Synchronization: Intro & Mutex

Topics

- How can we prevent two threads form having a race case?
- How can we code a mutex in C?
- What’s important to get right about locks?

Intro

- Synchronization
  ..
  - *Careful* synchronization avoids difficult to debug race cases.
  - Race cases are *hard* because:
    - ..
    - ..
      not just single path's correctness.
- We'll learn synchronization primitives:
  - locks (mutex)
  - condition variables (next slide deck)
  - semaphores (next slide deck)

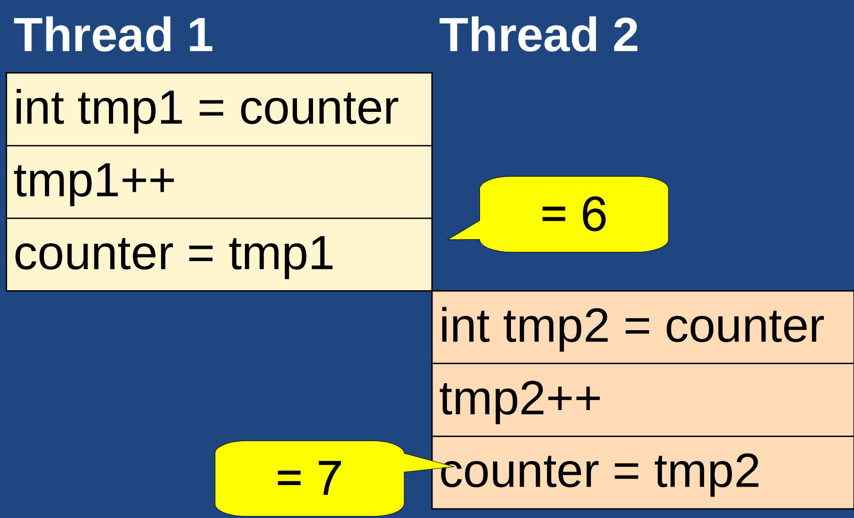
Details

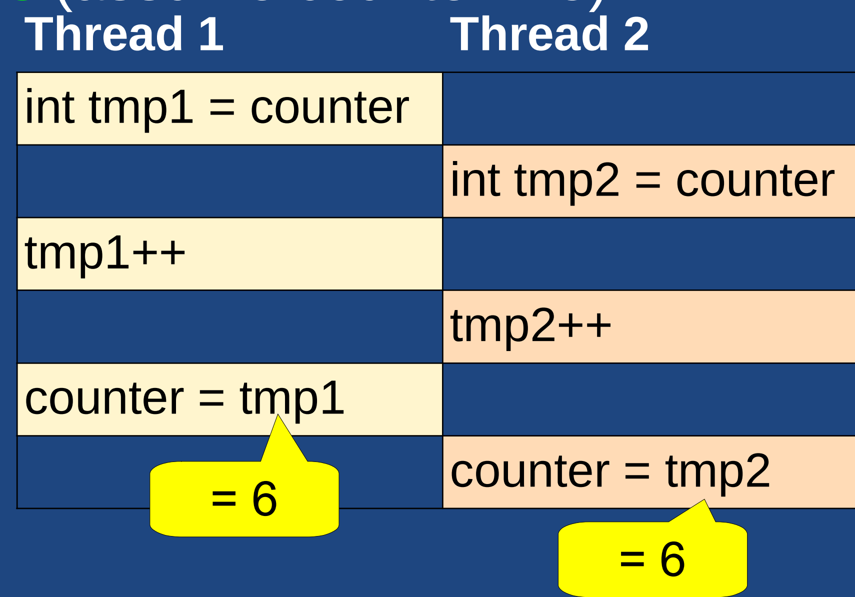
- Can find more info in OSTEP book
  (more depth than we require)
  - Chapter 28 Locks
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf
  - Chapter 30 Condition Variables
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf
  - Chapter 31 Semaphores
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf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

Locks:
Mutexes

Motivation

- Recall race case from Threads notes (assume counter = 5):





- What looks like one operation ..
  - We need to prevent this mix-up of sub-operations from different threads.
  - Use a lock or a mutex: ..

Locks

- Lock mechanisms consists of:
  - ..
  - .. function that grabs a lock
  - .. function that releases a lock
- 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: ..

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

T0 locks mutex

T0 tries to lock mutex

T0 access data[]

T0 unlocks mutex. This unblocks T0

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


    pthread_mutex_lock(&data_mutex);







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

Operation of Lock

- pthread_mutex_lock(&mutex) either:
  a) ..
  b) ..
- Mutual Exclusion
  - Even if multiple threads call lock() at once,
    ..
    all other threads wait
  - We cannot control the order in which threads grab the lock.
    It depends on the underlying lock mechanism.
- Non-deterministic
  - This behaviour is **non-deterministic**:
    ..
  - Opposed of **deterministic** behaviour.

ABCD: Code with Data Race

```
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 suffers a data race.

What is the cause of this data race?

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

```
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:

    ..

    Cannot be interfered with by other sections with same lock.
  - Mutex lock makes a section of code atomic.
  - Atomicity: **all or nothing** as it runs either
    all operations or no operations at all.
- Serialization and interleaving
  - Lock effectively serializes operations:

    ..
  - 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.. cnt++
  - e.g. Accessing same..
    `pSharedBuffer[i] = 52;`
- Solve data race with a lock
  - Controls and serializes access shared variable
- Where in the code?
  - Data race may be..
    e.g.: One function called by multiple threads
    tracking next free block to allocate.
  - May be in..

    thread fills buffer, one thread empties buffer.

Multiple locks

- 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.

Non-Blocking Lock

- Options to allow us to control blocking behaviour:
  - pthread_mutex_trylock()

    ..
  - pthread_mutex_timedlock()

    waits a maximum amount of time before returning if unable to lock.

Critical Section (CS)
and
Thread Safety

Critical Section (CS)

- Critical Section:

  A critical section is a piece of code that
  ..
  (or more generally, a shared resource) and
  ..
  -- From OSTEP

- Rephrased:
  - If a thread is executing the CS,
    no other threads should execute the CS.

Critical Section (CS)

- An ideal solution for CS problem must satisfy 3 requirements:
  - Mutual exclusion

    ..
  - Progress

    ..
  - Bounded waiting

    ..

    i.e., a thread should only be blocked for a finite amount of time.

Thread safety & Reentrant

- Thread safe function

  ..

  It either:

  a) does not access shared resources or

  b) provides proper protection for CS that access shared resources.

- Reentrant vs nonreentrant functions (related concept)
  - A reentrant function is a function that

    ..

  - Must work with different threads (thread safe), and also

    ..

  - i.e., a function called by main() might also be called by a signal handler on the same thread.

ABCD: Thread safety (1)

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

- How thread safe is this function?

```
int tmp = 0;

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

ABCD: Thread safety (2)

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

- How thread safe is this function?

```
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);
}
```

ABCD: Thread safety (2)

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

- How thread safe is this function?

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

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

}
```

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()
  - ..

Deadlock and Livelock

Deadlock

- Deadlock
  - a condition where a set of threads
  - ..
  - The threads get stuck and make no progress.
- E.g.:
  - 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

- [15 min]
  Write a program that creates two threads and two locks:

Thread #0:

Lock A
Print
Lock B
Print
Unlock B
Unlock A
Print

Thread #1:

Lock B
Print
Lock A
Print
Unlock A
Unlock B
Print

Useful Thread Code

```
#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 <u>necessary</u> for deadlock:
  These do not guarantee deadlock:
  deadlock also depends on timing of thread execution.

1) Hold and wait:
   ..

2) ..
   there exists a set {T0, T1, ..., Tn-1} of threads such that
   T0 is waiting for a resource that is held by T1,
   T1 is waiting for T2, ..., Tn–1 is waiting for T0.

3) Mutual exclusion:
   ..

4) No preemption:
   resource released only voluntarily by the thread holding it

Apply Deadlock Conditions

- E.g.: Thread 1 Thread 2

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?
  - Circular wait?
  - Mutual Exclusion?
  - No preemption?

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:..
  - ..
    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);
}
```

```
static void *thread1(void *arg) {
    pthread_mutex_lock(&another_lock);
    {
        pthread_mutex_lock(&mutex1);
        printf("thread1: mutex1\n");
        pthread_mutex_lock(&mutex0);
    }
    pthread_mutex_unlock(&another_lock);

    printf("thread1: mutex0\n");
    pthread_mutex_unlock(&mutex0);
    pthread_mutex_unlock(&mutex1);
    pthread_exit(0);
}
```

Preventing Deadlocks

- Technique 2:..
  - Acquiring locks in the same global order for all threads:
    ..
    as all threads try to grab locks in the exact same order.

```
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);
}
```

```
static void *thread1(void *arg) {
    pthread_mutex_lock(&mutex0);
    printf("thread1: mutex0\n");

    pthread_mutex_lock(&mutex1);
    printf("thread1: 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..
- 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 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.
  - In a deadlock scenario,
    ..

ABCD: Identify the problem

- 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):                      while (true):
    lock M0                            lock M0

    if cnt % 2 == 1 then:              if cnt % 2 == 0 then:
        lock M1                            lock M1
        cnt++                              cnt++
        unlock M1                          unlock M1

    unlock M0                          unlock M0
```

a) Race case

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