SWE2024

SUNG KYUN KWAN UNIVERSITY

Synchronization I

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Warm Up Example

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

int num;

void *inc (void *tid) {
	int iter = 10000;

	while(iter--) num++;
}

void *dec (void *tid) {
	int iter = 10000;

	while(iter--) num--;
}
```

```c
int main() {
   int t;

   for(t = 0; t < 10; t++) {
      num = 0;
      pthread_t thread_inc, thread_dec;
      pthread_create(&thread_inc, NULL, &inc, NULL);
      pthread_create(&thread_dec, NULL, &dec, NULL);

      pthread_join(thread_inc, NULL);
      pthread_join(thread_dec, NULL);

      printf("#%d, %d\n", t, num);
   }
   return 0;
}
```

Warm Up Example

```
#0, -2759
#1, 1200
#2, -10000
#3, 0
#4, 10000
#5, -264
#6, -584
#7, -662
#8, -5587
#9, -3864
```

Instruction-level Timeline

num = 0

T1 executes num++

T2 executes num--

```
mov rax, qword ptr [rbp - 8]
mov ecx, dword ptr [rax]
```

ecx = 0

```
                                 mov rax, qword ptr [rbp - 8]
                                 mov ecx, dword ptr [rax]
                                 sub ecx, 1
                                 mov dword ptr [rax], ecx
```

ecx = 0

ecx = -1

```
add ecx, 1
mov dword ptr [rax], ecx
```

ecx = 1

Mutex

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

Mutex Creation / Termination API

- Static initialization
  - `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- Dynamic initialization
  - `pthread_mutex_t m;`
  - `pthread_mutex_init(&m, (pthread_mutexattr_t*)NULL);`
- Mutex termination
  - `pthread_mutex_destroy(&m);`
  - Destroy mutex object
    - Free resources it might hold

Mutex Operation APIs

- `int pthread_mutex_lock (pthread_mutex_t *m)`
  - Acquire lock on specified mutex variable
  - If the **mutex** is already locked by another thread, block calling thread until **mutex** is unlocked
- `int pthread_mutex_unlock (pthread_mutex_t *m)`
  - Unlock **mutex** if called by owning thread
- `int pthread_mutex_trylock (pthread_mutex_t *m)`
  - Attempt to lock **mutex**
  - If the **mutex** is already locked, return immediately with "busy" error code

Mutex Example

```
int num;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *inc (void *tid) {
	int iter = 10000;

	while(iter--) {
		pthread_mutex_lock(&m);
		num++;
		pthread_mutex_unlock(&m);
	}
}

void *dec (void *tid) {
	int iter = 10000;

	while(iter--) {
		pthread_mutex_lock(&m);
		num--;
		pthread_mutex_unlock(&m);
	}
}
```

```
#1 0
#2 0
#3 0
#4 0
#5 0
#6 0
#7 0
#8 0
#9 0
#10 0
```

Condition Variables (CV)

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

Operation Mechanism of CV

- How condition variables work
  - Thread locks mutex associated with condition variable
  - Thread tests condition to see if it can proceed
  - If it can
    - Your thread does its work
    - Your thread unlocks mutex
  - If it cannot
    - The thread sleeps & mutex is automatically released
    - Some other threads signal condition variable
    - Your thread wakes up from waiting with mutex automatically locked, and it does its work
    - Your thread releases mutex when it is done

CV Creation / Termination API

- Static initialization
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Dynamic initialization
  - `pthread_cond_t cond;`
  - `pthread_cond_init(&cond, (pthread_condattr_t*)NULL);`
- Condition variable termination
  - `pthread_cond_destroy(&cond);`
  - Destroy condition variable object
    - Free resources it might hold

CV Operation APIs

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

Condition Variable Example

```
action() {
    pthread_mutex_lock(&mutex);
    while(x != 0)
        pthread_cond_wait(&cond, &mutex);
    real_action();
    pthread_mutex_unlock(&mutex);
}
```

```
counter() {
    pthread_mutex_lock(&mutex);
    x--;
    if(x == 0)
        pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Thread Safety

- Thread-safe
  - Functions called from 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 access invocations
    - Class 3: Returning pointer to static variable
    - Class 4: Calling thread-unsafe functions

Thread Safety 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 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 */
static unsigned int next = 1;

int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) %
32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

```
/* rand - return pseudo-random integer on
0..32767 */

int rand_r(int *nextp) {
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp/65536) %
32768;
}
```

Thread Safety Class #3

- Returning pointer to static variable
  - Case #1
    - Fix: rewrite code so caller passes pointer to struct
    - Issue: require changes in caller and callee
  - Case #2
    - Fix: lock-and-copy
    - Issue: require 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);
```

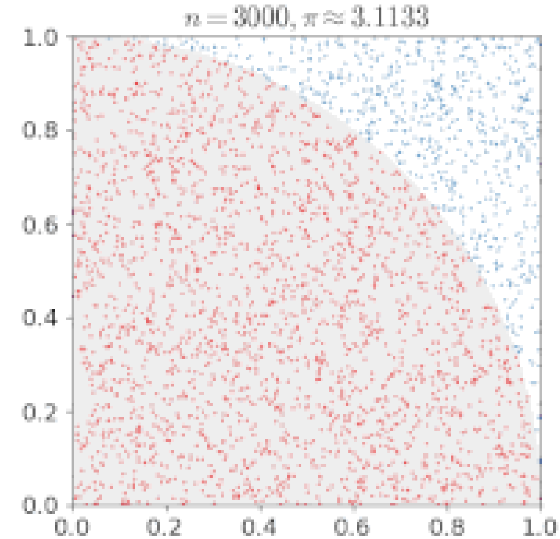
```
struct hostent
*gethostbyname_ts(char *name) {
  struct hostent *unshared
                       = malloc(...);
  pthread_mutex_lock(&lock); /* lock */
  shared = gethostbyname(name);
  *unshared = *shared;    /* copy */
  pthread_mutex_unlock(&lock);
  return unshared;
}
```

Thread Safety Class #4

- Calling thread-unsafe functions
  - Calling one thread-unsafe function makes entire function thread-unsafe
  - Fix: modify function so it calls only thread-safe functions

Exercise

- Monte Carlo method



- Single thread version

```
count = 0
r = 1

while i = 1~point_total
        x = random number between 0 and 1
        y = random number between 0 and 1

        if (x,y) within circle
                count += 1

pi = 4 * count / point_total
```

Exercise

- The parallel version gets two command line arguments
  - The number of threads
  - The number of random points per thread
- Main thread collects results from created threads and calculates value for pi
  - Each created thread must create random points and count the number of points that are inside circle
- Copy the skeleton code to your directory

$ cp ~swe2024-41_23s/2023s/p13_skeleton.c ./

```
❯ ./p13 10 10000000
pi: 3.142123
```

Exercise

- Submit your exercise source code
  - To InUiYeJi Cluster
  - Put your Makefile and *.c files in p13 folder
  - Submit using

  $ ~swe2024-41_23s/bin/submit p13 p13
  - We will compile by using command *make*
    - When compilation fails, you get zero points
    - Compiled binary name should be “*p13*“

- Due 2023/5/24 23:59