```
#include "mthread_internal.h"
#include <sched.h>


#ifdef TWO_LEVEL
#include <pthread.h>
#endif

#ifdef TWO_LEVEL
#define MTHREAD_LWP 4
#else
#define MTHREAD_LWP 1
#endif

#define MTHREAD_DEFAULT_STACK 128*1024 /*128 kO*/
#define MTHREAD_MAX_VIRUTAL_PROCESSORS 256

static mthread_virtual_processor_t virtual_processors[MTHREAD_MAX_VIRUTAL_PROCES
SORS];
static mthread_list_t joined_list;

#define MTHREAD_LIST_INIT {NULL,NULL,0}

static inline void mthread_list_init(mthread_list_t* list){
  mthread_list_t INIT=MTHREAD_LIST_INIT;
  *list = INIT;
}

static inline void mthread_init_thread(struct mthread_s* thread){
  thread->next = NULL;
  thread->status = RUNNING;
  thread->res = NULL;
}

void mthread_insert_first(struct mthread_s* item, mthread_list_t* list){
  mthread_spinlock_lock(&(list->lock));
  if(list->first == NULL){
    item ->next = NULL;
    list->first = item;
    list->last = item;
  } else {
    item ->next = list->first;
    list->first = item;
  }
  mthread_spinlock_unlock(&(list->lock));
}

void mthread_insert_last(struct mthread_s* item, mthread_list_t* list){
  mthread_spinlock_lock(&(list->lock));
  if(list->first == NULL){
    item ->next = NULL;
    list->first = item;
    list->last = item;
  } else {
    item ->next = NULL;
    list->last->next = item;
    list->last = item;
  }
  mthread_spinlock_unlock(&(list->lock));
}

struct mthread_s* mthread_remove_first(mthread_list_t* list){
```



```
  struct mthread_s* res = NULL;
  mthread_spinlock_lock(&(list->lock));
  if(list->first != NULL){
    res = (struct mthread_s*)list->first;
    list->first = res->next;
    if(list->first == NULL){
      list->last = NULL;
    }
  }
  mthread_spinlock_unlock(&(list->lock));
  return res;
}

static inline
int
mthread_mctx_set (struct mthread_s * mctx,
                  void (*func) (void *), char *stack, size_t size,
                  void *arg)
{
  /* fetch current context */
  if (getcontext (&(mctx->uc)) != 0)
    return 1;

  /* remove parent link */
  mctx->uc.uc_link = NULL;

  /* configure new stack */
  mctx->uc.uc_stack.ss_sp = stack;
  mctx->uc.uc_stack.ss_size = size;
  mctx->uc.uc_stack.ss_flags = 0;

  mctx->stack = stack;

  /* configure startup function (with one argument) */
  makecontext (&(mctx->uc), (void (*)(void)) func, 1 + 1, arg);

  return 0;
}

static inline int
mthread_mctx_swap (struct mthread_s * cur_mctx, struct mthread_s * new_mctx){
  swapcontext(&(cur_mctx->uc) ,&(new_mctx->uc));
  return 0;
}

static struct mthread_s* mthread_work_take(mthread_virtual_processor_t* vp){
  int i;
  struct mthread_s* tmp = NULL;
  for(i = 0; i < MTHREAD_LWP; i++){
    tmp = NULL;
    if(vp != &(virtual_processors[i])){
      if(virtual_processors[i].ready_list.first != NULL){
        tmp = mthread_remove_first(&(virtual_processors[i].ready_list));
      }
    }
    if(tmp != NULL){
      mthread_log("LOAD BALANCE","Work %p from %d to %d\n",tmp,i,vp->rank);
      return tmp;
    }
  }
  sched_yield();
  return tmp;
```

```
}

void __mthread_yield(mthread_virtual_processor_t* vp){
  struct mthread_s* next;
  struct mthread_s* current;

  current = (struct mthread_s*)vp->current;
  next = mthread_remove_first(&(vp->ready_list));

#ifdef TWO_LEVEL
  if(next == NULL){
    next = mthread_work_take(vp);
  }
#endif

  if(vp->resched != NULL){
    mthread_log("SCHEDULER","Insert %p in ready list of %d\n",vp->resched,vp->rank);
    mthread_insert_last((struct mthread_s*)vp->resched,&(vp->ready_list));
    vp->resched = NULL;
  }

  if(current != vp->idle){
    if((current->status != BLOCKED) && (current->status != ZOMBIE)){
      if(current->status == RUNNING){
        vp->resched = current;
      } else not_implemented();
    }

    if(next == NULL){
      next = vp->idle;
    }
  }


  if(next != NULL){
    if(vp->current != next){
      mthread_log("SCHEDULER","Swap from %p to %p\n",current,next);
      vp->current = next;
      mthread_mctx_swap(current,next);
    }
  }

}

static void mthread_idle_task(void* arg){
  mthread_virtual_processor_t* vp;
  long j;
  int done = 0;
  vp = (mthread_virtual_processor_t*)arg;

  vp->state = 1;
  while(done == 0){
    done = 1;
    sched_yield();
    for(j = 0; j < MTHREAD_LWP; j++){
      if(virtual_processors[j].state == 0){
        done = 0;
      }
    }
  }
  mthread_log("SCHEDULER","Virtual processor %d started\n",vp->rank);
  while(1){
```



```
    __mthread_yield(vp);
  }
  not_implemented();
}

#ifdef TWO_LEVEL
static pthread_key_t lwp_key;
#endif

mthread_virtual_processor_t* mthread_get_vp(){
#ifdef TWO_LEVEL
  return pthread_getspecific(lwp_key);
#else
  return &(virtual_processors[0]);
#endif
}

int mthread_get_vp_rank(){
  return mthread_get_vp()->rank;
}

static inline void mthread_init_vp(mthread_virtual_processor_t* vp, struct mthre
ad_s* idle,
                                   struct mthread_s* current, int rank){
  vp->current = current;
  vp->idle = idle;
  mthread_list_init(&(vp->ready_list));
  vp->rank = rank;
  vp->resched = NULL;
}

static void* mthread_main(void* arg){
  not_implemented();
  return NULL;
}

static inline void mthread_init_lib(long i){
  struct mthread_s * mctx;
  struct mthread_s * current = NULL;
  char* stack;

  stack = (char*)safe_malloc(MTHREAD_DEFAULT_STACK);
  mctx = (struct mthread_s *)safe_malloc(sizeof(struct mthread_s));
  mthread_init_thread(mctx);

  mthread_list_init(&(joined_list));

  if(i == 0){
    current = (struct mthread_s *)safe_malloc(sizeof(struct mthread_s));
    mthread_init_thread(current);
    current->__start_routine = mthread_main;
    current->stack = NULL;
#ifdef TWO_LEVEL
    pthread_key_create(&lwp_key,NULL);
#endif
  }
#ifdef TWO_LEVEL
  pthread_setspecific(lwp_key,&(virtual_processors[i]));
#endif

  mthread_init_vp(&(virtual_processors[i]),mctx,mctx,i);
  mthread_mctx_set(mctx,mthread_idle_task,stack,MTHREAD_DEFAULT_STACK,&(virtual_
```

```
processors[i]));
  if(i != 0){
    virtual_processors[i].current = mctx;
    setcontext(&(mctx->uc));
  } else {
    virtual_processors[i].current = current;
  }
}

static void* mthread_lwp_start(void* arg){
  mthread_init_lib((long)arg);
  not_implemented();
  return NULL;
}

static void mthread_start_thread(void* arg){
  struct mthread_s * mctx;
  mthread_virtual_processor_t* vp;
  mctx = (struct mthread_s *)arg;
  mthread_log("THREAD INIT","Thread %p started\n",arg);
  mctx->res = mctx->__start_routine(mctx->arg);
  mctx->status = ZOMBIE;
  vp = mthread_get_vp();
  mthread_log("THREAD END","Thread %p ended (%d)\n",arg,vp->rank);
  __mthread_yield(vp);
}

/* Function for handling threads.  */

static inline void __mthread_lib_init(){
    mthread_log_init();
#ifdef TWO_LEVEL
    do{
      long i;
      for(i = 0; i < MTHREAD_LWP; i++){
        virtual_processors[i].state = 0;
      }
    }while(0);
#endif
    mthread_init_lib(0);
    virtual_processors[0].state = 1;
#ifdef TWO_LEVEL
    do{
      long i;
      long j;
      int done = 0;
      for(i = 1; i < MTHREAD_LWP; i++){
        pthread_t pid;
        pthread_create(&pid,NULL,mthread_lwp_start,(void*)i);
      }
      while(done == 0){
        done = 1;
        sched_yield();
        for(j = 0; j < MTHREAD_LWP; j++){
          if(virtual_processors[j].state == 0){
            done = 0;
          }
        }
      }
    }while(0);
#endif
    mthread_log("GENERAL","MThread library started\n");
```



```
}

/* Create a thread with given attributes ATTR (or default attributes
   if ATTR is NULL), and call function START_ROUTINE with given
   arguments ARG.  */
int
mthread_create (mthread_t * __threadp,
                const mthread_attr_t * __attr,
                void *(*__start_routine) (void *), void *__arg)
{
  static int is_init = 0;
  mthread_virtual_processor_t* vp;
  if(is_init == 0){
    __mthread_lib_init();
    is_init = 1;
  }

  vp = mthread_get_vp();

  if(__attr == NULL){
    struct mthread_s * mctx;
    char* stack;

    mctx = mthread_remove_first(&(joined_list));
    if(mctx == NULL){
      mctx = (struct mthread_s *)safe_malloc(sizeof(struct mthread_s));
    }
    if(mctx->stack == NULL){
      stack = (char*)safe_malloc(MTHREAD_DEFAULT_STACK);
    } else {
      stack = mctx->stack;
    }

    mthread_init_thread(mctx);
    mthread_log("THREAD INIT","Create thread %p\n",mctx);
    mctx->arg = __arg;
    mctx->__start_routine = __start_routine;
    mthread_mctx_set(mctx,mthread_start_thread,stack,MTHREAD_DEFAULT_STACK,mctx)
;
    mthread_insert_last(mctx,&(vp->ready_list));
    *__threadp = mctx;
  } else {
    not_implemented();
  }

  return 0;
}

/* Obtain the identifier of the current thread.  */
mthread_t
mthread_self (void)
{
  mthread_virtual_processor_t* vp;
  vp = mthread_get_vp();
  return (mthread_t)vp->current;
}

/* Compare two thread identifiers.  */
int
mthread_equal (mthread_t __thread1, mthread_t __thread2)
{
  return (__thread1 == __thread2);
```

```
}

/* Terminate calling thread.  */
void
mthread_exit (void *__retval)
{
  struct mthread_s * mctx;
  mthread_virtual_processor_t* vp;
  vp = mthread_get_vp();
  mctx = (struct mthread_s*)vp->current;

  mctx->res = __retval;

  mctx->status = ZOMBIE;
  mthread_log("THREAD END","Thread %p exited\n",mctx);
  __mthread_yield(vp);
}

/* Make calling thread wait for termination of the thread TH.  The
   exit status of the thread is stored in *THREAD_RETURN, if THREAD_RETURN
   is not NULL.  */
int
mthread_join (mthread_t __th, void **__thread_return)
{
  mthread_log("THREAD END","Join thread %p\n",__th);

  while(__th->status != ZOMBIE){
    mthread_yield();
  }

  *__thread_return = (void*)__th->res;
  mthread_log("THREAD END","Thread %p joined\n",__th);
  mthread_insert_last(__th,&(joined_list));

  return 0;
}

void mthread_yield(){
  mthread_virtual_processor_t* vp;
  vp = mthread_get_vp();
  mthread_log("THREAD YIELD","Thread %p yield\n",vp->current);
  __mthread_yield(vp);
}
```

```
#ifndef __MTHREAD_MTHREAD_H__
#define __MTHREAD_MTHREAD_H__
#ifdef __cplusplus
extern "C"
{
#endif
  /* Types */
  typedef  volatile unsigned int mthread_tst_t;

  struct mthread_s;
  typedef struct mthread_s* mthread_t;

  struct mthread_attr_s;
  typedef struct mthread_attr_s mthread_attr_t;

  struct mthread_mutex_s;
  typedef struct mthread_mutex_s mthread_mutex_t;

  struct mthread_mutexattr_s;
  typedef struct mthread_mutexattr_s mthread_mutexattr_t;

  struct mthread_cond_s;
  typedef struct mthread_cond_s mthread_cond_t;

  struct mthread_condattr_s;
  typedef struct mthread_condattr_s mthread_condattr_t;

  typedef unsigned int mthread_key_t;

  struct mthread_once_s;
  typedef struct mthread_once_s mthread_once_t;

  struct mthread_sem_s;
  typedef struct mthread_sem_s mthread_sem_t;

  /* Function for handling threads.  */

  /* Create a thread with given attributes ATTR (or default attributes
     if ATTR is NULL), and call function START_ROUTINE with given
     arguments ARG.  */
  extern int mthread_create (mthread_t * __threadp,
                             const mthread_attr_t * __attr,
                             void *(*__start_routine) (void *), void *__arg);

  /* Obtain the identifier of the current thread.  */
  extern mthread_t mthread_self (void);

  /* Compare two thread identifiers.  */
  extern int mthread_equal (mthread_t __thread1, mthread_t __thread2);

  /* Terminate calling thread.  */
  extern void mthread_exit (void *__retval);

  /* Make calling thread wait for termination of the thread TH.  The
     exit status of the thread is stored in *THREAD_RETURN, if THREAD_RETURN
     is not NULL.  */
  extern int mthread_join (mthread_t __th, void **__thread_return);

  /* Functions for mutex handling.  */

  /* Initialize MUTEX using attributes in *MUTEX_ATTR, or use the
     default values if later is NULL.  */
```



```
  extern int mthread_mutex_init (mthread_mutex_t * __mutex,
                                 const mthread_mutexattr_t * __mutex_attr);

  /* Destroy MUTEX.  */
  extern int mthread_mutex_destroy (mthread_mutex_t * __mutex);

  /* Try to lock MUTEX.  */
  extern int mthread_mutex_trylock (mthread_mutex_t * __mutex);

  /* Wait until lock for MUTEX becomes available and lock it.  */
  extern int mthread_mutex_lock (mthread_mutex_t * __mutex);

  /* Unlock MUTEX.  */
  extern int mthread_mutex_unlock (mthread_mutex_t * __mutex);

  /* Functions for handling conditional variables.  */

  /* Initialize condition variable COND using attributes ATTR, or use
     the default values if later is NULL.  */
  extern int mthread_cond_init (mthread_cond_t * __cond,
                                const mthread_condattr_t * __cond_attr);

  /* Destroy condition variable COND.  */
  extern int mthread_cond_destroy (mthread_cond_t * __cond);

  /* Wake up one thread waiting for condition variable COND.  */
  extern int mthread_cond_signal (mthread_cond_t * __cond);

  /* Wake up all threads waiting for condition variables COND.  */
  extern int mthread_cond_broadcast (mthread_cond_t * __cond);

  /* Wait for condition variable COND to be signaled or broadcast.
     MUTEX is assumed to be locked before.  */
  extern int mthread_cond_wait (mthread_cond_t * __cond,
                                mthread_mutex_t * __mutex);

  /* Functions for handling thread-specific data.  */

  /* Create a key value identifying a location in the thread-specific
     data area.  Each thread maintains a distinct thread-specific data
     area.  DESTR_FUNCTION, if non-NULL, is called with the value
     associated to that key when the key is destroyed.
     DESTR_FUNCTION is not called if the value associated is NULL when
     the key is destroyed.  */
  extern int mthread_key_create (mthread_key_t * __key,
                                 void (*__destr_function) (void *));

  /* Destroy KEY.  */
  extern int mthread_key_delete (mthread_key_t __key);

  /* Store POINTER in the thread-specific data slot identified by KEY. */
  extern int mthread_setspecific (mthread_key_t __key, const void *__pointer);

  /* Return current value of the thread-specific data slot identified by KEY.  *
/
  extern void *mthread_getspecific (mthread_key_t __key);


  /* Functions for handling initialization.  */

  /* Guarantee that the initialization function INIT_ROUTINE will be called
     only once, even if mthread_once is executed several times with the
```

```
     same ONCE_CONTROL argument. ONCE_CONTROL must point to a static or
     extern variable initialized to MTHREAD_ONCE_INIT.

     The initialization functions might throw exception which is why
     this function is not marked with .  */
  extern int mthread_once (mthread_once_t * __once_control,
                           void (*__init_routine) (void));

  /* Functions for handling semaphore.  */

  extern int mthread_sem_init (mthread_sem_t * sem, unsigned int value);
  extern int mthread_sem_wait (mthread_sem_t * sem);    /* P(sem), wait(sem) */
  extern int mthread_sem_post (mthread_sem_t * sem);    /* V(sem), signal(sem) *
/

  extern int mthread_sem_getvalue (mthread_sem_t * sem, int *sval);
  extern int mthread_sem_trywait (mthread_sem_t * sem);

  extern int mthread_sem_destroy (mthread_sem_t * sem); /* undo sem_init() */

  extern void mthread_yield();

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "mthread_internal.h"

  /* Functions for handling conditional variables.  */

  /* Initialize condition variable COND using attributes ATTR, or use
     the default values if later is NULL.  */
int
mthread_cond_init (mthread_cond_t * __cond,
                   const mthread_condattr_t * __cond_attr)
{
  not_implemented ();
  return 0;
}

  /* Destroy condition variable COND.  */
int
mthread_cond_destroy (mthread_cond_t * __cond)
{
  not_implemented ();
  return 0;
}

  /* Wake up one thread waiting for condition variable COND.  */
int
mthread_cond_signal (mthread_cond_t * __cond)
{
  not_implemented ();
  return 0;
}

  /* Wake up all threads waiting for condition variables COND.  */
int
mthread_cond_broadcast (mthread_cond_t * __cond)
{
  not_implemented ();
  return 0;
}

  /* Wait for condition variable COND to be signaled or broadcast.
     MUTEX is assumed to be locked before.  */
int
mthread_cond_wait (mthread_cond_t * __cond, mthread_mutex_t * __mutex)
{
  not_implemented ();
  return 0;
}
```

```
#include "mthread_internal.h"
#include <assert.h>
#include <stdarg.h>
#include <string.h>
#ifdef TWO_LEVEL
#include <pthread.h>
#endif


void
__not_implemented (const char *func, char *file, int line)
{
  fprintf (stderr, "Function %s in file %s at line %d not implemented\n",
           func, file, line);
  abort ();
}

void *safe_malloc(size_t size){
  void * tmp;
  tmp = malloc(size);
  assert(tmp != NULL);
  return tmp;
}

#ifdef TWO_LEVEL
static pthread_mutex_t mthread_fprintf_lock = PTHREAD_MUTEX_INITIALIZER;
#endif
static char* mthread_output_log_name = "mthread_log";
static FILE* mthread_output_log = NULL;

int mthread_log_init(){
  mthread_output_log = fopen(mthread_output_log_name,"w");
  return 0;
}

#define MTHREAD_LOG_PART 15

int mthread_log(char* part, const char *format, ...){
  char msg[4096];
  char part2[MTHREAD_LOG_PART +1];
  va_list ap;
  int i;
  int len;
#ifdef TWO_LEVEL
  int res;
  pthread_mutex_lock(&mthread_fprintf_lock);
#endif
  for(i = 0; i < MTHREAD_LOG_PART; i++){
    part2[i] = ' ';
  }
  len = strlen(part);
  if(len >= MTHREAD_LOG_PART){
    len = MTHREAD_LOG_PART;
  }
  memcpy(part2,part,len);
  part2[MTHREAD_LOG_PART] = '\0';

  sprintf(msg,"[LWP %02d Thread %p %s INFO:] %s",mthread_get_vp_rank(),
          mthread_self(),part2,format);

  va_start(ap, format);
  res = vfprintf(mthread_output_log, msg, ap);
```



```
  va_end(ap);
  fflush(mthread_output_log);
#ifdef TWO_LEVEL
  pthread_mutex_unlock(&mthread_fprintf_lock);
#endif
  return res;

}

int fprintf(FILE *stream, const char *format, ...){
  va_list ap;
#ifdef TWO_LEVEL
  int res;
  pthread_mutex_lock(&mthread_fprintf_lock);
#endif
  va_start(ap, format);
  res = vfprintf(stream, format, ap);
  va_end(ap);
  fflush(stream);
#ifdef TWO_LEVEL
  pthread_mutex_unlock(&mthread_fprintf_lock);
#endif
  return res;
}
```

```
#ifndef __MTHREAD_MTHREAD_INTERNAL_H__
#define __MTHREAD_MTHREAD_INTERNAL_H__
#ifdef __cplusplus
extern "C"
{
#endif


#define TWO_LEVEL


#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>

#ifndef __GNUC__
#define inline
#endif

#include "mthread.h"

typedef struct {
  volatile struct mthread_s* first;
  volatile struct mthread_s* last;
  mthread_tst_t lock;
}mthread_list_t;

typedef struct {
  struct mthread_s* idle;
  volatile struct mthread_s* current;
  mthread_list_t ready_list;
  int rank;
  volatile int state;
  volatile struct mthread_s* resched;
}mthread_virtual_processor_t;

typedef enum{RUNNING,BLOCKED,ZOMBIE} mthread_status_t;

struct mthread_s{
  ucontext_t uc;
  volatile void * res;
  void* arg;
  void *(*__start_routine) (void *);
  volatile struct mthread_s* next;
  volatile mthread_status_t status;
  void* stack;
};

#define MTHREAD_LIST_INIT {NULL,NULL,0}

  extern int mthread_test_and_set(mthread_tst_t *atomic);
  extern void mthread_spinlock_lock(mthread_tst_t *atomic);
  extern void mthread_spinlock_unlock(mthread_tst_t *atomic);
  extern int mthread_get_vp_rank();


  extern void __not_implemented (const char *func, char *file, int line);
  extern void *safe_malloc(size_t size);
  extern int mthread_log(char* part, const char *format, ...);
  extern int mthread_log_init();

  extern void mthread_insert_first(struct mthread_s* item, mthread_list_t* list)
```



```
;
  extern void mthread_insert_last(struct mthread_s* item, mthread_list_t* list);
  extern struct mthread_s* mthread_remove_first(mthread_list_t* list);

  extern void __mthread_yield(mthread_virtual_processor_t* vp);
  extern mthread_virtual_processor_t* mthread_get_vp();
#define not_implemented() __not_implemented(__FUNCTION__,__FILE__,__LINE__)

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "mthread_internal.h"

  /* Functions for handling thread-specific data.  */

  /* Create a key value identifying a location in the thread-specific
     data area.  Each thread maintains a distinct thread-specific data
     area.  DESTR_FUNCTION, if non-NULL, is called with the value
     associated to that key when the key is destroyed.
     DESTR_FUNCTION is not called if the value associated is NULL when
     the key is destroyed.  */
int
mthread_key_create (mthread_key_t * __key, void (*__destr_function) (void *))
{
  not_implemented ();
  return 0;
}

  /* Destroy KEY.  */
int
mthread_key_delete (mthread_key_t __key)
{
  not_implemented ();
  return 0;
}

  /* Store POINTER in the thread-specific data slot identified by KEY. */
int
mthread_setspecific (mthread_key_t __key, const void *__pointer)
{
  not_implemented ();
  return 0;
}

  /* Return current value of the thread-specific data slot identified by KEY.  *
/
void *
mthread_getspecific (mthread_key_t __key)
{
  not_implemented ();
  return NULL;
}
```

```
#include "mthread_internal.h"

  /* Functions for mutex handling.  */

  /* Initialize MUTEX using attributes in *MUTEX_ATTR, or use the
     default values if later is NULL.  */
int
mthread_mutex_init (mthread_mutex_t * __mutex,
                    const mthread_mutexattr_t * __mutex_attr)
{
  not_implemented ();
  return 0;
}

  /* Destroy MUTEX.  */
int
mthread_mutex_destroy (mthread_mutex_t * __mutex)
{
  not_implemented ();
  return 0;
}

  /* Try to lock MUTEX.  */
int
mthread_mutex_trylock (mthread_mutex_t * __mutex)
{
  not_implemented ();
  return 0;
}

  /* Wait until lock for MUTEX becomes available and lock it.  */
int
mthread_mutex_lock (mthread_mutex_t * __mutex)
{
  not_implemented ();
  return 0;
}

  /* Unlock MUTEX.  */
int
mthread_mutex_unlock (mthread_mutex_t * __mutex)
{
  not_implemented ();
  return 0;
}
```

```
#include "mthread_internal.h"
  /* Functions for handling initialization.  */

  /* Guarantee that the initialization function INIT_ROUTINE will be called
     only once, even if mthread_once is executed several times with the
     same ONCE_CONTROL argument. ONCE_CONTROL must point to a static or
     extern variable initialized to MTHREAD_ONCE_INIT.

     The initialization functions might throw exception which is why
     this function is not marked with .  */
int
mthread_once (mthread_once_t * __once_control, void (*__init_routine) (void))
{
  not_implemented ();
  return 0;
}
```

```
#include "mthread_internal.h"

  /* Functions for handling semaphore.  */

int
mthread_sem_init (mthread_sem_t * sem, unsigned int value)
{
  not_implemented ();
  return 0;
}

/* P(sem), wait(sem) */
int
mthread_sem_wait (mthread_sem_t * sem)
{
  not_implemented ();
  return 0;
}

/* V(sem), signal(sem) */
int
mthread_sem_post (mthread_sem_t * sem)
{
  not_implemented ();
  return 0;
}

int
mthread_sem_getvalue (mthread_sem_t * sem, int *sval)
{
  not_implemented ();
  return 0;
}

int
mthread_sem_trywait (mthread_sem_t * sem)
{
  not_implemented ();
  return 0;
}

/* undo sem_init() */
int
mthread_sem_destroy (mthread_sem_t * sem)
{
  not_implemented ();
  return 0;
}
```

```
#include "mthread_internal.h"
#include <sched.h>

#if defined(i686_ARCH) || defined(x86_64_ARCH)

static inline int __mthread_test_and_set(mthread_tst_t *atomic)
{
    int ret;

    __asm__ __volatile__("lock; xchgl %0, %1":"=r"(ret), "=m"(*atomic)
                         :"0"(1), "m"(*atomic)
                         :"memory");
    return ret;
}

#elif defined(sparc_ARCH)
static inline int __mthread_test_and_set(mthread_tst_t *spinlock)
{
  char ret = 0;

  __asm__ __volatile__("ldstub [%0], %1"
        : "=r"(spinlock), "=r"(ret)
        : "0"(spinlock), "1" (ret) : "memory");

  return (unsigned)ret;
}

#elif defined(ia64_ARCH)
static __inline__ int __mthread_test_and_set(mthread_tst_t *atomic)
{
    int ret;

    __asm__ __volatile__("xchg4 %0=%1, %2":"=r"(ret), "=m"(*atomic)
                         :"0"(1), "m"(*atomic)
                         :"memory");
    return ret;
}
#else
#define USE_GENERIC_ASM
#warning "Using generic test and set using pthread"
#include <pthread.h>
static pthread_mutex_t tst_mutex = PTHREAD_MUTEX_INITIALIZER;

static inline int __mthread_test_and_set(mthread_tst_t *atomic)
{
  int res;
  pthread_mutex_lock(&tst_mutex);
  res = *atomic;
  if(*atomic == 0){
    *atomic = 1;
  }
  pthread_mutex_unlock(&tst_mutex);
  return res;
}
#endif

int mthread_test_and_set(mthread_tst_t *atomic){
  return __mthread_test_and_set(atomic);
}

void mthread_spinlock_lock(mthread_tst_t *atomic){
#ifdef USE_GENERIC_ASM
```



```
  static pthread_mutex_t spin_tst_mutex = PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_lock(&spin_tst_mutex);
#endif
  while(mthread_test_and_set(atomic)){
    sched_yield();
  }
#ifdef USE_GENERIC_ASM
  pthread_mutex_unlock(&spin_tst_mutex);
#endif
}

void mthread_spinlock_unlock(mthread_tst_t *atomic){
  *atomic = 0;
}
```