4. Synchronization

4.1 Background

Proper synchronization is an important part of the solutions to these problems. Any synchronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there's no possibility for race conditions. Therefore, it's tempting to solve all synchronization problems this way, but **don't**. Instead, use **semaphores**, **locks**, and **condition variables** to solve the bulk of your synchronization problems. Read the tour section on synchronization or the comments in threads/synch.c if you're unsure what synchronization primitives may be used in what situations.

In the Pintos projects, the only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks. This means that data shared between kernel threads and an interrupt handler must be protected within a kernel thread by turning off interrupts.

This project only requires accessing a little bit of thread state from interrupt handlers. For the alarm clock, the timer interrupt needs to wake up sleeping threads. In the advanced scheduler, the timer interrupt needs to access a few global and per-thread variables. When you access these variables from kernel threads, you will need to disable interrupts to prevent the timer interrupt from interfering.

When you do turn off interrupts, take care to do so for the least amount of code possible, or you can end up losing important things such as timer ticks or input events. Turning off interrupts also increases the interrupt handling latency, which can make a machine feel sluggish if taken too far.

The synchronization primitives themselves in synch.c are implemented by disabling interrupts. You may need to increase the amount of code that runs with interrupts disabled here, but you should still try to keep it to a minimum.

Disabling interrupts can be useful for debugging, if you want to make sure that a section of code is not interrupted. You should remove debugging code before turning in your project. (Don't just comment it out, because that can make the code difficult to read.)

There should be no busy waiting in your submission. A tight loop that calls thread_yield() is one form of busy waiting.

4.2 Disabling Interrupts

If interrupts are off, no other thread will preempt the running thread, because thread preemption is driven by the timer interrupt. If interrupts are on, as they normally are, then the running thread may be preempted by another at any time, whether between two C statements or even within the execution of one.

Incidentally, this means that Pintos is a "preemptible kernel," that is, kernel threads can be preempted at any time. Traditional Unix systems are "nonpreemptible," that is, kernel threads can only be preempted at points where they explicitly call into the scheduler. (User programs can be preempted at any time in both models.) As you might imagine, preemptible kernels require more explicit synchronization.

You should have little need to set the interrupt state directly. Most of the time you should use the other synchronization primitives described in the following sections. The main reason to disable interrupts is to synchronize kernel threads with external interrupt handlers, which cannot sleep and thus cannot use most other forms of synchronization

Some external interrupts cannot be postponed, even by disabling interrupts. These interrupts, called non-maskable interrupts (NMIs), are supposed to be used only in emergencies, e.g. when the computer is on fire. Pintos does not handle non-maskable interrupts.

Types and functions for disabling and enabling interrupts are in threads/interrupt.h.

Type: **enum intr_level**

One of INTR*OFF or INTR*ON, denoting that interrupts are disabled or enabled, respectively.

Function: enum intr*level **intr*get_level (void)**

Returns the current interrupt state.

Function: enum intr*level **intr*set*level** (enum intr*level level)

Turns interrupts on or off according to level. Returns the previous interrupt state.

Function: enum intr*level **intr*enable** (void)

Turns interrupts on. Returns the previous interrupt state.

Function: enum intr*level **intr*disable** (void)

Turns interrupts off. Returns the previous interrupt state.

Interrupts might hard to understand for you. Because you should learn Interupt in the class of Principle of Computer Organization.

A easy way to use Interrupts:

eg:

enum intr*level old*level;

old*level = intr*disable();

-------- ADD YOUR CODE HERE --------

intr*set*level(old_level);

If you need to disable Interrupt, you can use it.

4.3 Semaphores

A semaphore is a nonnegative integer together with two operators that manipulate it atomically, which are:

- "Down" or "P": wait for the value to become positive, then decrement it.
- "Up" or "V": increment the value (and wake up one waiting thread, if any).

A semaphore initialized to **0** may be used to wait for an event that will happen exactly once. For example, suppose thread A starts another thread B and wants to **wait for B** to signal that some activity is complete. **A can create a semaphore** initialized to 0, **pass it to B** as it starts it, and then **A "down" the semaphore**. When B finishes its activity, it **"ups" the semaphore**. This works regardless of whether A "downs" the semaphore or B "ups" it first.

A semaphore initialized to 1 is typically used for controlling access to a resource. Before a block of code starts using the resource, it "downs" the semaphore, then after it is done with the resource it "ups" the resource. In such a case a lock, described below, may be more appropriate.

Semaphores can also be initialized to values larger than 1. These are rarely used.

Semaphores were invented by Edsger Dijkstra and first used in the THE operating system ([Dijkstra]).

Pintos' semaphore type and operations are declared in threads/synch.h.

Type: **struct semaphore** Represents a semaphore.

Function: void **sema_init** (struct semaphore *sema, unsigned value) Initializes sema as a new semaphore with the given initial value.

Function: void **sema_down** (struct semaphore *sema)

Executes the "down" or "P" operation on sema, waiting for its value to become positive and then decrementing it by one.

Function: bool **sema*try*down** (struct semaphore *sema) Tries to execute the "down" or "P" operation on sema, without waiting. Returns true if sema was successfully decremented, or false if it was already zero and thus could not be decremented without waiting. Calling this function in a tight loop wastes CPU time, so use sema_down() or find a different approach instead.

Function: void **sema_up** (struct semaphore *sema) Executes the "up" or "V" operation on sema, incrementing its value. If any threads are waiting on sema, wakes one of them up.

Unlike most synchronization primitives, sema_up() may be called inside an external interrupt handler.

Semaphores are internally built out of disabling interrupt (see section A.3.1 Disabling Interrupts) and thread blocking and unblocking (thread*block() and thread*unblock()). Each semaphore maintains a list of waiting threads, using the linked list implementation in lib/kernel/list.c.

4.4 Locks

A lock is like a semaphore with an initial value of 1. A lock's equivalent of "up" is called "release", and the "down" operation is called "acquire".

Compared to a semaphore, a lock has one added restriction: only the thread that acquires a lock, called the lock's "owner", is allowed to release it. If this restriction is a problem, it's a good sign that a semaphore should be used, instead of a lock.

Locks in Pintos are not "recursive," that is, **it is an error for the thread currently holding a lock to try to acquire that lock**.

Lock types and functions are declared in threads/synch.h.

Type: **struct lock** Represents a lock.

Function: void **lock_init** (struct lock *lock) Initializes lock as a new lock. The lock is not initially owned by any thread.

Function: void **lock_acquire** (struct lock *lock) Acquires lock for the current thread, first waiting for any current owner to release it if necessary.

Function: bool **lock*try*acquire** (struct lock *lock) Tries to acquire lock for use by the current thread, without waiting. Returns true if successful, false if the lock is already owned. Calling this function in a tight loop is a bad idea because it wastes CPU time, so use lock_acquire() instead.

Function: void **lock_release** (struct lock *lock) Releases lock, which the current thread must own.

Function: bool **lock*held*by*current*thread** (const struct lock *lock) Returns true if the running thread owns lock, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

4.5 Assignment

Description

Reimplement timer*sleep(), defined in 'devices/timer.c'. Although a working imple- mentation is provided, it "busy waits," that is, it spins in a loop checking the current time and calling thread*yield() until enough time has gone by. Reimplement it to avoid busy waiting.

void timer_sleep (int64 t ticks) [Function]

- Suspends execution of the calling thread until time has advanced by at least x timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly x ticks. Just put it on the ready queue after they have waited for the right amount of time.

timer_sleep() is useful for threads that operate in real-time, e.g. for blinking the cursor once per second. The argument to **timer_sleep()** is expressed in timer ticks, not in milliseconds or any another unit. There are **TIMER_FREQ** timer ticks per second, where **TIMER_FREQ** is a macro defined in *devices/timer.h*. The default value is 100. We don't recommend changing this value, because any change is likely to cause many of the tests to fail.

Separate functions **timer_msleep()**, **timer_usleep()**, and **timer_nsleep()** do exist for sleeping a specific number of milliseconds, microseconds, or nanoseconds, respectively, but these will call **timer_sleep()** automatically when necessary. You do not need to modify them.

Requirement

You should pass all tests with a prefix alarm-*. That is:

1. alarm-single
2. alarm-multiple
3. alarm-simultaneous
4. alarm-priority
5. alarm-zero
6. alarm-negative

Run `make check` in ***src/threads/*** to make sure your design can pass all these tests.

You should download and **fill** the [**DESIGNDOC**]((http://jeason.gitcafe.io/file/OSTA/Assignments/Assignment*4/DESIGNDOC) and archive your code(**the whole pintos folder**) and DESIGNDOC into a **zip** file **named sid*nameInPinYin*vid.zip(e.g *12330441*zhangsan_v0.zip)** and upload to the ftp.