Pthreads

Dong-kun Shin
Embedded Software Laboratory
Sungkyunkwan University
http://nyx.skku.ac.kr

The Pthreads API

- **ANSI/IEEE POSIX1003.1-1995 Standard**
- **Thread management**
  - Work directly on threads – creating, terminating, joining, etc.
  - Include functions to set/query thread attributes.
- **Mutexes**
  - Provide for creating, destroying, locking and unlocking mutexes.
- **Condition variables (next time)**
  - Include functions to create, destroy, wait and signal based upon specified variable values.

Creating Threads (1)

- **`int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`**
  - **`pthread_create()`** returns the new thread ID via the **`thread`** argument.
    - The caller can use this thread ID to perform various operations on the thread.
  - The **`attr`** parameter is used to set thread attributes.
    - NULL for the default values.
  - The **`start_routine`** denotes the C routine that the thread will execute once it is created.
    - C routine that the thread will execute once it is created.
  - A single argument may be passed to **`start_routine()`** via **`arg`**.

Creating Threads (2)

- **Notes:**
  - Initially, **`main()`** comprises a single, default thread.
  - All other threads should must be explicitly created by the programmer.
  - Once created, threads are peers, and may create other threads.
  - The maximum number of threads that may be created by a process is implementation dependent.

Terminating Threads

- **`void pthread_exit (void *retval)`**
  - **`pthread_exit()`** terminates the execution of the calling thread.
    - Typically, this is called after a thread has completed its work and is no longer required to exist.
  - The **`retval`** argument is the return value of the thread.
    - It can be consulted from another thread using **`pthread_join()`.**
  - It does not close files; any files opened inside the thread will remain open after the thread is terminated.

Cancelling Threads

- **`int pthread_cancel (pthread_t thread)`**
  - **`pthread_cancel()`** sends a cancellation request to the thread denoted by the **`thread`** argument.
  - Depending on its settings, the target thread can then either ignore request, honor it immediately, or defer it till it reaches a cancellation point.
    - **`pthread_setcancelstate(): PTHREAD_CANCEL_(ENABLE|DISABLE)`**
    - **`pthread_setcanceltype(): PTHREAD_CANCEL_(DEFERRED|ASYNCHRONOUS)`**
  - Threads are always created by **`pthread_create()`** with cancellation enabled and deferred.

Joining Threads

- **`int pthread_join (pthread_t thread, void **retval)`**
  - **`pthread_join()`** suspends the execution of the calling thread until the thread identified by **`thread`** terminates, either by calling **`pthread_exit()`** or by being cancelled.
  - The return value of **`thread`** is stored in the location pointed by **`retval.`**
  - It returns **`PTHREAD_CANCELED`** if thread was canceled.
  - It is impossible to join a detached thread.

Detaching Threads

- **`int pthread_detach (pthread_t thread)`**
  - **`pthread_detach()`** puts the thread in the detached state.
    - This guarantees that the memory resources consumed by **`thread`** will be freed immediately when thread terminates.
    - However, this prevents other threads from synchronizing on the termination of thread using **`pthread_join().`**
  - A thread can be detached when it is created:

```
pthread_t tid;
pthread_attr_t attr;

pthread_attr_init (&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, start_routine, NULL);
pthread_attr_destroy (&attr);
```

Thread Identifiers

- **`pthread_t pthread_self (void)`**
  - **`pthread_self()`** returns the unique, system assigned thread ID of the calling thread.

- **`int pthread_equal (pthread_t t1, pthread_t t2)`**
  - **`pthread_equal()`** returns a non-zero value if **`t1`** and **`t2`** refer to the same thread.
  - Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs against each other.

Example


```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS  10

void *PrintHello(void *threadid) {
   printf("\n%d: Hello World!\n", (int)threadid);
   pthread_exit(NULL);
}

int main () {
   pthread_t tid[NTHREADS];
   int t;
   for (t = 0; t < NTHREADS; t++)
      pthread_create(&tid[t], NULL, PrintHello, (void *)t);
   for (t = 0; t < NTHREADS; t++)
      pthread_join(tid[t], NULL);

   return 0;
}
```

Mutex (1)

- **Mutex is an abbreviation for "mutual exclusion"**
  - Primary means of implementing thread synchronization.
    - Protects shared data when multiple writes occurs.
  - A mutex variable acts like a "lock" protecting access to a shared resource.
    - Only one thread can lock (or own) a mutex variable at any given time.
    - Even if several threads try to lock a mutex, only one thread will be successful. Other threads are blocked until the owner releases the lock.
  - Mutex is used to prevent "race" conditions.
    - race condition: anomalous behavior due to unexpected critical dependence on the relative timing of events.

Mutex (2)


```
int deposit(int amount)
{
  int balance;

  balance = get_balance();
  balance += amount;
  put_balance(balance);
  return balance;
}
```

```
int withdraw(int amount)
{
  int balance;

  balance = get_balance();
  balance -= amount;
  put_balance(balance);
  return balance;
}
```

T1 executes deposit(100)

```
balance = get_balance();
balance += 100;



put_balance(balance);
```

T2 executes withdraw(300)

```
balance = get_balance();
balance -= 300;
put_balance(balance);
```

Creating/Destroying Mutexes

- **Static initialization**
  - `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- **Dynamic initialization**
  - `pthread_mutex_t m;`
    `pthread_mutex_init (&m, (pthread_mutexattr_t *)NULL);`
- **Destroying a mutex**
  - `pthread_mutex_destroy (&m);`
  - Destroys a mutex object, freeing the resources it might hold.

Using Mutexes (1)

- **`int pthread_mutex_lock (pthread_mutex_t *mutex)`**
  - Acquire a lock on the specified **`mutex`** variable.
  - If the **`mutex`** is already locked by another thread, block the calling thread until the **`mutex`** is unlocked.

- **`int pthread_mutex_unlock (pthread_mutex_t *mutex)`**
  - Unlock a **`mutex`** if called by the owning thread.

- **`int pthread_mutex_trylock (pthread_mutex_t *mutex)`**
  - Attempt to lock a **`mutex`**.
  - If the **`mutex`** is already locked, return immediately with a "busy" error code.

Using Mutexes (2)


```
pthread_mutex_t m =
     PTHREAD_MUTEX_INITIALIZER;

int deposit(int amount)
{
  int balance;

  pthread_mutex_lock(&m);

  balance = get_balance();
  balance += amount;
  put_balance(balance);

  pthread_mutex_unlock(&m);

  return balance;
}
```

```
int withdraw(int amount)
{
  int balance;

  pthread_mutex_lock(&m);

  balance = get_balance();
  balance -= amount;
  put_balance(balance);

  pthread_mutex_unlock(&m);

  return balance;
}
```

Exercise

- **Prove the following facts through your program**
  - Facts
    - Consistency may not be guaranteed when the mutex is not used.
    - Longer *critical sections* increase execution time.
  - Implement using mutex example (p.15)
  - **You have to increase the number of CPU on the virtual machine.**