its wake-up time.
4. The thread is blocked and relinquishes control to the scheduler.
5. During each timer interrupt, the interrupt handler checks the `sleep_list` for any threads whose wake-up time has been reached or passed. If so, these threads are unblocked and removed from the list, making them eligible to run again.

Related functions and variables in timer.c:

- timer_sleep (line 92)
- timer_interrupt (line 161)

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

We maintain the `sleep_list`, in a sorted state to minimize the amount of time spent in the interrupt handler, achieving this by reducing the list searching time. Since it is sorted by `wakeUpTick` with threads with the soonest wake-up times at the front, we can terminate the list search once the `wakeUpTick` of an entry exceeds the current time tick.

Related functions and variables in timer.c:

- maintaining the `sleep_list`, in a sorted state: (line 105)
- terminating the list search once the `wakeUpTick` of an entry exceeds the current time tick: (line 166)

A4. How are race conditions avoided when multiple threads call timer_sleep() simultaneously? Describe the race conditions.

- Acquiring a lock before modifying the `sleep_list` and release the lock afterward. This ensures that no other thread can modify the list concurrently, preventing race conditions where multiple threads try to add themselves to the list simultaneously.

Describe Race Conditions : Multiple threads could concurrently attempt to add themselves to `sleep_list`, potentially corrupting the list or leading to lost updates if not properly synchronized.

Related functions and variables in timer.c:

- Acquiring a lock before modifying the sleep list: (line 100)

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

Race conditions during timer interrupts are avoided by:

- Acquiring a lock before manipulating the `sleep_list` within the timer interrupt handler. This prevents a scenario where a thread calls `timer_sleep()` and attempts to modify the list concurrently with the interrupt handler, ensuring atomicity of operations on the list.

Describe Race Conditions: A timer interrupt could occur while a thread is in the process of adding itself to the `sleep_list`, leading to potential corruption of the list or incorrect wake-up times.

Related functions and variables in timer.c:

- Acquiring a lock before manipulating the sleep list: (line 100)

A6. Why did you choose this design? In what ways is it superior to another design you considered? Be certain to compare between two working designs.

This design focuses on performance and simplicity:
We maintain the sleep list in a sorted state and have adopted one of the most common and effective methods to avoid race conditions: acquiring a lock. This design stands out for two main reasons. Firstly, the use of a sorted list minimizes the execution time of the frequently called `timer_interrupt`, making it more efficient than a mere linked list implementation for the Sleep-list. Secondly, by acquiring a lock to prevent race conditions, we not only simplify the implementation process but also address concerns related to prolonged interrupt disabling, which could lead to the operating system missing critical task timings.

PRIORITY SCHEDULING (10 points)

The 10 total points are divided equally among the following sub-questions.

B1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary for your implementation of priority scheduling and priority donation. Identify the purpose of each in 25 words or less.

1. **In 'struct thread' (in thread.h):**
   - `int base_priority; (line 99)` - Stores the original priority of the thread before any donation.
   - `struct list donations; (line 100)` - A list of donations made to the thread, used for priority donation.
   - `struct list_elem donation_elem; (line 101)` - Allows a thread to be part of another thread's donation list.
   - `struct lock *waiting_on_lock; (line 102)` - Points to the lock that the thread is currently waiting on, if any.
   - `int64_t sequence; (line 103)` - Sequence number for ordering threads with the same priority.
   - `bool thread_compare_priority(const struct list_elem *a, const struct list_elem *b, void *aux); (line 159)` - Compares two threads based on their priority.
2. **Global functions and variables in thread.c:**
   - `static int64_t next_thread_sequence; (line 256)` - Sequence number generator for threads. It is used to maintain the order of thread execution when multiple threads have the same priority.
   - `t->base_priority = priority; (line 524)` - Init the base priority with the given priority.
   - `list_init(&t->donations); (line 525)` - Init the list of priority donations.
   - `thread_unblock(struct thread *t); (line 259)` - Modified to insert threads into the ready list in an ordered fashion based on priority, ensuring that higher-priority threads are run before lower-priority ones.
   - `thread_yield(void); (line 334)` - Modified to ensure that if a thread yields the CPU, the next highest priority thread is chosen to run, maintaining priority scheduling.
   - `void thread_set_priority(int new_priority); (line 383)` - Modified to update current thread's priority, considering priority donations, and yields CPU if necessary to ensure scheduler respects priority order.
3. **Global functions and variables in synch.c:**
   - `donate_priority_recursive(struct thread *holder, struct lock *lock); (line 219)` - Recursively donates priority from a thread through a chain of locks to ensure priority inversion is minimized. Adjusts priorities across multiple lock holders.
   - `void donate_priority(struct lock *lock); (line 231)` - Function to donate priority to the lock holder.

- `void remove_donation(struct lock *lock); (line 252)` - Reverts the priority donation effects when the lock is released.

B2. How do you ensure that the highest priority thread waiting for a lock or semaphore wakes up first? Explain for both.

For both locks and semaphores, the highest priority thread is ensured to wake up first by:

- Semaphore: Using the sema_up and sema_down functions, the semaphore is managed, and the priority of waiting threads is checked to wake up the thread with the highest priority first.
- Condition Variables: Managing condition variables using the cond_signal and cond_wait functions ensures that waiting threads are awakened based on priority.
- Lock: Releasing locks with the lock_release function selects and wakes up the thread with the highest priority among the waiting threads.
- Donation Functions: Utilizing priority donation functions resolves priority inversion issues and adjusts the priority of threads waiting for locked resources.

Related functions and variables in synch.c:

- `sema_down (line 60)`
- `sema_up (line 122)`
- `lock_release (line 346)`
- `donate_priority_recursive (line 219)`
- `donate_priority (line 231)`
- `remove_donation (line 252)`
- `cond_wait (line 408)`
- `cond_signal (line 442)`
- `bool compare_sema (line 424)`

B3. Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?

The sequence:

- The current thread attempts to acquire a lock.
- If the lock is already held, the current thread's priority is donated to the lock holder if the current thread's priority is higher.
- This donation is recursively applied if the lock holder is waiting on another lock, ensuring that priority is donated through a chain of locks.

Related functions and variables in synch.c:

- `donate_priority (line 231)`
- `donate_priority_recursive (line 219)`

B4. Describe the sequence of events when lock_release() is called on a lock on which a higher-priority thread is waiting. What happens to the priority of the thread releasing the lock?

Upon lock_release():

- The lock holder removes the donation made by the waiting higher-priority thread.
- The lock holder's priority is recalculated, potentially reverting to its original priority if no higher priority donations remain.
- The lock is released, and the next highest priority thread waiting on the lock (if any) is unblocked.

Related functions and variables in synch.c:

- `remove_donation (line 252)`

B5. Describe a potential race in thread_set_priority() and explain how your implementation avoids it. Can you use a lock to avoid this race? Defend your answer.

A potential race could occur if multiple threads attempt to change a thread's priority simultaneously, especially if priority donation is involved. This implementation avoids such races by:

- Acquiring a lock before modifying a thread's priority or accessing/changing its donation list, ensuring atomicity of these operations.
- Utilizing a locking mechanism ensures that critical sections, which involve modifications to thread priorities and the donation list, are not disrupted by concurrent operations.

Related functions and variables in thread.c:

- `thread_set_priority (line 351)`

B6. Why did you choose this design for your priority scheduling and donation? In what ways is it superior to another design you considered? Please make certain you discuss both priority scheduling and donation, and be certain to compare against working designs.

This design focuses on simplicity and efficiency:

- Priority Scheduling: By maintaining the ready list in a sorted state based on priority and using sequence numbers for tie-breaking, the scheduler efficiently selects the highest priority thread.
- Priority Donation: Implementing priority donation with the donation linked-list ensures that priority inversions are minimized, and the system remains responsive even under complex lock contention scenarios.

Priority scheduling could be achieved by adopting a heap data structure for the ready list. Since adding or deleting an entry in the heap is accomplished in O(log n) time, compared to O(n) for a linked list, it could be more efficient, especially if there are frequent thread additions, deletions, or priority changes in the kernel. However, we opted for a sorted linked list, believing that maintaining a sorted ready list is more efficient for the scope and scale of PintOS. Additionally, it offers greater predictability and manageability, which are crucial for a lightweight OS.

When it comes to the ready list order, we believe that maintaining the ready list in an ordered state is more efficient. This approach is advantageous over searching the linked list with O(n) cost to select the highest priority thread each time a new thread needs to be scheduled. This efficiency is particularly important because the scheduling() function is invoked in various situations, including when threads yield, block, or exit.

Priority Donation could be achieved by managing multiple priority queues. When a thread requests a lock, it is added to the queue corresponding to its priority, and upon releasing the lock, servicing can begin from the highest priority queue. Compared to this approach, maintaining a sorted ready list conserves kernel memory and also reduces the overhead associated with queue management.

SURVEY QUESTIONS (optional) Answering these questions is optional, but it will help us improve the course in future semesters. 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 semester.

In your opinion, was this assignment, or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?

Your answer here.

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

Your answer here.

Is there some particular fact or hint we should give students in future semesters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

Your answer here.

Do you have any suggestions for the TAs to more effectively assist students, either for future semesters or the remaining projects?

Your answer here.

Any other comments?

Your answer here.