Kelvin

OSY Home 6

Class lecturer: OLI10

Deadline: 12.12.2025, 0:00 (expired)

Submit 1 / 1 - Adam Vlček (VLC028), 12.12.2025 01:33, **1 hour, 33 minutes after deadline**

Select multiple files using CTRL or SHIFT or **drag them to this window**

Assignment | Source code | Upload

- gthr.c
- gthr.h
- gtswtch.S
- main.c

gthr.c

```
1
2 #include <assert.h>
3 #include <stdbool.h>
4 #include <stdint.h>
5 #include <stdio.h>
6 #include <string.h>
7 #include <stdlib.h>
8 #include <signal.h>
9+#include <unistd.h>
10 #include <errno.h>
11 #include <sys/time.h>
12
13 #include "gthr.h"
14
15 struct gt_context_t g_gttbl[ MaxGThreads ];
16 struct gt_context_t * g_gtcur;
17 uint64_t g_start_time_ms = 0;
18
19 // --- get current time in milliseconds ---
20 static uint64_t gt_get_time_ms( void ) {
21     struct timeval l_tv;
22     gettimeofday( &l_tv, NULL );
23
24     return (uint64_t)l_tv.tv_sec * 1000 + l_
25 }
26
27 #if ( GT_PREEMPTIVE != 0 )
28
29 volatile int g_gt_yield_status = 1;
30
31 void gt_sig_handle( int t_sig );
32
33 // initialize and start SIGALRM
34 void gt_sig_start( void )
35 {
36     struct sigaction l_sig_act;
37     memset( &l_sig_act, 0, sizeof( l_sig_act
38     l_sig_act.sa_handler = gt_sig_handle;
39
40     sigaction( SIGALRM, &l_sig_act, NULL );
41
42     struct itimerval l_tv_alarm = { { 0, Ti
```

```
    setitimer( ITIMER_REAL, & l_tv_alarm, NU
}

// deinitialize and stop SIGALRM
void gt_sig_stop( void )
{
    struct itimerval l_tv_alarm = { { 0, 0 }
    setitimer( ITIMER_REAL, & l_tv_alarm, NU

    struct sigaction l_sig_act;
    memset( &l_sig_act, 0, sizeof( l_sig_act
    l_sig_act.sa_handler = SIG_DFL;

    sigaction( SIGALRM, &l_sig_act, NULL );
}

// unblock SIGALRM
void gt_sig_mask_reset( void )
{
    sigset_t l_set;                     // (
    sigemptyset( & l_set );             // (
    sigaddset( & l_set, SIGALRM );      // S

    sigprocmask( SIG_UNBLOCK, & l_set, NULL
}

// function triggered periodically by timer
void gt_sig_handle( int t_sig )
{
    gt_sig_mask_reset();                // e

    // --- manage timers - check all blocke
    uint64_t l_now = gt_get_time_ms();
    for ( struct gt_context_t * p = &g_gttbl
        if ( p->thread_state == Blocked && p
            p->delay_until = 0;
            p->thread_state = Ready;
        }
    }

    g_gt_yield_status = gt_yield();     // s
}

#endif

// initialize first thread as current contex
void gt_init( void )
{
    // --- clear all thread contexts ------
    memset( g_gttbl, 0, sizeof( g_gttbl ) )
    // ------------------------------------

    g_gtcur = & g_gttbl[ 0 ];           // i
    g_gtcur -> thread_state = Running;  // s

    // --- set name of initial thread -----
    strncpy( g_gtcur->name, "main", MaxTaskN
    g_gtcur->delay_until = 0;

    g_start_time_ms = gt_get_time_ms(); // r
}


// exit thread
void gt_stop( void )
```

```
{
    if ( g_gtcur != & g_gttbl[ 0 ] )    // 
    {
        g_gtcur -> thread_state = Unused;//
        gt_yield();                         // 
        assert( !"reachable" );             // 
    }
}

void gt_start_scheduler( void )
{
    // --- start time measurement ----------
    g_start_time_ms = gt_get_time_ms(); // 
    // ------------------------------------

#if ( GT_PREEMPTIVE != 0 )
    gt_sig_start();                     // 
    while ( g_gt_yield_status )
    {
        usleep( TimePeriod * 1000 + 1000 );
    }
    gt_sig_stop();
#else
    while ( gt_yield() ) {}             // 
#endif
}


// switch from one thread to other
int gt_yield( void )
{
    struct gt_context_t * p;
    struct gt_regs * l_old, * l_new;
    int l_no_ready = 0;                 // 

    // --- check timers and unblock tasks i
#if ( GT_PREEMPTIVE == 0 )
    // In cooperative mode, check timers he
    uint64_t l_now = gt_get_time_ms();
    for ( struct gt_context_t * t = &g_gttb
        if ( t->thread_state == Blocked && 
            t->delay_until = 0;
            t->thread_state = Ready;
        }
    }
#endif
    // ------------------------------------

    p = g_gtcur;
    while ( p -> thread_state != Ready )// 
    {
        if ( p -> thread_state == Blocked |
            l_no_ready++;               // 

        if ( ++p == & g_gttbl[ MaxGThreads 
            p = & g_gttbl[ 0 ];
        if ( p == g_gtcur )             // 
        {
            return - l_no_ready;        // 
        }
    }

    if ( g_gtcur -> thread_state == Running
        g_gtcur -> thread_state = Ready;
    p -> thread_state = Running;
```

```
173     l_old = & g_gtcur -> regs;          // p
174     l_new = & p -> regs;                // a
175     g_gtcur = p;                        // s
176 #if ( GT_PREEMPTIVE != 0 )
177     gt_pree_swtch( l_old, l_new );      // p
178 #else
179     gt_swtch( l_old, l_new );           // p
180 #endif
181     return 1;
182 }
183
184
185 // create new thread by providing pointer to
186 int gt_go( void ( * t_run )( void ), const c
187 {
188     char * l_stack;
189     struct gt_context_t * p;
190
191     for ( p = & g_gttbl[ 0 ];; p++ )
192         if ( p == & g_gttbl[ MaxGThreads ] )
193             return -1;
194         else if ( p -> thread_state == Unuse
195             break;
196
197     l_stack = ( char * ) malloc( StackSize )
198     if ( !l_stack )
199         return -1;
200
201     *( uint64_t * ) & l_stack[ StackSize - 8
202     *( uint64_t * ) & l_stack[ StackSize - 1
203     p -> regs.rsp = ( uint64_t ) & l_stack[
204     p -> thread_state = Ready;
205     p -> delay_until = 0;
206
207     // set task name
208     if ( t_name )
209         strncpy( p->name, t_name, MaxTaskNam
210     else
211         snprintf( p->name, MaxTaskName, "tas
212     p->name[ MaxTaskName - 1 ] = '\0';
213
214     // return handle if requested
215     if ( t_handle )
216         *t_handle = p;
217
218     return 0;
219 }
220
221
222 // --- suspend a task (NULL = current task,
223 void gt_suspend( gt_handle_t t_handle ) {
224     struct gt_context_t * p = t_handle ? t_h
225
226     if ( p->thread_state == Running || p->th
227         p->thread_state = Suspended;
228         p->delay_until = 0;  // cancel any p
229
230         // if suspending current task, yield
231         if ( p == g_gtcur )
232             gt_yield();
233     }
234 }
235
236
237 // --- resume a suspended task (like FreeRTO
```

```
238 void gt_resume( gt_handle_t t_handle ) {
239     if ( t_handle && t_handle->thread_state
240         t_handle->thread_state = Ready;
241     }
242 }
243
244
245 // --- delay current task for specified mil
246 void gt_delay( uint32_t t_ms ) {
247     if ( g_gtcur == &g_gttbl[0] )
248         return;  // don't block main/schedu
249
250     g_gtcur->delay_until = gt_get_time_ms()
251     g_gtcur->thread_state = Blocked;
252     gt_yield();
253 }
254
255
256 // --- print list of all tasks (like FreeRT
257 void gt_task_list( void ) {
258     static const char * l_state_names[] = {
259
260     printf( "\n%-16s %-10s %s\n", "task name
261     printf( "----------------------------
262
263     for ( int i = 0; i < MaxGThreads; i++ )
264         struct gt_context_t * p = &g_gttbl[
265         if ( p->thread_state != Unused ) {
266             printf( "%-16s %-10s %d\n",
267                     p->name,
268                     l_state_names[ p->threa
269                     i );
270         }
271     }
272     printf( "\n" );
273 }
274
275
276 int uninterruptibleNanoSleep( time_t t_sec,
277 {
278 #if 0
279     struct timeval l_tv_cur, l_tv_limit;
280     struct timeval l_tv_tmp = { t_sec, t_na
281     gettimeofday( &l_tv_cur, NULL );
282     timeradd( &l_tv_cur, &l_tv_tmp, &l_tv_l
283     while ( timercmp( &l_tv_cur, &l_tv_limi
284     {
285         timersub( &l_tv_limit, &l_tv_cur, &
286         struct timespec l_tout = { l_tv_tmp
287         if ( nanosleep( & l_tout, &l_res ) <
288         {
289             if ( errno != EINTR )
290                 return -1;
291         }
292         else
293         {
294             break;
295         }
296         gettimeofday( &l_tv_cur, NULL );
297     }
298
299 #else
300     struct timespec req;
301     req.tv_sec = t_sec;
302     req.tv_nsec = t_nanosec;
```

```
303
304     do {
305         if (0 != nanosleep( & req, & req)) 
306         if (errno != EINTR)
307             return -1;
308         } else {
309             break;
310         }
311     } while (req.tv_sec > 0 || req.tv_nsec 
312 #endif
313     return 0; /* Return success */
314 }
315
316
```