+---+

| Assignment 3 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+---+

---- GROUP ----

Rahul Aditya 18CS30032

Abhishek Srivastava 18CS10068

---- PRELIMINARIES ----

We added the line ‘outw(0xB004, 0x2000)’ to the shutdown_power_off(void) function in $HOME/pintos/src/devices/shutdown.c to handle the shutdown problem occurring due to new release of qemu. Without this our we encountered a TIMEOUT error for each process.

ALARM CLOCK

==========

---- DATA STRUCTURES ----

A. In $HOME/pintos/src/threads/threads.h
static struct list_elem sleeping_thread_element in **struct thread**
This is a list element to be inserted in the list of sleeping threads when the thread goes to sleep. This helps in efficiently maintaining the list of sleeping threads and accessing each thread quickly.

uint64_t sleep_to_ready_transition_time in **struct thread**
This saves the time after which the thread needs to be woken.

B. In $HOME/pintos/src/devices/timer.c

struct list list_of_sleeping_threads
This is a list of threads in THREAD_BLOCKED state. This stores not the entire thread, but only the ‘sleeping_thread_element’ of the threads that are in THREAD_BLOCKED state.

---- ALGORITHMS ----

A) Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.

- When timer_sleep() is called, we check whether the sleep time provided (int64_t ticks) is positive.
- If it is so, we get the current time in ‘start’.
- Then we check if the interrupt is on or not. If it is, we disable it using ‘intr_disable()’.
- Next we get the current thread using the ‘thread_current()’ function, and set its ‘sleep_to_ready_transition_time’ to ‘start + ticks’ to ensure it wakes up only after ‘ticks’ time from now.
- We insert this current thread’s ‘sleeping_thread_element’ in ‘list_of_sleeping_threads’ using the ‘list_insert_ordered()’, ordering them by their wake time using the sorting_comparator() function.
- Finally we block the thread using ‘thread_block()’ and re-enable interrupts.
- We have disabled interrupts so that timer interrupt handler and other asynchronous interrupts cannot affect timer_sleep().

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

- To reduce the amount of time spent in the interrupt handler we insert blocked threads in the ‘list_of_sleeping_threads’ in ascending order of wake time.
- While iterating over the list, if we encounter a thread which has wake time greater than current time, we are assured that all threads present after this will have wake time greater than current time (since ascending order). So we can stop the iteration. This ordering prevents the OS from iterating over all the threads.
- Otherwise, we simply set the sleep_to_ready_transition time to 0, and unblock the thread using the ‘thread_unblock()’ function.
- Finally we have to manually call the ‘list_remove()’ function to remove the current thread element from ‘list_of_sleeping_threads’ as we have woken it.

---- SYNCHRONIZATION ----

A) How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

To avoid race conditions when multiple threads call timer_sleep(), we simply disable interrupts before executing the block of code in the timer_sleep() function. This means no other process interrupts its execution and the entire block of code is executed without any context switching. Interrupts are enabled only at the end of the function, i.e., after the thread is put in THREAD_BLOCKED state.

B) How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?

All asynchronous interrupts are disabled at the beginning of the execution of timer_sleep() function. Hence, timer interrupts cannot occur during its execution, and thus, race conditions are avoided.

---- RATIONALE ----

A) Why did you choose this design? In what ways is it superior to another design you considered?

We chose this design mainly because it implements quick waking up of sleeping threads, and very nicely utilizes the data structures and relevant functions defined in the ‘list.h’ file. The ordering of sleeping threads in a list based on wake time helps us in waking up a thread as soon as its ‘wake time’ arrives. On average it takes constant time, but may go up to linear if there are multiple threads waking up at the same, which is rare.

We considered using a linked list without any ordering, so we could insert a sleeping thread in the linked list in constant time unlike linear time taken to insert it based on wake time. But while waking up threads, we would have to iterate over the entire linked list and this would take linear time which was bad. We believed that waking a thread should be fast so that it can be allotted the CPU quickly. And if we have to spend some time compulsory, we should do it while inserting a thread in the linked list, which is not as critical as waking a thread.

Another design we considered was using a static array of threads as a queue, and ordering threads based on their wake times just like before. In this case we found two problems

1. Insertion of a thread may take linear time.
2. The number of threads sleeping simultaneously will have an upper bound. If we keep this upper bound very high, we might end up wasting memory.

Another design we considered was using a priority queue. Insertion and deletion would have been fast, but the drawback we felt was its inability to make use of the excellent data structures and functions already described in the ‘list.h’ file.