Synchronization: Intro and Mutex

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- How do we prevent two threads from racing?
- How do we work with pthread_mutex_t?
- What’s important to get right about locks?

Synchronization

- ***Synchronization***
  - coordinating execution among different threads
  - *Careful* synchronization avoids difficult-to-debug races.
  - Races are *hard* because:
    - Problems don’t always manifest (some very rare)
    - You must reason about multiple threads
- We’ll learn synchronization primitive
  - locks (mutex)
  - condition variables (next slides)
  - semaphores (next slides)

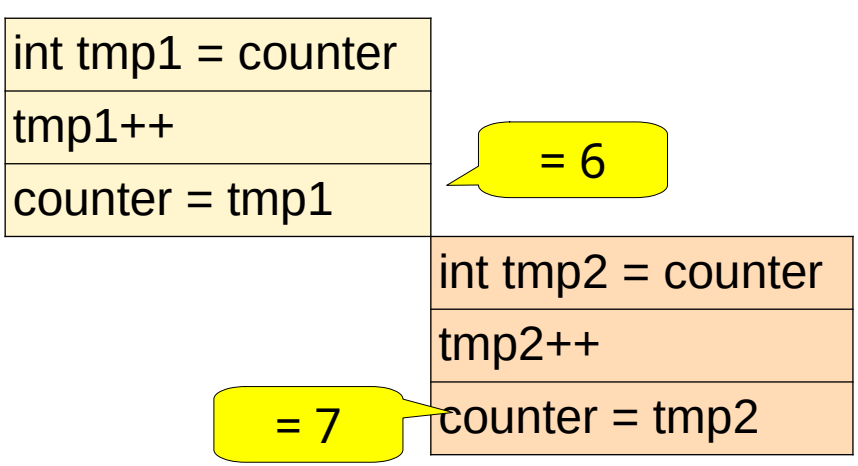
Details

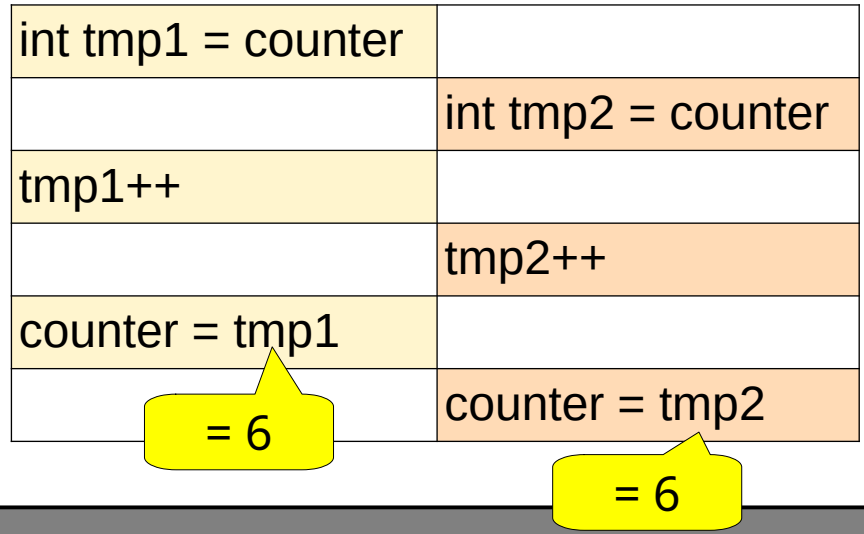
- Can find more info in OSTEP book (more depth than we require)
  - Chapter 28 Locks
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf
  - Chapter 30 Condition Variables
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf
  - Chapter 31 Semaphores
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf
  - Chapter 32 Concurrency Bugs
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf

Locks:
Mutexes

Motivation

- *Recall this race (counter starts at 5)*
  - What looks like one operation can actually be many sub-operations
  - We need to prevent this mix-up of sub-operations from different threads
  - Use a lock or a **mutex:** for **mut**ual **ex**clusion





Locks

- *Lock mechanisms consist of:*
  - A way to declare the lock variable, to create the lock
  - `lock()` function, to acquire it
  - `unlock()` function, to release it
- *E.g., pthread library's lock:*
  - *Define lock:*
    ```
    pthread_mutex_t myLock = PTHREAD_MUTEX_INITIALIZER;
    ```
  - *Mutex `lock()` function:*
    ```
    int pthread_mutex_lock(pthread_mutex_t *mutex)
    ```
  - Mutex unlock function:
    ```
    int pthread_mutex_unlock(pthread_mutex_t *mutex)
    ```

  Other languages (e.g., Java, Python, etc.) have similar lock mechanisms.

pthread Example

- Locks guarantee only a single thread will hold a lock

```c
static pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;
static int data[10];

static void *thread0(void *arg) {
    int count = 0;

    pthread_mutex_lock(&data_mutex);
    {

        for (int i = 0; i < 10; i++) {
            count += data[i];
        }

    }
    pthread_mutex_unlock(&data_mutex);
    printf("Sum is %d\n", count);
    pthread_exit(0);
}
```

```c
static void *thread1(void *arg) {


    pthread_mutex_lock(&data_mutex);


{
    for (int i = 0; i < 10; i++) {
        data[i] += 1;
    }
}
pthread_mutex_unlock(&data_mutex);
printf("Done update!\n");
pthread_exit(0);
}
```

T0 locks mutex

T1 tries to lock mutex

Mutex is locked so lock() blocks thread until mutex is free

T0 access data[]

Unblocks

T0 unlocks mutex. This unblocks T1

Operation of Lock

- *`pthread_mutex_lock(&mutex)` either:*
  - *if it's free,* locks mutex and returns immediately, or
  - blocks, then *once it's free,* locks the mutex and returns
- *Mutual Exclusion*
  - Even if multiple threads *call* ..`lock()` at once, only a single thread can hold a lock: all other threads wait
  - We cannot control the order in which threads grab the lock; it depends on the underlying locking mechanism (often the scheduler, also)
- *Non-deterministic*
  - This behaviour is **non-deterministic:** it exhibits different behaviour when run with the same input
  - (As opposed to deterministic behaviour)

Audience Particiation - Code with Data Race

```c
int cnt = 0;

static void *thread_func(void *arg) {
    for (int i = 0; i < 10000000; i++)
        cnt++;
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t t1;
    pthread_t t2;

    pthread_create(&t1, NULL, thread_func, NULL);
    pthread_create(&t2, NULL, thread_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("%d\n", cnt);

    exit(EXIT_SUCCESS);
}
```

- This code has a data race. What is it?

a) **T2** may start before **T1**

b) **T2** may end before **T1**

c) **T1** and **T2** share **cnt**

d) **T1** and **T2** share **i**

Code with Error Checking

```c
int cnt = 0;

static void *thread_func(void *arg) {
    for (int i = 0; i < 10000000; i++)
        cnt++;
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t t1;
    pthread_t t2;

    if (pthread_create(&t1, NULL, thread_func, NULL) != 0)
        perror("pthread_create");

    if (pthread_create(&t2, NULL, thread_func, NULL) != 0)
        perror("pthread_create");

    if (pthread_join(t1, NULL) != 0)
        perror("pthread_join");
    if (pthread_join(t2, NULL) != 0)
        perror("pthread_join");

    printf("%d\n", cnt);

    exit(EXIT_SUCCESS);
}
```

This is the same code as previous slide,
but shows error checking on functions.

You should do this!
(Slides omit for brevity)

Mutex Protected

```c
int cnt = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static void *thread_func(void *arg) {
    for (int i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t t1;
    pthread_t t2;

    pthread_create(&t1, NULL, thread_func, NULL);
    pthread_create(&t2, NULL, thread_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("%d\n", cnt);
    exit(EXIT_SUCCESS);
}
```

- Protect the critical section with a lock!
- A thread trying to change `cnt` must do so with mutex locked
- `man pthread_mutex_lock`
- *Why not lock outside the loop?*

Lock Usage

Atomicity

- *Atomicity*
  - **Atomic:** operations run as if they are a single operation; *indivisible*
  - Cannot be interfered with by other sections with *same* lock
  - Mutex lock makes a section of code atomic
  - **Atomicity:** *all or nothing* either all operations are run, or none
- *Serialization and interleaving*
  - Lock effectively **serializes** operations: only one thread at a time can run operations guarded by a lock
  - Operations from different threads are **interleaved** in some order
  - We *cannot control* the order in which different threads run!

Protecting Shared Variables

- *Can have a data race when threads share a variable*
  - e.g. Accessing same global variable: `cnt++`
  - e.g. Accessing same memory via a pointer: `pSharedBuffer[i] = 52;`
- *Solve data race with a lock*
  - Controls and serializes access shared variable
- *Where in the code?*
  - Data race may be from one piece of code!
    - e.g.: a function called by multiple threads tracking next free block to allocate
  - May be in different sections of code, each using the same lock
    - e.g.: one thread fills buffer, another thread empties buffer

Multiple Lock

- *Can have multiple locks*
  - if they are protecting independent shared variables
  - e.g.: `data_samples_mutex`, `printer_mutex`
  - Each code section / thread locks the mutex(es) it needs to lock be safe
  - Reducing *lock contention* is important for performance (reduces parallelism!)

Non-Blocking Lock

- *Options to allow us to control blocking behaviour*
  - `pthread_mutex_trylock()`: returns immediately if unable to lock
  - `pthread_mutex_timedlock()`: waits a maximum amount of time before returning if unable to lock

Critical Section (CS)
and
Thread Safety

Critical Sections (CS)

- ***Critical Section:***
  - A critical section is a piece of code that (from OSTEP)
    - *accesses a shared variable* (or more generally, a shared resource) and
    - must not be concurrently executed by more than one thread
- *Rephrased*
  - If a thread is executing the CS, no other threads should execute the CS

Critical Sections (CS)

- *An ideal solution for CS problem must satisfy 3 requirements:*
  - **Mutual exclusion:** Only one thread should be allowed to run in the CS
  - **Progress:** A thread should eventually complete (i.e., make progress)
  - **Bounded waiting:** An upper bound must exist for the amount of time a thread waits to enter the CS
    - i.e., a thread should only be blocked for a finite amount of time

Thread Safety & Reentrance

- ***Thread safe function:***
  - a function that multiple threads can run safely.
  - It either:
    - does not access shared resources, or
    - provides proper protection for critical sections that access shared resources
- ***Reentrant** vs **nonreentrant** functions* (related concept)
  - A reentrant function is a function that produces the correct output even when called again while executing
  - Must work with different threads (thread safe), and *also the same thread* (such as in a signal handler)
  - i.e., a function called by `main()` might also be called by a signal handler on the same thread

Audience Participation - Thread Safety

- How thread safe is this function?
- Analysis:
  - Not thread safe: shared variable overwritten by each call.
  - Therefore not reentrant.

```
int tmp = 0;

int swap(int *pA, int *pB) {
    tmp = *pA;
    *pA = *pB;
    *pB = tmp;
}
```

a) Thread safe: YES Reentrant YES
b) Thread safe: YES Reentrant NO
c) Thread safe: NO Reentrant YES
d) Thread safe: NO Reentrant NO

Audience Participation - Thread Safety

- How thread safe is this function?
- Analysis:
  - Is thread safe: multiple threads will block.
  - Not reentrant: if threads gets interrupted by a signal while holding mutex then signal handler will block.

```
int tmp = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int swap(int *pA, int *pB) {
    pthread_mutex_lock(&mutex);
    tmp = *pA;
    *pA = *pB;
    *pB = tmp;
    pthread_mutex_unlock(&mutex);
}
```

a) Thread safe: YES Reentrant YES

b) Thread safe: YES Reentrant NO

c) Thread safe: NO Reentrant YES

d) Thread safe: NO Reentrant NO

Audience Participation - Thread Safety

- How thread safe is this function?
- Analysis:
  - Is thread safe: no shared data
  - Is reentrant: no saved or shared data

```
int swap(int *pA, int *pB) {
    int tmp = 0;

    tmp = *pA;
    *pA = *pB;
    *pB = tmp;

}
```

a) Thread safe: YES Reentrant YES

b) Thread safe: YES Reentrant NO

c) Thread safe: NO Reentrant YES

d) Thread safe: NO Reentrant NO

Making Functions Reentrant

- *What makes a function non-reentrant?*
  - A function might work with some data, like a buffer:
    - use a shared global buffer
    - use a shared thread-local buffer
- *Solutions*
  - allocate its own local variable buffer on the stack
  - dynamically allocate and free new buffer in the heap
  - have calling code allocate space and pass it in
- *Caller Allocates Technique*
  - Many functions make calling code pass in the buffer, e.g., `write()`
  - Any space returned to caller or maintained across function calls is allocated by the caller.

Deadlock and Livelock

Deadlock

- *Deadlock*
  - a condition where a set of threads each hold a resource and wait to acquire a resource held by another thread
  - The threads get stuck and make no progress
- *Example*
  - Create mutex locks A & B;
  - *Thread **1**:* locks **A**
  - *Thread **2**:* locks **B**, then blocks trying to lock **A**
  - *Thread **1**:* blocks trying to lock **B**

Deadlock Activity

- Write a program that [15m]
  - creates two threads, two locks:

```
Lock A        Lock B
Print         Print
Lock B        Lock A
Print         Print
Unlock B      Unlock A
Unlock A      Unlock B
Print         Print
```

Reference code:

```c
#include <pthread.h>
static void *func(void *arg) {
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t t1;

    pthread_create(&t1, NULL, func, NULL);

    pthread_join(t1, NULL);
}
```

- *Investigation*
  - Does it always finish (run multiple times)?
  - Does it always not finish (run multiple times)?
  - What happens if both threads lock A and B in the same order?

Necessary Conditions for Deadlock

- *4 conditions are necessary for deadlock*
  - These do not *guarantee* deadlock: deadlock also depends on timing of thread execution

  1) **Hold and wait:** threads are *already holding resources,* but also are *waiting for additional resources* being held by other threads
  2) **Circular wait:** there exists a set $\{T_0, T_1, \ldots, T_{n-1}\}$ of threads such that
     - T_0 is waiting for a resource that is held by T_1;
     - T_1 is waiting for T_2, ..., T_{n-1} is waiting for T_0
  3) **Mutual exclusion:** threads hold resources exclusively
  4) **No preemption:** resource released only voluntarily by the thread holding it

Apply Deadlock Conditions

- *E.g.:* Thread 1 Thread 2

```
Lock A      Lock B
Print       Print
Lock B      Lock A
Print       Print
Unlock B    Unlock A
Unlock A    Unlock B
Print       Print
```

- *4 Conditions to Check*
  - Hold and wait? **Check!**
  - Circular wait? **Check!**
  - Mutual Exclusion? **Check!**
  - No preemption? **Check!**

All 4 conditions hold. Therefore, it's POSSIBLE to have deadlock.

- *Deadlock Prevention*
  - Break one of these for conditions to prevent deadlocks.

Preventing Deadlocks

- *Technique 1:* Grab all locks at once, atomically
  - Breaks hold-and-wait condition: you grab *all the locks* together, or *no locks* at all

```
static pthread_mutex_t mutex0 = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t another_lock = PTHREAD_MUTEX_INITIALIZER;

static void *thread0(void *arg) {
    pthread_mutex_lock(&another_lock);
    {
        pthread_mutex_lock(&mutex0);
        printf("thread0: mutex0\n");
        pthread_mutex_lock(&mutex1);
    }
    pthread_mutex_unlock(&another_lock);

    printf("thread0: mutex1\n");
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex0);
    pthread_exit(0);
}
```

Preventing Deadlocks

- *Technique 2:* Acquire locks in *same order*
  - Acquiring locks in the same global order for all threads breaks the *circular wait condition*

```
static pthread_mutex_t mutex0 = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;


static void *thread0(void *arg) {
    pthread_mutex_lock(&mutex0);
    printf("thread0: mutex0\n");

    pthread_mutex_lock(&mutex1);
    printf("thread0: mutex1\n");

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex0);
    pthread_exit(0);
}
```

Livelock

- ***Livelock:***
  - where a set of threads each execute instructions actively, but they still don't make any progress
- *E.g.: Threads T0 and T1*
  - Each attempts to acquire two resources R0 and R1

```
while (true)
    Acquire R0
    if R1 is free, then
        Acquire R1
        do work
        Free R1, R0
        return
    else
        Free R0
```

```
while (true)
    Acquire R1
    if R0 is free, then
        Acquire R0
        do work
        Free R0, R1
        return
    else
        Free R1
```

  - Problem: T0 and T1 run concurrently, each locking the first resource then trying to lock the second
  - Each frees first resource, and then tries again forever.

Livelock vs. Deadlock

```
while (true)
    Acquire R0
    if R1 is free, then
        Acquire R1
        do work
        Free R1, R0
        return
    else
        Free R0
```

```
while (true)
    Acquire R1
    if R0 is free, then
        Acquire R0
        do work
        Free R0, R1
        return
    else
        Free R1
```

- *Livelock*
  - Thread 0 and Thread 1 actively execute code but do not make any progress
- *Deadlock vs Livelock*
  - Both deadlocks and livelocks *do not make any progress;* in a livelock scenario, threads do still execute (busy-waiting)
  - In a deadlock scenario, threads are stuck and do not execute anything actively

Audience Participation - Identification

- What synchronization problem is present in this code with two threads (left and right), where M0 and M1 are mutexes.

```
global int cnt = 0;

while (true):
    lock M0

    if cnt % 2 == 1 then:
        lock M1
        cnt++
        unlock M1

    unlock M0
```

```
while (true):
    lock M0

    if cnt % 2 == 0 then:
        lock M1
        cnt++
        unlock M1

    unlock M0
```

a) Race
b) Non-reentrant
c) Livelock
d) Deadlock

Summary

- *Mutex*
  - Used for Mutual Exclusion from a critical section
  - Guarantees only one thread can hold the lock
- *Critical Section*
  - Area of the code which accesses a shared variable that must not be concurrently accessed from another thread
- *Thread safe:* Correctly runs with multiple threads
- *Reentrant:* Correctly runs when called again while running (same thread?)
- *Deadlock:* Two threads blocking each other. Necessary conditions:
  - Hold and wait
  - Circular wait
  - Mutual exclusion
  - No preemption