Threads Synchronization

Prof. Jin-Soo Kim (jinsookim@skku.edu)
TA – Dong-Yun Lee (dylee@csl.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)

```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

- `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.
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
Class 1: Failing to protect shared variables.

- Fix: Use mutex operations.
- Issue: Synchronization operations will slow down code.

```c
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;
}
```
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.

  ![Diagram showing the relationship between different types of functions](image)

  - 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

- With deposit & withdraw example, check out why we need synchronization

```c
long long balance = 10000;

int main(void) {
    pthread_create(&tid[0], NULL, deposit_thread, 0)
    pthread_create(&tid[0], NULL, withdraw_thread, 0)

    pthread_join(tid[0], NULL)
    pthread_join(tid[1], NULL)

    printf("%lld\n", balance);
}
```

```c
void withdraw(int amount) {
    balance -= amount;
}
void deposit(int amount) {
    balance += amount;
}
void *withdraw_thread(void *arg) {
    for(i=0; i<400000000; i++)
        withdraw(1);
}
void *deposit_thread(void *arg) {
    for(i=0; i<400000000; i++)
        deposit(1);
}
```