+--------------------+

| CS 451 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Hampton Terry [hterry@uw.edu](mailto:hterry@uw.edu)

Kevin Tang [kevin.tang2@gmail.com](mailto:kevin.tang2@gmail.com)

Matthew Dorsett [madman2@cs.washington.edu](mailto:madman2@cs.washington.edu)

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them 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.

CONTEXT SWITCHING

=================

>> C1: When "first yielder" finishes, which thread deletes its stack? Why?

------------------------------------------------------------------

Just before entering switch\_thread for the first time.

Thread: Main

#0 schedule () at ../../threads/thread.c:627

cur = 0xc000e000

next = 0xc0104000

prev = 0x0

\_\_func\_\_ = "schedule"

#1 0xc002102a in thread\_yield () at ../../threads/th

cur = 0xc000e000

old\_level = INTR\_ON

\_\_func\_\_ = "thread\_yield"

#2 0xc00206ce in main () at ../../threads/init.c:166

argv = 0xc00345e0

---------------------------------------------------------------------

Upon entering switch\_threads, registers are swapped out and switch\_threads

returns to switch\_entry.

switch\_entry then calls thread\_tail where the current\_thread() now is

first\_yielder.

Thread: first\_yielder

#0 thread\_schedule\_tail (prev=0xc000e000) at ../../threads/thread.c:580

#1 0xc00211da in switch\_entry () at ../../threads/switch.S:60

-----------------------------------------------------------------------

The kernel thread then runs the function first\_yielder leaves off with then

calls thread\_exit once this is complete. An important point is that when

thread\_exit is called, the currently running thread will have its status marked

as THREAD\_DYING.

This is the backtrace right before thread\_switch is called the second time.

Thread: first\_yielder

#0 schedule () at ../../threads/thread.c:627

#1 0xc002109c in thread\_exit () at ../../threads/thread.c:333

#2 0xc002110b in kernel\_thread (function=0xc0020190 <yielder>, aux=0x0)

at ../../threads/thread.c:479

#3 0x00000000 in ?? ()

--------------------------------------------------------------------------

In this backtrace, inside thread\_schedule\_tail, there is a check to see

if the previous thread is dying, and if it is, the page will be freed and in

thread\_exit it was already removed from the thread queue. The result is that

the first\_yielder is now completely gone.

Thread: second\_yielder

#0 thread\_schedule\_tail (prev=0xc0104000) at ../../threads/thread.c:601

#1 0xc0020faf in schedule () at ../../threads/thread.c:628

#2 0xc002102a in thread\_yield () at ../../threads/thread.c:352

#3 0xc0020198 in yielder () at ../../threads/init.c:42

#4 0xc0021106 in kernel\_thread (function=0xc0020190 <yielder>, aux=0x0)

at ../../threads/thread.c:478

#5 0x00000000 in ?? ()

>> C2: Give the backtrace of the kernel stack of a thread suspended

>> by a timer interrupt.

This is the backtrace of an interrupt when a thread has met or exceeded its

time slice from the time an interrupt has started. The interrupt handler is

setup, then run. Timer\_interrupt is run in this case because it is a timer

interrupt and calls thread\_tick. Inside thread\_tick, because the current

thread has exceeded its time slice, calls intr\_yield\_on\_return which sets yield\_on\_return that variable to be true.

#0 intr\_yield\_on\_return () at ../../threads/interrupt.c:232

#1 0xc0020c49 in thread\_tick () at ../../threads/thread.c:160

#2 0xc00237cb in timer\_interrupt (args=0xc000ef54)

at ../../devices/timer.c:206

#3 0xc00215f8 in intr\_handler (frame=0xc000ef54)

at ../../threads/interrupt.c:373

#4 0xc00217ff in intr\_entry () at ../../threads/intr-stubs.S:37

#5 0xc000ef54 in ?? ()

#6 0xc0020190 in init\_ram\_pages ()

#7 0x00000008 in ?? ()

#8 0xc0020165 in start () at ../../threads/start.S:180

The thread then continues for one more tick until it is interrupted again.

This time in the interrupt handler, because yield\_on\_return is true,

will call thread\_yield, which calls schedule, which then calls thread\_switch

to finalize the process of switching threads.

#0 schedule () at ../../threads/thread.c:627

#1 0xc0020f2a in thread\_yield () at ../../threads/thread.c:352

#2 0xc0021705 in intr\_handler (frame=0xc000ef54)

at ../../threads/interrupt.c:400

#3 0xc00217ff in intr\_entry () at ../../threads/intr-stubs.S:37

#4 0xc000ef54 in ?? ()

#5 0xc0020190 in init\_ram\_pages ()

#6 0x00000008 in ?? ()

#7 0xc0020165 in start () at ../../threads/start.S:180

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.

/\* timer.c \*/

static struct list sleep\_list;

Stores a list of sleeping threads by storing thread’s sleep\_elem field. Threads are added as they go to sleep, and removed as they wake.

/\* thread.h \*/

**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;** */\* Effective Priority. \*/*

**int original\_priority;**  */\* Original Priority \*/*

**int waiting\_priority;** */\* Priority thread wants to change to as soon as donating lock is released \*/*

**struct** list\_elem allelem; */\* List element for all threads list. \*/*

**struct thread \*blocker\_thread;** */\* If waiting for a lock, pointer to thread holding lock we need \*/*

**int64\_t wakeup;** */\* When timer should wake up \*/*

**struct list\_elem sleep\_elem;** */\* For the sleep list \*/*

*/\* Shared between thread.c and synch.c. \*/*

**struct** list\_elem elem; */\* List element. \*/*

**struct** list lock\_list; */\*list of locks this thread currently holds\*/*

**#ifdef USERPROG**

*/\* Owned by userprog/process.c. \*/*

**uint32\_t** **\***pagedir; */\* Page directory. \*/*

**#endif**

*/\* Owned by thread.c. \*/*

**unsigned** magic; */\* Detects stack overflow. \*/*

};

ADDED FIELDS:

* **struct list\_elem sleep\_elem:** An element for sleep\_list in timer.c.
* **int64\_t wakeup**: If the thread is sleeping, this field represents the number of ticks(since the OS has booted) the thread should be woken up.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

When timer\_sleep() is called, the calling thread’s sleep\_ticks field is set to the number of ticks the thread is to sleep + the current time, giving the wakeup time for the thread. The thread is then placed on the sleep\_list, which is a priority queue, backed by a sorted linked list, and then blocked. When timer interrupt is called, the wakeup time for the thread on the front of sleep\_list is compared to the current time. If the current time is greater than the thread’s wakeup time, the thread will unblock.

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

Using a priority queue minimizes the time spent in the interrupt handler because only the front of the sleep\_list needs to be examined to appropriately wake up threads which is especially helpful in the case when no thread should be woken up. This also allows us to remove the front of the list, so for each new thread that needs to be queued up and ran, we have an O(1) runtime.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer\_sleep() simultaneously?

The main issue that could occur when timer\_sleep is called by multiple threads simultaneously is at the point when the function accesses and tries to insert the thread into the sleep\_list. Through interrupts, it would be possible for instructions in list\_insert\_ordered to occur in an interleaved order. This has the potential to place threads in the wrong order or even lose threads. To prevent this, interrupts are turned off before inserting into the list, and when a thread is going to be blocked then immediately turned on right after. This takes care of synchronization for this shared object.

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer\_sleep()?

The timer interrupt can occur between reading how much time a thread is to sleep and the point when the thread is actually put to sleep. We want to be able to atomically find how long the thread is supposed to sleep and start sleeping immediately. If a timer reads to sleep for a certain amount of time, but then gets pre-empted for an arbitrary for number of ticks, those ticks will not be accounted for in sleep, so, it makes sense to include all of these between an interrupt disable and enable so that these can be done atomically to ensure the right number of sleep ticks.

---- RATIONALE ----

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

>> another design you considered?

We decided to go with storing the wakeup time in the threads struct and using a priority queue to order the threads by which thread will wake first. Other designs we considered were just placing the threads on an unordered list and iterating through and wake up threads as we notice that the thread’s sleep time would expire. This would make it necessary to do a traversal of the sleep list on every single interrupt instead of looking at the front of the list. This would increase the time we would spend in an interrupt context which would not be ideal. The tradeoff for this however, is that unlike a fifo ordered list where we would be able to insert in constant time, we must insert elements in sorted order which takes us O(N) time in the worst case. Perhaps a heap data structure or something similar could have been used to implement a priority queue which would allow us to insert items in O(log N) time, however, we chose to implement the priority queue with a linked list for simplicity’s sake.

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.

/\* thread.h \*/

**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;** */\* Effective Priority. \*/*

**int original\_priority;**  */\* Original Priority \*/*

**int waiting\_priority;** */\* Priority thread wants to change to as soon as donating lock is released \*/*

**struct** list\_elem allelem; */\* List element for all threads list. \*/*

**struct thread \*blocker\_thread;** */\* If waiting for a lock, pointer to thread holding lock we need \*/*

**int64\_t wakeup;** */\* When timer should wake up \*/*

**struct list\_elem sleep\_elem;** */\* For the sleep list \*/*

*/\* Shared between thread.c and synch.c. \*/*

**struct** list\_elem elem; */\* List element. \*/*

**struct** list lock\_list; */\*list of locks this thread currently holds\*/*

**#ifdef USERPROG**

*/\* Owned by userprog/process.c. \*/*

**uint32\_t** **\***pagedir; */\* Page directory. \*/*

**#endif**

*/\* Owned by thread.c. \*/*

**unsigned** magic; */\* Detects stack overflow. \*/*

};

ADDED FIELDS:

* **int priority**: The priority level the thread is currently running at
* **int original\_priority**: The default priority level for the thread. It will run at this priority unless something has changed it.
* **int waiting\_priority**: The priority level the thread is waiting to be changed to. This is used in the case where a thread is trying to change it’s priority, but it must wait because it holds a lock and is having a larger priority being donated to itself. Set to zero if not currently being used.
* **struct thread \*blocker\_thread**: If the thread is waiting for a lock, this field holds a pointer to the thread that currently holds the needed lock.
* **int64\_t wakeup**: If the thread is sleeping, this field represents the number of ticks(since the OS has booted) the thread should be woken up.
* **struct list lock\_list**: List of locks the thread currently holds.

/\* synch.h \*/

**struct** lock

{

**struct** **thread** **\***holder; */\* Thread holding lock (for debugging). \*/*

**struct** semaphore semaphore; */\* Binary semaphore controlling access. \*/*

**struct** list\_elem lock\_elem; */\* Reference to thread holding lock \*/*

};

ADDED FIELDS:

* **struct list\_elem lock\_elem**: Used to keep a reference to the thread the lock is being currently held by.

**struct** semaphore\_elem

{

**struct** list\_elem elem; */\* List element. \*/*

**struct** semaphore semaphore; */\* This semaphore. \*/*

**int** priority; */\* Priority \*/*

};

ADDED FIELDS:

* **int priority:** Used to store the priority the waiting thread.

>> B2: Explain the data structure used to track priority donation.

>> Use ASCII art to diagram a nested donation. (Alternately, submit a

>> .png file.)

Each struct thread has a field for it’s current priority, as well as a pointer to the thread who currently holds the lock the thread is trying to acquire. This field is set to NULL when it is not trying to acquire a lock or there is no one holding the lock it wants.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| **Thread 1**  |

--------------------------------------------------------

| int priority = 30; |

| struct thread \*blocker\_thread = thread 2; |

--------------------------------------------------------

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**| Thread 2**  |

--------------------------------------------------------

| int priority = 20; |

| struct thread \*blocker\_thread = thread 3; |

--------------------------------------------------------

Thread 1 will see that thread 2 has the lock it wants. It will call thread\_set\_priority\_donate(\*thread 2, 30). The function will see that thread 2’s current priority is 20, and will update it to the donated priority of 30.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**| Thread 2 (updated)** |

--------------------------------------------------------

| int priority = 30; |

| struct thread \*blocker\_thread = thread 3; |

--------------------------------------------------------

thread\_set\_priority will then check to see if thread 2 has a blocker thread. In this case, it will see that thread 3 has the lock thread 2 is waiting on, so it will recursively call thread\_set\_priority\_donated(\*thread 3, 30)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| **Thread 3**  |

--------------------------------------------------------

| int priority = 10; |

| struct thread \*blocker\_thread = NULL; |

--------------------------------------------------------

thread\_set\_priority\_donated will see that thread 3’s current priority is 10, which is less than the donated priority of 30, and will update it to the donated priority of 30.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| **Thread 3 (updated**) |

--------------------------------------------------------

| int priority = 30; |

| struct thread \*blocker\_thread = NULL; |

--------------------------------------------------------

At this point, thread\_set\_priority\_donated will see that thread 3’s blocker thread = NULL, which means it is not waiting on a lock from any other thread. With it’s new donated priority, it will be able to complete it’s task and release its lock. This will allow thread 2 to acquire it’s lock, complete it’s task, and release the lock it is currently holding. This will allow thread 1 to acquire the lock it is waiting for, and finish it’s process.

When the thread releases it’s lock, it’s priority is set back to it’s default value. The one exception is when multiple threads have donated the current thread priority. In this case, it will revert back to the highest donated priority.

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

For the synchronization constructs, a list of waiting threads is typically available. In the case of the ready queue, we are able to insert threads in order and just pull from the top making a priority queue, however, priority donation causes issues for the lists being kept by these constructs because they aren’t going to be reordered every time a thread changes priority. This is why instead of inserting threads ordered by priority, we insert threads in FIFO order to ensure a constant time insert, but then we sort the list before we queue up the top element. This ensures that we are always able to queue up the highest priority thread despite the priorities of the threads constantly changing because of donation.

>> B4: Describe the sequence of events when a call to lock\_acquire()

>> causes a priority donation. How is nested donation handled?

Suppose thread A is holding a lock that thread B is waiting for. If thread B has a high priority then thread A, thread A will never get scheduled to run. Since thread A does not have priority high enough to run, it will never be able to release the lock, therefore thread B will never be able to get to run. In order to work around this problem, thread B will donate it’s priority level to thread A so it will have a chance to be scheduled to run. This priority donation will occur whenever a thread is trying to acquire a lock from another thread with a lower priority than itself.

When a thread needs to donate it’s priority to another thread, it calls thread\_set\_priority\_donated(struct thread \*blocker\_t, int priority) where blocker\_t is the thread holding the lock that is receiving the increased priority, and int priority is the new priority level it can be increased to while it is holding the lock.

Nested donation is handled in this function by the recursive call to thread\_set\_priority\_donated() at the bottom of the function. It checks to see if the current\_thread is waiting on a lock, and if it is, checks to see the priority level of the thread that holds this lock. If it’s level is lower than the current thread’s priority level, it will donate it’s new priority level to the lock holder’s thread(which in turn will check to see if it is waiting on a lock and if it needs to donate it’s new priority level to another thread). One thing we need to worry about with this implementation is the stack overflowing when multiple recursive calls happen. However, the documentation mentioned we only need to support nested donation to up 8 threads deep, so we went forward with this implementation.

>> B5: Describe the sequence of events when lock\_release() is called

>> on a lock that a higher-priority thread is waiting for.

When lock\_release is called on a lock that a higher priority thread is waiting for, the lock’s state is updated and then goes into sema\_up. Inside sema\_up, the sema value will increase and then call unblock on the next highest priority thread waiting. Unblock will cause the current thread to immediately yield to the higher priority thread if the next thread’s priority is higher. If the next thread’s priority is equal (such as with donation) or less than the next thread’s priority, and return from sema\_up. Next, thread\_set\_priority undoes the effects of priority donation by setting the priority field to be equal to the original priority. This causes a yield to the donating thread right at the end of the donated thread’s call to lock\_release();

---- 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?

A potential race condition in thread\_set\_priority() would occur when a thread has called thread\_set\_priority(int priority) and is running through that function updating it’s own priority fields. The race condition could occur if it is trying to modify it’s own priority field at the same time another thread is trying to update the same field(possibly a thread donating priority to the current thread). Our implementation avoids this by turning off interrupts while it runs the lines of code that modify shared data, then turns them back on. This prevents the thread from being context\_switched out in the middle of writing data, leaving the thread in an unstable state.

A lock would not be a good solution to this problem. If you give each thread it’s own lock, and require each function to acquire the lock before it can access the threads data, you would run into the problem of threads being time sliced out in the middle of writing data to the thread struct. When a thread gets time slided out in the middle of writing data, any other thread that needs to access the current thread’s data will have to wait on the lock, possibly donate priority to the current thread, and wait for it to get scheduled again so it can finish. I would be better if you could stop the thread from being time sliced out in the first place, which you can do by disabling interrupts.

---- RATIONALE ----

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

>> another design you considered?

Implementing a priority scheduler with a priority queue made the most sense, as we did with ready list. However, to implement our linked list version of the priority queue correctly, we had to ensure that all the elements were sorted at all times. Doing this was especially tricky because of the many cases that involve thread’s changing their priorities, such as set\_thread\_priority, and priority donation. Because of this, the other design we considered was just having an unordered list and traversing the list every time we needed to grab an element. This design offers a simpler implementation because sorted order does not need to be maintained, at the cost of efficiency. Despite a priority queue being tougher to maintain, we felt it made the most logical sense to implement a priority scheduler in this way. In the case of priority scheduling with the synchronization constructs, we were not able to keep track and maintain order for all the waiting lists with all the changing priorities, and decided for that case it was easier to implement with the second method describe above where an unordered list is kept.

As with the priority queue we used for sleeping, we did not implement the most efficient priority queue, instead we just took advantage of the list that was already provided as well as the function list\_insert\_ordered. This caused or inserts to take linear time rather than logarithmic but for the sake of simplicity, this seemed like the best option.

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the

course in future quarters. 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 quarter.

>> In your opinion, was this assignment, or any one of the three problems

>> in it, too easy or too hard? Did it take too long or too little time?

This assignment took us a lot longer to do than we expected. The hardest part for us was getting the priority-donation tests to pass.

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

Working on the alarm clock and thread scheduler gave us a lot better understanding of how context switching works as well as the general flow of execution of threads running on our operation system.

>> Is there some particular fact or hint we should give students in

>> future quarters to help them solve the problems? Conversely, did you

>> find any of our guidance to be misleading?

Pointing students to the files/functions/structs they should be modifying for any given problem or type of problems would be useful. It takes awhile to get started when you are giving the entire PintOS source code and you have to dig through it all to find where to start.

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

Walking through the PintOS code in quiz section was really helpful. Possibly having the TA’s show us how to implement one of the tests in quiz section to see the general approach would help alot.

Clarifying the requirements for part A, with the backtracking would be helpful. There was some time spent being confused because we did not know what exactly was wanted in that section.l

>> Any other comments?

Cool project, very rewarding to get all the tests passing.