Threads Synchronization

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 occur.
  - 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)

```c
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)

```c
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 pthreadCondWait**( 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 pthreadCondSignal**( pthread_cond_t *cond )
  - Signals another thread which is waiting on the condition variable.
  - Calling thread should have a lock.

- **int pthreadCondBroadcast**( pthread_cond_t *cond )
  - Used if more than one thread is in a blocking wait state.
#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);
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();
}
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);
}
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);
}
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.

```c
#include <pthread.h>

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;
}
```
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.

```c
/* 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.

```c
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:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>3</td>
<td>asctime_r</td>
</tr>
<tr>
<td>ctime</td>
<td>3</td>
<td>ctime_r</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>3</td>
<td>gethostbyaddr_r</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>3</td>
<td>gethostbyname_r</td>
</tr>
<tr>
<td>inet_ntoa</td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime</td>
<td>3</td>
<td>localtime_r</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>rand_r</td>
</tr>
</tbody>
</table>
Exercise

- Make echo server
  - Support multithread
  - Maintains
    - Clients connected
    - Bytes echoed