A user thread scheduler

Following the pthread API, the following is the interface of the user thread scheduler:

```
// th_t initialization, you should allocate stack space here
// the return value is 0 if succeeded.
int th_init(th_t *thread);
// the scheduler will be preemptive, it always returns 0.
int setpreemptive();
// the scheduler will be nonpreemptive, it always returns 0.
int setnonpreemptive();
// thread creation
// thread must be initialized by th_init() before th_fork() is used.
// returns 0 if succeeded.
int th_fork(th_t *thread, void *(*start_routine)(void*), void *arg);
// thread joining, wait for thread termination
// if status is not NULL, store the exit status of the thread into status,
// that is, the value given to th_exit(),
// or the return value of start_routine() if th_exit() is not called.
// returns 0 if succeeded.
int th_wait(th_t *thread, void *status);
// thread termination
void th_exit(int status);
// thread canceled by another thread
// returns 0 if succeeded.
int th_kill(th_t *thread);
// thread relinquish use of the processor
// and wait in the ready queue
void th_yield();
```

Scheduling

The scheduler is implemented in the **th_yield()** procedure. It is also called internally by the preemptive handler, **th_wait()** and **th_exit()**.

First of all, the state of the current thread is checked, if it is running, than it is put into the ready queue. If it is waiting, than the **th_yield()** must have been invoked by **th_wait()**, thus, the thread is put into the wait queue.

The scheduler uses a simple FCFS algorithm to select the next thread from the ready queue to execute. A higher priority process would get additional time slice to execute.

Once a thread is selected, the scheduler sets the current thread to the selected thread, and uses **swapcontext()** to transfer the control to the next thread.

Preemptive Scheduling

The preemptive scheduling is achieved by an interval timer and a signal handler. The **set_timer()** subroutine sets the time interval for the timer. When the interval passed, a **SIGVTALARM** signal is generated, and the handler set by **set_signal()** is executed. Because the timer only activates once, the timer must be reset before the next thread is executed.
Notice that when the internal data of the scheduler is being modified, the timer must be disabled to avoid the corruption of the data. This is accomplished by **entry_section()**, which disables the timer, and **exit_section()**, which resets the timer when leaving the scheduler.
The signal handler, **dispatcher()**, is fairly simple. It just calls **th_yield()**.

Thread Creation

A thread is initialized by **th_init()**, all data is set to the default values. But it is **th_fork()** that truly creates the thread and puts it into the ready queue. Also, **th_start()** is called to initialize the whole scheduler for the first use. If the scheduler is in preemptive mode, **th_fork()** also activates the timer.
The thread is created with a wrapper **th_caller()**, which calls the **start_routine()** stored by **th_fork()**, it also calls **th_exit()** with the return value if **th_exit()** is not called by the thread.

Waiting

When a thread calls **th_wait()** to wait for another thread, it first checks whether the thread has terminated, if it has, **th_wait()** retrieves the exit status directly from wstatus[] and returns (notice that this information may not be correct if too many threads have been created after the termination of the thread, since we implemented it using a circular array.)
If the waited thread has not terminated, marked the current thread as waiting for the waited thread and uses **th_yield()** to put the current thread into the wait queue, and the thread would keep waiting until the waited thread terminated. Finally, it retrieves the exit status and returns.

Termination

The destruction of a thread is done by **put_dead()**.
It checks whether there are threads waiting for the dead thread and put them into the ready queue. Also, it removes the destroyed thread from the wait queue or ready queue if **put_dead()** is called by **th_kill()**.
The thread information is not destroyed immediately, since the thread may still be running. (If a thread calls **th_exit()**, it would terminate itself.) Therefore, it is stored into the dead_thread variable. We must check whether there is a dead_thread already, if there is, the data of the thread is destroyed. (The thread cannot be the current thread.)
If the terminated thread is not the current thread, we can destroy it directly. Also, every time a new thread is selected from the ready queue, a check is made to determine whether there is a thread waiting to be destroyed.

Source Code

```
#include <assert.h>
#include <ucontext.h>
#include <sys/types.h>
#include <sys/time.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "mythread.h"
// ------ The Timer ------ //
const int QUANTUM = 100;
void set_timer(int t)
{
        unsigned long long qa = t*QUANTUM;
        struct itimerval it;
        it.it_interval.tv_sec = 0;
        it.it_interval.tv_usec = 0;
        it.it_value.tv_sec = qa / 1000000;
        it.it_value.tv_usec = qa % 1000000;
        assert(setitimer(ITIMER_VIRTUAL, &it, NULL) == 0);
}
void set_signal(void (*dispatcher)(int , siginfo_t *, void *))
{
        struct sigaction sa;
        sa.sa_sigaction = dispatcher;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART | SA_SIGINFO;
        assert(sigaction(SIGVTALRM, &sa, NULL) == 0);
}
// ------ The Threads Scheduler ------ //
#define MAX_THREAD 2000
#define MAX_QUEUE  2002
#define STACK_SIZE    10386
#define DEF_T         10
#define DEF_PR        0
enum {RUNNING, CREATED, READY, WAITING, TERMINATED};
typedef struct {
        char stack[STACK_SIZE];
        int status;
```

```
        int waitfor;
        void *(*func)(void*);
        void *param;
} th_data;
static th_t main_thread;
static th_data main_data;
static th_t *threads[MAX_THREAD];
static int wstatus[MAX_THREAD];
static int dead_thread = -1;
static int th_it = 0;
int curr = 0;
// internal control
void dispatcher(int t, siginfo_t *si, void *old_context) { th_yield(); }
static int th_initialized = 0;
void th_start()
{
        if (th_initialized) return;
        main_thread.u = &main_data;
        threads[0] = &main_thread;
        th_it = 1;
        set_signal(dispatcher);
        th_initialized = 1;
}
void th_start_timer()
{
        int T = DEF_T;
        if (threads[curr]->time_slice > 0)
                T = threads[curr]->time_slice;
        if (threads[curr]->priority > 0)
                T += threads[curr]->priority;
        set_timer(T);
}
static int th_preemptive = 0;
static int inside = 0;
void entry_section()
{
        if (!inside) {
                set_timer(0);
                inside = 1;
        }
}
void exit_section()
```

```
{
        if (inside) {
                inside = 0;
                if (th_preemptive) th_start_timer();
        }
}
static inline void th_swap(th_t *prev, th_t *next)
{
        swapcontext(&prev->uct, &next->uct);
}
static inline int get_tid(th_t *thread)
{
        if (thread && thread->state != TERMINATED && threads[thread->tid])
                return thread->tid;
        return -1;
}
static inline void th_next()
{
        while (threads[(++th_it) % MAX_THREAD]);
}
// process queue
static int ready_queue[MAX_QUEUE], wait_queue[MAX_QUEUE];
static int rq_begin, rq_end, wq_begin, wq_end;
static int rq_front() { return ready_queue[rq_begin]; }
static int rq_pop() { return rq_begin = (rq_begin + 1) % MAX_QUEUE; }
static void rq_push(int tid)
{
        ready_queue[rq_end] = tid;
        rq_end = (rq_end + 1) % MAX_QUEUE;
        threads[tid]->state = READY;
}
static int wq_front() { return wait_queue[wq_begin]; }
static int wq_pop() { return wq_begin = (wq_begin + 1) % MAX_QUEUE; }
static void wq_push(int tid)
{
        wait_queue[wq_end] = tid;
        wq_end = (wq_end + 1) % MAX_QUEUE;
        threads[tid]->state = WAITING;
}
static void recycle_dead()
{
        if (dead_thread != -1 && dead_thread != curr) {
```

```
                if (dead_thread != 0 && threads[dead_thread]->u) {
                        free(threads[dead_thread]->u);
                }
                threads[dead_thread]->u = 0;
                threads[dead_thread] = 0;
                dead_thread = -1;
        }
}
static void put_dead(int tid)
{
        th_data *dtd = threads[tid]->u;
        wstatus[tid] = dtd->status;
        // check wait queue that waits for tid
        int i;
        for (i=wq_begin; i!=wq_end; i=(i+1)%MAX_QUEUE) {
                int ctid = wait_queue[i];
                th_data *td = threads[ctid]->u;
                if (td->waitfor == tid) {
                        wait_queue[i] = wq_front();
                        wq_pop();
                        rq_push(ctid);
                }
        }
        // check ready queue if tid in it
        if (threads[tid]->state == READY)
                for (i=rq_begin; i!=rq_end; i=(i+1)%MAX_QUEUE) {
                        if (ready_queue[i] == tid) {
                                rq_end = (rq_end+MAX_QUEUE-1)%MAX_QUEUE;
                                if (rq_end != i) {
                                        ready_queue[i] = ready_queue[rq_end];
                                        break;
                                }
                        }
                }
        // check wait queue if tid in it
        if (threads[tid]->state == WAITING)
                for (i=wq_begin; i!=wq_end; i=(i+1)%MAX_QUEUE) {
                        if (wait_queue[i] == tid) {
                                wq_end = (wq_end+MAX_QUEUE-1)%MAX_QUEUE;
                                if (wq_end != i) {
                                        wait_queue[i] = wait_queue[rq_end];
                                        break;
```

```
                                        }
                                }
                        }
                recycle_dead();
                dead_thread = tid;
                threads[tid]->state = TERMINATED;
                recycle_dead();
}
int th_init(th_t *thread)
{
                th_data *data = thread->u = malloc(sizeof(th_data));
                if (data == 0) return -1;
                data->status = 0;
                thread->state = CREATED;
                thread->priority = DEF_PR;
                thread->time_slice = DEF_T;
                ucontext_t *uct = &thread->uct;
                getcontext(uct);
                uct->uc_stack.ss_sp = &data->stack;
                uct->uc_stack.ss_size = STACK_SIZE;
                uct->uc_stack.ss_flags = 0;
                sigemptyset(&uct->uc_sigmask);

                return 0;
};
int setpreemptive()
{
                th_preemptive = 1;
                return 0;
}
int setnonpreemptive()
{
                set_timer(0);
                th_preemptive = 0;
                return 0;
}
void th_caller()
{
                th_data *td = threads[curr]->u;
                int *ret = NULL;
                ret = td->func(td->param);
                if (ret && threads[curr]->state != TERMINATED)
```

```
                th_exit(*ret);
        else
                th_exit(0);
}
// thread creation
int th_fork(th_t *thread, void *(*start_routine)(void*), void *arg)
{
        if (thread == 0 || start_routine == 0) return -1;
        entry_section();
        th_start();
        th_data *td = thread->u;
        td->param = arg;
        td->func = start_routine;
        thread->tid = th_it;
        threads[th_it] = thread;
        makecontext(&thread->uct, th_caller, 0);
        rq_push(th_it);
        th_next();
        exit_section();
        return 0;
}
// thread joining, wait for thread termination
int th_wait(th_t *thread, void *status)
{
        if (thread && thread->state == TERMINATED) {
                if (status) *((int * )status) = wstatus[thread->tid];
                return 0;
        }
        int tid = get_tid(thread);
        if (tid == -1) return -1;
        entry_section();
        th_data *td = threads[curr]->u;
        td->waitfor = tid;
        threads[curr]->state = WAITING;
        th_yield();
        if (status) *((int * )status) = wstatus[tid];
        return 0;
}
// thread termination
void th_exit(int status)
{
        entry_section();
```

```
        th_data *td = threads[curr]->u;
        td->status = status;
        put_dead(curr);
        th_yield();
}
// thread canceled by another thread
int th_kill(th_t *thread)
{
        int tid = get_tid(thread);
        if (tid == -1) return -1;
        entry_section();
        th_data *td = thread->u;
        td->status = 1;
        put_dead(tid);
        exit_section();
        return 0;
}
// thread relinquish use of the processor
// and wait in the ready queuee
void th_yield()
{
        if (rq_begin == rq_end) return;
        entry_section();
        if (threads[curr]->state == RUNNING)
                rq_push(curr);
        else if (threads[curr]->state == WAITING)
                wq_push(curr);
        // transfer the control to the next thread
        if (rq_begin != rq_end) {
                int old = curr;
                curr = rq_front();
                rq_pop();
                threads[curr]->state = RUNNING;
                exit_section();
                th_swap(threads[old], threads[curr]);
                entry_section();
                recycle_dead();
        }
        exit_section();
}
```