```
#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_PROCESSORS];
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 mthread_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);
}
```