Pthreads (2)

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

Last Class Review

- **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 (Today)**
  - Include functions to create, destroy, wait and signal based upon specified variable values.

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.
  2. *Lock-and-copy*
     - Issue: Requires only simple changes in caller (and none in callee)
       - However, caller must free memory.

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

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

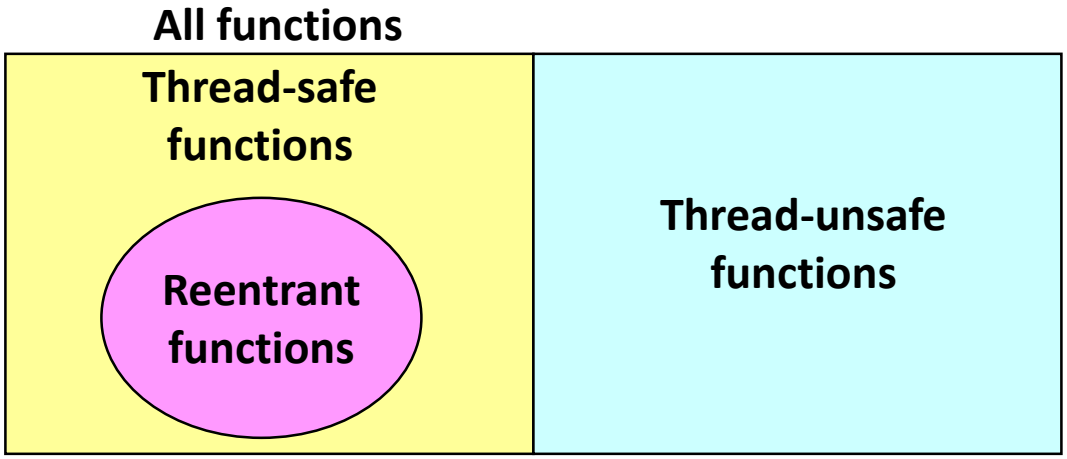
```
struct hostent
*gethostbyname_ts(char *name)
{
  struct hostent *unshared
                      = malloc(...);
  pthread_mutex_lock(&lock); /* lock */
  shared = gethostbyname(name);
  *unshared = *shared;   /* copy */
  pthread_mutex_unlock(&lock);
  return q;
}
```

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

Exercise

- **아래의 내용을 conditional variable을 사용하여 구현 후, 자신의 프로그램이 잘 작동함을 증명**

  - 가정
    - deposit(): 수당을 받음
    - withdraw(): 카드 값이 빠져나감

  - 조건
    - 하나의 카드 값은 나눠서 빠져나가지 않는다.
    - 카드 값이 빠져나간 후의 잔고는 0원보다 작을 수 없다.
    - 카드 값은 잔고가 충분해 지면 즉시 출금된다.