```
#include <stdio.h>
#include <sys/queue.h>

/* 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 is written for an Intel x86.
 * It has not been tested thoroughly.
 *
 * Written by M.F. Kaashoek (kaashoek@mit.edu) */

/* 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

struct thread {
    long        sp;             /* current stack pointer */
    char        stack[STACK_SIZE]; /* the thread's stack */
    void        (*func)();      /* thread entry point */
    int         state;          /* running, runnable, waiting */
};

typedef struct thread thread_t, *thread_p;

thread_p  current_thread;
thread_p  next_thread;

static thread_t all_thread[MAX_THREAD];

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;
    }

    return t;
}

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

static void
thread_schedule(void)
{
    thread_p t;

    /* 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;
}

static void
thread_terminate(void)
{
    int i;

    current_thread->state = FREE;
    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;
    /* Found a free thread; init it. */

    t = thread_find();

    *thread = t;

    /*
     * Set up stack for return from thread_switch().
     * stack should have 8 words of registers as if saved by pushal,
     * then the return program counter.
     */
    t->sp = (long) (t->stack + STACK_SIZE - BODY_OFFSET);
    i = (int *) (t->sp);
    i[8] = (long) thread_body;

    t->state = RUNNABLE;
    t->func = func;
}

/*
 * Sleep until someone sets *flag = 1 and calls thread_notify().
 */
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;
}
```

```
1           .text
2
3   /* Switch from current_thread to next thread_thread, and make
4    * next_thread the current_thread.  Use eax as a temporary register,
5    * which should be caller saved. */
6           .globl thread_switch
7           .globl current_thread
8           .globl next_thread
9   thread_switch:
10          pushal                          /* save registers on stack */
11          movl     current_thread, %eax  /* load addr of current_thread  */
12          movl    %esp, (%eax)            /* save esp reg in sp of thread */
13          movl    next_thread, %eax      /* load addr of next_thread */
14          movl    (%eax), %esp            /* load sp of thread in esp reg */
15          movl    %eax, current_thread   /* next_thread -> current_thread */
16          movl    $0, next_thread        /* clear next_thread */
17          popal                           /* restore registers from stack */
18          ret                             /* pop return address from stack */
```