#include "threads/synch.h"

#include <stdio.h>

#include <string.h>

#include "threads/interrupt.h"

#include "threads/thread.h"

void

sema\_init (struct semaphore \*sema, unsigned value)

{

ASSERT (sema != NULL);

sema->value = value;

list\_init (&sema->waiters);

}

void

sema\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

while (sema->value == 0)

{

list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem,

more\_prio, NULL);

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

bool

sema\_try\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

bool success;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (sema->value > 0)

{

sema->value--;

success = true;

}

else

success = false;

intr\_set\_level (old\_level);

return success;

}

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

bool yield = false;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (!list\_empty (&sema->waiters))

{

struct thread \*t = list\_entry (list\_pop\_front (&sema->waiters),

struct thread, elem);

thread\_unblock (t);

if (t->priority > thread\_current ()->priority)

yield = true;

}

sema->value++;

intr\_set\_level (old\_level);

if (yield)

{

if (!intr\_context ())

thread\_yield ();

else

intr\_yield\_on\_return ();

}

}

static void sema\_test\_helper (void \*sema\_);

void

sema\_self\_test (void)

{

struct semaphore sema[2];

int i;

printf ("Testing semaphores...");

sema\_init (&sema[0], 0);

sema\_init (&sema[1], 0);

thread\_create ("sema-test", PRI\_DEFAULT, sema\_test\_helper, &sema);

for (i = 0; i < 10; i++)

{

sema\_up (&sema[0]);

sema\_down (&sema[1]);

}

printf ("done.\n");

}

static void

sema\_test\_helper (void \*sema\_)

{

struct semaphore \*sema = sema\_;

int i;

for (i = 0; i < 10; i++)

{

sema\_down (&sema[0]);

sema\_up (&sema[1]);

}

}

void

lock\_init (struct lock \*lock)

{

ASSERT (lock != NULL);

lock->holder = NULL;

sema\_init (&lock->semaphore, 1);

}

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (!lock\_held\_by\_current\_thread (lock));

if (!lock\_try\_acquire (lock)) /\* Someone else holding the lock. \*/

{

sema\_down (&lock->semaphore);

lock->holder = thread\_current ();

}

}

bool

lock\_try\_acquire (struct lock \*lock)

{

bool success;

ASSERT (lock != NULL);

ASSERT (!lock\_held\_by\_current\_thread (lock));

success = sema\_try\_down (&lock->semaphore);

if (success)

lock->holder = thread\_current ();

return success;

}

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (lock\_held\_by\_current\_thread (lock));

lock->holder = NULL;

sema\_up (&lock->semaphore);

bool

lock\_held\_by\_current\_thread (const struct lock \*lock)

{

ASSERT (lock != NULL);

return lock->holder == thread\_current ();

}

struct semaphore\_elem

{

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

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

};

void

cond\_init (struct condition \*cond)

{

ASSERT (cond != NULL);

list\_init (&cond->waiters);

}

void

cond\_wait (struct condition \*cond, struct lock \*lock)

{

struct semaphore\_elem waiter;

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (lock\_held\_by\_current\_thread (lock));

sema\_init (&waiter.semaphore, 0);

list\_push\_back (&cond->waiters, &waiter.elem);

lock\_release (lock);

sema\_down (&waiter.semaphore);

lock\_acquire (lock);

}

void

cond\_signal (struct condition \*cond, struct lock \*lock UNUSED)

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (lock\_held\_by\_current\_thread (lock));

if (!list\_empty (&cond->waiters))

sema\_up (&list\_entry (list\_pop\_front (&cond->waiters),

struct semaphore\_elem, elem)->semaphore);

}

void

cond\_broadcast (struct condition \*cond, struct lock \*lock)

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

while (!list\_empty (&cond->waiters))

cond\_signal (cond, lock);

}