```c
#include <stdio.h>
#include <assert.h>
#include <sys/queue.h>
#include "thread.h"

/* Dirt simple thread package.  It doesn't deal with any of the hard issues
 * like blocking I/O and multiprocessors.  The scheduling policy is broken
 * too: it can starve a thread. The code also hasn't been tested
 * thoroughly, so use it your own risk; i haven't tried to run it on a sparc
 * processor for a while.
 *
 * Written by M.F. Kaashoek (kaashoek@mit.edu) */


/*
 *              Global thread variables
 */

struct thread {
    long        sp;             /* curent stack pointer */
    long        pc;             /* program counter */
    char        stack[STACK_SIZE]; /* the thread's stack */
    void        (*func)();      /* thread entry point */
    int         state;          /* running, runnable, waiting */
    int         njoin;
    int         joiner[MAX_THREAD];
#ifdef __sparc__
    int         dummy;                  /* make sp 8-byte aligned */
#endif
    LIST_ENTRY(thread) next;    /* List of threads */
};


struct mutex {
    int locked;
    LIST_HEAD(thread_list, thread) threads; /* blocked threads */
    thread_p last;
};

thread_p  current_thread;
thread_p  next_thread;

static thread_t all_thread[MAX_THREAD];

/*
 *                      Thread code
 */

void
thread_print(void)
{
    thread_p t;

    printf("current_thread: %d (0x%X)\n", current_thread - all_thread,
           current_thread);
    printf("next_thread: %d (0x%X)\n", next_thread - all_thread, next_thread);
    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        printf("%d: sp 0x%X pc 0x%X state 0x%X njoin %d\n", t - all_thread,
               t->sp, t->pc, t->state, t->njoin);
    }
}


static thread_p
thread_find(void)
{
    thread_p t;

    /* Look for a free thread. */
    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }

```



```c
    return t;
}


void
thread_init(void)
{
    current_thread = &all_thread[0];
    current_thread->state = RUNNING;
}


static void
thread_schedule(void)
{
    thread_p t;

    assert(next_thread == 0);

    /* Find another runnable thread. */
    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == RUNNABLE && t != current_thread) {
            next_thread = t;
            break;
        }
    }

    if (t >= all_thread + MAX_THREAD && current_thread->state == RUNNABLE) {
        /* The current thread is the only runnable thread; run it. */
        next_thread = current_thread;
    }

    if (next_thread == 0) {
        printf("thread_schedule: no runnable threads; deadlock\n");
        exit(1);
    }

    if (current_thread != next_thread) {        /* switch threads?  */
        next_thread->state = RUNNING;
        thread_switch();
    } else next_thread = 0;

    assert(next_thread == 0);
}

static void
thread_terminate(void)
{
    int i;

    current_thread->state = FREE;
    for (i = 0; i < current_thread->njoin; i++) {
        all_thread[current_thread->joiner[i]].state = RUNNABLE;
    }
    current_thread->njoin = 0;
    thread_schedule();
}


static void
thread_body()
{
    thread_p t;

    t = current_thread;
    (*(t->func))();
    thread_terminate();
}


void
thread_create(thread_p *thread, void (*func)())
{
```

```
    thread_p t;
    int *i;
#if __sparc__
    struct frame *sp;
#endif
    /* Found a free thread; init it. */

    t = thread_find();
    assert(t < all_thread + MAX_THREAD);

    *thread = t;
#if __sparc__
    sp = (struct frame *) (t->stack + SA(STACK_SIZE + BODY_OFFSET
                                          - WINDOWSIZE));
    sp->fr_savfp = (struct frame *) (t->stack + SA(STACK_SIZE +
                                          BODY_OFFSET));
    t->sp = (long) sp;
    t->pc = (long) thread_body - RETURN_OFFSET;
#endif
#if __i386__
    t->sp = (long) (t->stack + STACK_SIZE - BODY_OFFSET);
    i = (int *) (t->sp);
    t->pc = (long) thread_body;
    i[RETURN_OFFSET] = t->pc;
#endif
    t->state = RUNNABLE;
    t->func = func;
}


/*
 * The semantics of thread_join are akward.  If a thread joins too late
 * on another thread, that other thread might have terminated and its
 * thread struct might be used by new thread.
 */
void
thread_join(thread_p thread)
{
    if (thread->state != FREE) {
        thread->joiner[thread->njoin] = current_thread - all_thread;
        thread->njoin++;
        current_thread->state = WAITING;
        thread_schedule();
    }
}


void
thread_yield(void)
{
    current_thread->state = RUNNABLE;
    thread_schedule();
}

void
thread_wait(int *flag)
{
    while (!(*flag)) {
        current_thread->state = WAITING;
        thread_schedule();
    }
}

void
thread_notify(thread_p thread)
{
    if (thread->state != FREE)
        thread->state = RUNNABLE;
}


mutex_p
thread_newmutex(void)
```



```
{
    mutex_p m;

    m = (mutex_p) malloc(sizeof(*m));
    LIST_INIT(&m->threads);
    m->locked = 0;
    m->last = 0;
}


void
thread_acquire(mutex_p m)
{
    if (!m->locked) {
        m->locked = 1;
    } else {
        if (m->last) {
            LIST_INSERT_AFTER(m->last, current_thread, next);
        } else {
            LIST_INSERT_HEAD(&m->threads, current_thread, next);
        }
        m->last = current_thread;
        thread_wait(&m->locked);
    }
}

void
thread_release(mutex_p m)
{
    thread_p t;

    if (m->last) {
        t = m->threads.lh_first;
        LIST_REMOVE(t, next);
        if (t == m->last) m->last = 0;
        thread_notify(t);
    } else {
        m->locked = 0;
    }
}
```

```
#ifdef __sparc__
#include <sys/errno.h>
#include <sun4/trap.h>
#define KERNEL
#include <sun4/asm_linkage.h>
#undef KERNEL

#define t_sp    0

        .seg "text"

!       void thread_switch
!

        .globl _thread_switch
_thread_switch:
        set     _current_thread,%g1             ! Get current thread
        set     _next_thread,%g2                ! Get next thread
        save    %sp, -SA(WINDOWSIZE),%sp        ! Get new window
        ld      [%g1],%l0                       ! Load current thread
        ld      [%g2],%i0                       ! Load next thread
        ta      ST_FLUSH_WINDOWS                ! Flush windows current thread
        std     %fp,[%l0+t_sp]                  ! Save the current sp
        ldd     [%i0+t_sp],%fp                  ! Load new sp
        st      %i0,[%g1]                       ! Set current thread
        st      %g0,[%g2]                       ! Clear next thread
! Return into the new context.  Since all windows have been flushed, the
! RESTORE will cause a window underflow trap, restoring the registers of the
! new context from its stack.
        restore
        retl
        nop


#endif

#if __i386__

        .text

/* Switch from current_thread to next thread_thread, and make
 * next_thread the current_thread.  Use eax as a temporary register,
 * which should be caller saved. */
        .globl _thread_switch
_thread_switch:
        pushal                          /* save registers on stack */
        movl     _current_thread, %eax  /* load addr of current_thread  */
        movl    %esp, (%eax)            /* save esp reg in sp of thread */
        movl    _next_thread, %eax      /* load addr of next_thread */
        movl    (%eax), %esp            /* load sp of thread in esp reg */
        movl    %eax, _current_thread   /* next_thread -> current_thread */
        movl    $0, _next_thread        /* clear next_thread */
        popal                           /* restore registers from stack */
        ret                             /* pop return address from stack */
#endif
```

```
#ifndef __THREAD_H__

#define __THREAD_H__

/* Possible states of a thread; */
#define         FREE            0x0
#define         RUNNING         0x1
#define         RUNNABLE        0x2
#define         WAITING         0x4

#define         STACK_SIZE      32768
#define         BODY_OFFSET     200
#define         MAX_THREAD      4

#if __sparc__
#define         RETURN_OFFSET   8
#endif
#if __i386
#define         RETURN_OFFSET   8
#endif

typedef struct thread thread_t, *thread_p;
typedef struct mutex mutex_t, *mutex_p;

void thread_init(void);
void thread_create(thread_p *thread, void (*func)());
void thread_yield(void);
void thread_join(thread_p thread);
void thread_print(void);
void thread_wait(int *flag);
void thread_notify(thread_p thread);
void thread_acquire(mutex_p mutex);
void thread_release(mutex_p mutex);
mutex_p thread_newmutex(void);

#endif __THREAD_H__

```