pthread.h

This includes a thread ID that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, an `errno` variable and thread-specific data. Everything within a process is sharable among the threads in a process, including the text of the executable program, the program's global and heap memory, the stacks, and the file descriptors.

the `pid_t` data type, is a non-negative integer. A thread ID is represented by the pthread_t data type.

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

The `pthread_equal()` function returns a non-zero value if t1 and t2 are equal; otherwise, zero is returned.

```
int pthread_cancel(pthread_t thread);
```

The `pthread_cancel()` function requests that thread be canceled.

If successful, the `pthread_cancel()` function returns zero. Otherwise, an error number is returned to indicate the error.

The `pthread_cancel()` function may fail if, No thread could be found corresponding to that specified by the given thread ID.

Note that `pthread_cancel` doesn't wait for the thread to terminate; it merely makes the request.

`pthread_cancel()` is behave like `pthread_exit()` with an argument of `PTHREAD_CANCELED`.

Mutexes:

We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces. A mutex is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done.

A mutex variable is represented by the `pthread_mutex_t` data type. Before we can use a mutex variable, we must first initialize it by either setting it to the constant `PTHREAD_MUTEX_INITIALIZER` (for statically allocated mutexes only) or calling `pthread_mutex_init`.

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

If successful, the `pthread_mutex_init()` and `pthread_mutex_destroy()` functions return zero. Otherwise, an error number is returned to indicate the error.

To initialize a mutex with the default attributes, we set `attr` to `NULL`.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions return zero. Otherwise, an error number is returned to indicate the error.

Deadlock Avoidance:

Assume that you have two mutexes, A and B, that you need to lock at the same time. If all threads always lock mutex A before mutex B, no deadlock can occur from the use of the two mutexes.

You can use the `pthread_mutex_trylock` interface to avoid deadlock.

```
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict tsptr);
```

The timeout specifies how long we are willing to wait in terms of absolute time (in seconds). Both return: 0 if OK, error number on failure

Reader–Writer Locks:

Reader–writer locks are similar to mutexes, except that they allow for higher degrees of parallelism. With a mutex, the state is either **locked** or **unlocked**, and only one thread can lock it at a time. Three states are possible with a reader–writer lock: locked in read mode, locked in write mode, and unlocked. Only one thread at a time can hold a reader–writer lock in write mode, but multiple threads can hold a reader–writer lock in read mode at the same time.

Reader–writer locks are well suited for situations in which data structures are read more often than they are modified.

Reader–writer locks are also called **shared–exclusive** locks. When a reader–writer lock is read locked, it is said to be locked in **shared** mode. When it is write locked, it is said to be locked in **exclusive** mode.

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

If successful, the `pthread_rwlock_init()` and `pthread_rwlock_destroy()` functions return 0. Otherwise, an error number is returned to indicate the error.

To lock a reader–writer lock in read mode, we call `pthread_rwlock_rdlock`. To write lock a reader–writer lock, we call `pthread_rwlock_wrlock`.

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

If successful, the `pthread_rwlock_wrlock()` function returns zero. Otherwise, an error number is returned to indicate the error.

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

If successful, the `pthread_rwlock_rdlock()` function returns zero. Otherwise, an error number is returned to indicate the error.

```
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
const struct timespec *restrict tsptr);

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
const struct timespec *restrict tsptr);
```

Both return: 0 if OK, error number on failure.

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Regardless of how we lock a reader–writer lock, we can unlock it by calling `pthread_rwlock_unlock`.
The `pthread_rwlock_unlock()` function is called to release a lock held on the read-write lock object referenced by `rwlock`. Results are undefined if the read-write lock `rwlock` is not held by the calling thread.

If successful, the `pthread_rwlock_unlock()` function returns zero. Otherwise, an error number is returned to indicate the error.