

Pthreads

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
<http://csl.skku.edu>



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**
 - 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_CANCELLED** if thread was cancelled.
 - 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);
}
```

Threads Synchronization

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
<http://csl.skku.edu>



Mutex (1)

- **Mutex is an abbrev. 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;  
}
```


Condition Variables (1)

- **Another way for thread synchronization**
 - While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
 - Without condition variables, the programmer would need to have threads continually polling to check if the condition is met.
 - This can be very resource consuming since the thread would be continuously busy in this activity.
 - A condition variable is always used in conjunction with a mutex lock.

Condition Variables (2)

■ How condition variables work

- A thread locks a mutex associated with a condition variable.
- The thread tests the condition to see if it can proceed.
- If it can
 - Your thread does its work
 - Your thread unlocks the mutex
- If it cannot
 - The thread sleeps. **The mutex is automatically released.**
 - Some other threads signals the condition variable.
 - Your thread wakes up from waiting **with the mutex automatically locked**, and it does its work.
 - Your thread releases the mutex when it's done.

Creating/Destroying CV

■ Static initialization

- `pthread_cond_t cond =
PTHREAD_COND_INITIALIZER;`

■ Dynamic initialization

- `pthread_cond_t cond;
pthread_cond_init (&cond, (pthread_condattr_t
*)NULL);`

■ Destroying a condition variable

- `pthread_cond_destroy (&cond);`
- Destroys a condition variable, freeing the resources it might hold.

Using Condition Variables

- **int pthread_cond_wait** (pthread_cond_t *cond, pthread_mutex_t *mutex)
 - Blocks the calling thread until the specified condition is signalled.
 - This should be called while mutex is locked, and it will automatically release the mutex while it waits.
- **int pthread_cond_signal** (pthread_cond_t *cond)
 - Signals another thread which is waiting on the condition variable.
 - Calling thread should have a lock.
- **int pthread_cond_broadcast**(pthread_cond_t *cond)
 - Used if more than one thread is in a blocking wait state.

Producer-Consumer (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define QSIZE          5
#define LOOP          30

typedef struct {
    int data[QSIZE];
    int index;
    int count;
    pthread_mutex_t lock;
    pthread_cond_t notfull;
    pthread_cond_t notempty;
} queue_t;

void *produce (void *args);
void *consume (void *args);
void put_data (queue_t *q, int d);
int get_data (queue_t *q);
```

Producer-Consumer (2)

```
int main ()
{
    queue_t *q;
    pthread_t producer, consumer;

    q = qinit();

    pthread_create(&producer, NULL, produce, (void *)q);
    pthread_create(&consumer, NULL, consume, (void *)q);

    pthread_join (producer, NULL);
    pthread_join (consumer, NULL);

    qdelete();
}
```

Producer-Consumer (3)

```
queue_t *qinit()
{
    queue_t *q;

    q = (queue_t *) malloc(sizeof(queue_t));
    q->index = q->count = 0;
    pthread_mutex_init(&q->lock, NULL);
    pthread_cond_init(&q->notfull, NULL);
    pthread_cond_init(&q->notempty, NULL);

    return q;
}

void qdelete(queue_t *q)
{
    pthread_mutex_destroy(&q->lock);
    pthread_cond_destroy(&q->notfull);
    pthread_cond_destroy(&q->notempty);
    free(q);
}
```

Producer-Consumer (4)

```
void *produce(void *args)
{
    int i, d;
    queue_t *q = (queue_t *)args;
    for (i = 0; i < LOOP; i++) {
        d = random() % 10;
        put_data(q, d);
        printf("put data %d to queue\n", d);
    }
    pthread_exit(NULL);
}

void *consume(void *args)
{
    int i, d;
    queue_t *q = (queue_t *)args;
    for (i = 0; i < LOOP; i++) {
        d = get_data(q);
        printf("got data %d from queue\n", d);
    }
    pthread_exit(NULL);
}
```


Producer-Consumer (5)

```
void put_data(queue_t *q, int d)
{
    pthread_mutex_lock(&q->lock);
    while (q->count == QSIZE) pthread_cond_wait(&q->notfull, &q->lock);
    q->data[(q->index + q->count) % QSIZE] = d;
    q->count++;
    pthread_cond_signal(&q->notempty);
    pthread_mutex_unlock(&q->lock);
}

int get_data(queue_t *q)
{
    int d;
    pthread_mutex_lock(&q->lock);
    while (q->count == 0) pthread_cond_wait(&q->notempty, &q->lock);
    d = q->data[q->index];
    q->index = (q->index + 1) % QSIZE;
    q->count--;
    pthread_cond_signal(&q->notfull);
    pthread_mutex_unlock(&q->lock);
    return d;
}
```

Thread Safety (1)

▪ Thread-safe

- Functions called from a thread must be **thread-safe**.
- We identify four (non-disjoint) classes of thread-unsafe functions:
 - Class 1: Failing to protect shared variables
 - Class 2: Relying on persistent state across invocations
 - Class 3: Returning a pointer to a static variable
 - Class 4: Calling thread-unsafe functions

Thread Safety (2)

- **Class 1: Failing to protect shared variables.**
 - Fix: Use mutex operations.
 - Issue: Synchronization operations will slow down code.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int cnt = 0;

/* Thread routine */
void *count(void *arg) {
    int i;

    for (i=0; i<NITERS; i++) {
        pthread_mutex_lock (&lock);
        cnt++;
        pthread_mutex_unlock (&lock);
    }
    return NULL;
}
```

Thread Safety (3)

- **Class 2: Relying on persistent state across multiple function invocations.**
 - Random number generator relies on static state
 - Fix: Rewrite function so that caller passes in all necessary state.

```
/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
    static unsigned int next = 1;
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

Thread Safety (4)

- **Class 3: Returning a ptr to a static variable.**

- **Fixes:**

1. Rewrite code so caller passes pointer to **struct**.

- Issue: Requires changes in caller and callee.

```
struct hostent
*gethostbyname(char *name){
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}
```

```
hostp = malloc(...);
gethostbyname_r(name, hostp);
```

2. *Lock-and-copy*

- Issue: Requires only simple changes in caller (and none in callee)

- » However, caller must free memory.

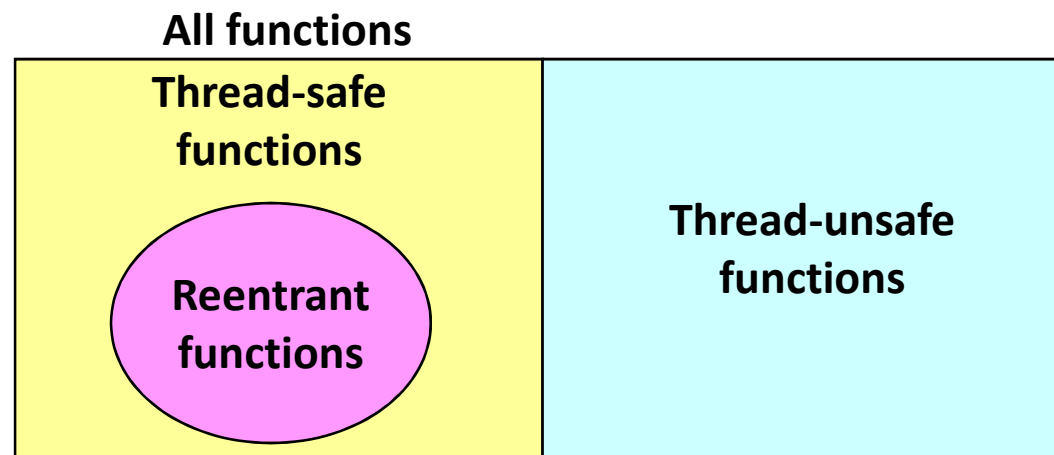
```
struct hostent
*gethostbyname_ts(char *name)
{
    struct hostent *unshared
        = malloc(...);
    pthread_mutex_lock(&lock); /* lock */
    shared = gethostbyname(name);
    *unshared = *shared; /* copy */
    pthread_mutex_unlock(&lock);
    return q;
}
```

Thread Safety (5)

- **Class 4: Calling thread-unsafe functions.**
 - Calling one thread-unsafe function makes an entire function thread-unsafe.
 - Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

- A function is **reentrant** iff it accesses **NO** shared variables when called from multiple threads.
 - Reentrant functions are a proper subset of the set of thread-safe functions.



- NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant.

Thread-Safe Library

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe.
 - Examples: **malloc**, **free**, **printf**, **scanf**
- Most Unix system calls are thread-safe, with a few exceptions:

| Thread-unsafe function | Class | Reentrant version |
|------------------------|-------|-------------------|
| asctime | 3 | asctime_r |
| ctime | 3 | ctime_r |
| gethostbyaddr | 3 | gethostbyaddr_r |
| gethostbyname | 3 | gethostbyname_r |
| inet_ntoa | 3 | (none) |
| localtime | 3 | localtime_r |
| rand | 2 | rand_r |