POSIX Threads (Pthreads)

- POSIX Threads define POSIX standard for multithreaded API (IEEE POSIX 1003.1-1995)
- The functions comprising core functionality of Pthreads can be divided into three classes:
  - Thread management
  - Mutexes
  - Condition variables
- Pthreads define the interface using C language types, function prototypes and macros
- Naming conventions for identifiers:
  - *pthread_*: Threads themselves and miscellaneous subroutines
  - *pthread_attr_*: Thread attributes objects
  - *pthread_mutex_*: Mutexes
  - *pthread_mutexattr_*: Mutex attributes objects
  - *pthread_cond_*: Condition variables
  - *pthread_condattr_*: Condition attributes objects
  - *pthread_key_*: Thread-specific data keys

References:
1.http://www.llnl.gov/computing/tutorials/pthreads/
2.http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

Pthreads: Thread Creation

Function: pthread_create()

int pthread_create(pthread_t *_thread_, const pthread_attr_t *_attr_,
void *(*_routine_)(void *), void *_arg_);

Description:

Creates a new thread within a process. The created thread starts execution of *routine*, which is passed a pointer argument *arg*. The attributes of the new thread can be specified through *attr*, or left at default values if *attr* is null. Successful call returns 0 and stores the id of the new thread in location pointed to by *thread*, otherwise an error code is returned.

```
#include <pthread.h>
...
void *do_work(void *input_data)
{ /* this is thread's starting routine */
  ...
}
...
pthread_t id;
struct {. . .} args = {. . .}; /* struct containing thread arguments */
int err;
...
/* create new thread with default attributes */
err = pthread_create(&id, NULL, do_work, (void *)&args);
if (err != 0) {/* handle thread creation failure */}
...
```

Pthreads: Thread Join

Function: pthread_join()

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

Description:

Suspends the execution of the calling thread until the target thread terminates (either by returning from its startup routine, or calling *pthread_exit*()), unless the target thread already terminated. If *value_ptr* is not null, the return value from the target thread or argument passed to *pthread_exit*() is made available in location pointed to by *value_ptr*. When *pthread_join*() returns successfully (i.e. with zero return code), the target thread has been terminated.

```
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
void *result_ptr;
int err;
...
/* create worker thread */
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_join(id, &result_ptr);
if (err != 0) {/* handle join error */}
else {/* the worker thread is terminated and result_ptr points to its return value */
  ...
}
```

Pthreads: Thread Exit

Function: pthread_exit()

void pthread_exit(void *value_ptr);

Description:

Terminates the calling thread and makes the *value_ptr* available to any successful join with the terminating thread. Performs cleanup of local thread environment by calling cancellation handlers and data destructor functions. Thread termination does not release any application visible resources, such as mutexes and file descriptors, nor does it perform any process-level cleanup actions.

```
#include <pthread.h>
...
void *do_work(void *args)
{
  ...
  pthread_exit(&return_value);
  /* the code following pthread_exit is not executed */
  ...
}
...
void *result_ptr;
pthread_t id;
pthread_create(&id, NULL, do_work, (void *)&args);
...
pthread_join(id, &result);
/* result_ptr now points to return_value */
...
```

Pthreads: Thread Termination

Function: pthread_cancel()

void pthread_cancel(thread_t *thread*);

Description:

The *pthread_cancel*() requests cancellation of thread *thread*. The ability to cancel a thread is dependent on its state and type.

```
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
pthread_t id;
int err;
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_cancel(id);
if (err != 0) {/* handle cancelation failure */}
...
```

Pthreads: Detached Threads

Function: pthread_detach()

int pthread_detach(pthread_t *thread*);

Description:

Indicates to the implementation that storage for thread *thread* can be reclaimed when the thread terminates. If the thread has not terminated, *pthread_detach*() is not going to cause it to terminate. Returns zero on success, error number otherwise.

```
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
pthread_t id;
int err;
...
/* start a new thread */
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_detach(id);
if (err != 0) {/* handle detachment failure */}
else {/* master thread doesn't join the worker thread;
         the worker thread resources will be released automatically
         after it terminates */
  ...
}
```

Pthreads: Operations on Mutex Objects (I)

Function: pthread_mutex_lock(), pthread_mutex_unlock()

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

Description:

The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. After successful return from the call, the mutex object referenced by *mutex* is in locked state with the calling thread as its owner.

The mutex object referenced by *mutex* is released by calling *pthread_mutex_unlock*(). If there are threads blocked on the mutex, scheduling policy decides which of them shall acquire the released mutex.

```
#include <pthread.h>
...
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
/* lock the mutex before entering critical section */
pthread_mutex_lock(&mutex);
/* critical section code */
...
/* leave critical section and release the mutex */
pthread_mutex_unlock(&mutex);
...
```

Pthreads: Operations on Mutex Objects (II)

Function: pthread_mutex_trylock()

int pthread_mutex_trylock(pthread_mutex_t *mutex);

Description:

The function *pthread_mutex_trylock*() is equivalent to *pthread_mutex_lock*() , except that if the mutex object is currently locked, the call returns immediately with an error code EBUSY. The value of 0 (success) is returned only if the mutex has been acquired.

```
#include <pthread.h>
...
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int err;
...
/* attempt to lock the mutex */
err = pthread_mutex_trylock(&mutex);
switch (err) {
case 0: /* lock acquired; execute critical section code and release mutex */
  ...
  pthread_mutex_unlock(&mutex);
  break;
case EBUSY: /* someone already owns the mutex; do something else instead of blocking */
  ...
  break;
default: /* some other failure */
  ...
  break;
}
```

Pthread Mutex Types

- Normal
  - No deadlock detection on attempts to relock already locked mutex
- Error-checking
  - Error returned when locking a locked mutex
- Recursive
  - Maintains *lock count* variable
  - After the first acquisition of the mutex, the lock count is set to one
  - After each successful relock, the lock count is increased; after each unlock, it is decremented
  - When the lock count drops to zero, thread loses the mutex ownership
- Default
  - Attempts to lock the mutex recursively result in an undefined behavior
  - Attempts to unlock the mutex which is not locked, or was not locked by the calling thread, results in undefined behavior

Pthreads: Condition Variables

Function: pthread_cond_wait(),
pthread_cond_signal(), pthread_cond_broadcast()

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
Int pthread_cond_broadcast(pthread_cond_t *cond);
```

Description:

The *pthread_cond_wait*() blocks on a condition variable associated with a mutex. The function must be called with a locked mutex argument. It atomically releases the mutex and causes the calling thread to block. While in that state, another thread is permitted to access the mutex. Subsequent mutex release should be announced by the accessing thread through *pthread_cond_signal*() or *pthread_cond_broadcast*(). Upon successful return from *pthread_cond_wait*(), the mutex is in locked state with the calling thread as its owner.

The *pthread_cond_signal*() unblocks at least one of the threads that are blocked on the specified condition variable *cond*.

The pthread_cond_broadcast() unblocks all threads currently blocked on the specified condition variable *cond*.

All of these functions return zero on successful completion, or an error code otherwise.

Example: Condition Variable

Initialization and startup

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* create default mutex */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; /* create default condition variable */
pthread_t prod_id, cons_id;
item_t buffer; /* storage buffer (shared access) */
int empty = 1; /* buffer empty flag (shared access) */
...
pthread_create(&prod_id, NULL, producer, NULL); /* start producer thread */
pthread_create(&cons_id, NULL, consumer, NULL); /* start consumer thread */
...
```

Simple producer thread

```
void *producer(void *none) {
  while (1) {
    /* obtain next item, asynchronously */
    item_t item = compute_item();
    pthread_mutex_lock(&mutex);
    /* critical section starts here */
    while (!empty)
      /* wait until buffer is empty */
      pthread_cond_wait(&cond, &mutex);
    /* store item, update status */
    buffer = item;
    empty = 0;
    /* wake waiting consumer (if any) */
    pthread_cond_signal(&cond);
    /* critical section done */
    pthread_mutex_unlock(&mutex);
  }
}
```

Simple consumer thread

```
void *consumer(void *none) {
  while (1) {
    item_t item;
    pthread_mutex_lock(&mutex);
    /* critical section starts here */
    while (empty)
      /* block (nothing in buffer yet) */
      pthread_cond_wait(&cond, &mutex);
    /* grab item, update buffer status */
    item = buffer;
    empty = 1;
    /* critical section done */
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    /* process item, asynchronously */
    consume_item(item);
  }
}
```

Pthreads: Dynamic Initialization

Function: pthread_once()

int pthread_once(pthread_once_t *control, void (**init_routine*)(void));

Description:

The first call to *pthread_once*() by any thread in a process will call the *init_routine*() with no arguments. Subsequent calls to *pthread_once*() with the same control will not call *init_routine*().

```
#include <pthread.h>
...
pthread_once init_ctrl = PTHREAD_ONCE_INIT;
...
void initialize() {/* initialize global variables */}
...
void *do_work(void *arg)
{ /* make sure global environment is set up */
  pthread_once(&init_ctrl, initialize);
  /* start computations */
  ...
}
...
pthread_t id;
pthread_create(&id, NULL, do_work, NULL);
...
```

Pthreads: Get Thread ID

Function: pthread_self()

```
pthread_t pthread_self(void);
```

Description:

Returns the thread ID of the calling thread.

```
#include <pthread.h>
...
pthread_t id;
id = pthread_self();
...
```