POSIX.1c Thread API

#include<pthread.h>
gcc ... -D_REENTRANT -lpthread

I. Thread Creation and Destruction

II. Synchronization:

A. Mutex

B. Condition Variables

C. Semaphores

■
Computer Sciences 640
University of Wisconsin - Madison
Fall 1997

Thread Creation and Termination

int *pthread_create*(pthread_t *thread, pthread_attr_t *attr, void *(*start_function)(void *), void *arg);

all thread start functions must look like:

*void *start_function(void *arg);*

"void *" serves as a generic type, from/to which you can cast useful information

void *pthread_exit*(void *status);

kills the calling thread, returning its exit status in **status*

int *pthread_join*(pthread_t thread, void **status);

wait for the specified thread to exit, and store its exit status in *status

pthread_t *pthread_self*();

returns calling thread's thread_id

■
Computer Sciences 640
University of Wisconsin - Madison
Fall 1997

Mutex - guarantees mutual exclusion

int *pthread_mutex_init*(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

initializes mutex state (does not allocate memory)

you probably want to pass *attr* as **NULL**

int *pthread_mutex_destroy*(pthread_mutex_t *mutex);

destroy mutex state (does not free memory)

int *pthread_mutex_lock*(pthread_mutex_t *mutex);

lock mutex, or block until it becomes unlocked

int *pthread_mutex_unlock*(pthread_mutex_t *mutex);

unlock mutex

int *pthread_mutex_trylock*(pthread_mutex_t *mutex);

lock mutex, or returns **EBUSY** if it is already locked

■
Computer Sciences 640
University of Wisconsin - Madison
Fall 1997

Condition Variables

int *pthread_cond_init*(pthread_cond_t *cond, const pthread_condattr_t *attr);

initializes a condition variable (does not allocate memory)

you probably want to pass *attr* as **NULL**

int *pthread_cond_destroy*(pthread_cond_t *cond);

destroys a condition variable's state (does not free memory)

int *pthread_cond_signal*(pthread_cond_t *cond);

signal a thread waiting on a condition variable

int *pthread_cond_broadcast*(pthread_cond_t *cond);

signal all threads waiting on a condition variable

int *pthread_cond_wait*(pthread_cond_t *cond, pthread_mutex_t *mutex);

wait on a condition variable

should only occur when the calling thread holds the *mutex* lock

will release the *mutex* lockl while waiting, and automatically regain it when awoken

■
Computer Sciences 640
University of Wisconsin - Madison
Fall 1997

Condition Variables *continued...*

int *pthread_cond_timedwait*(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

Identical to ***pthread_cond_wait*** above, except will wake-up at *abstime*

very nice for implementing timeouts: returns **ETIME** if *abstime* was reached

```
struct timespec {
     time_t tv_sec;  /* seconds */
     long   tv_nsec; /* and nanoseconds */
};

...
timeval current_time;
timespec timeout_time;

gettimeofday(&current_time, NULL);

timeout_time.tv_sec = current_time.tv_sec + 2;
/* convert from micro to nano */
timeout_time.tv_nsec = current_time.tv_usec * 1000;

result = pthread_cond_timedwait(&cond, &mutex, &timeout_time);
if (result == ETIME)
  printf("timeout occured");
else
  printf("cond was signalled");

...
```


Semaphores

**** not part of POSIX.1c thread interface, instead is part of POSIX.4. Interface is slightly different than POSIX.1c mutexes and condition variables above***

#include<semaphore.h>

gcc ... -lposix4

int *sem_init*(sem_t *sem, int pshared, unsigned int initial_value);

initialize a semaphore (does not allocate memory)

int *sem_destroy*(sem_t *sem);

destroy a semaphore's state (does not free allocated memory)

int *sem_post*(sem_t *sem);

do an "up" or a "V" operation on a semaphore

int *sem_wait*(sem_t *sem);

do a "down" or a "P" operation on a semaphore, blocking if appropriate

int *sem_trywait*(sem_t *sem);

same as ***sem_wait***, except returns -1 and sets errno = **EAGAIN** if call would block