CS 858: Software Security

Offensive and Defensive Approaches

Attacks: data race

Meng Xu *(University of Waterloo)*

Fall 2022

Outline

What is data race?

What is data race?

```
global var count = 0
```

```
for(i = 0; i < x; i++) {
   /* do sth critical */
   ......
   count++;
}
```

Thread 1

```
for(i = 0; i < y; i++) {
   /* do sth critical */
   ......
   count++;
}
```

Thread 2

Q: What is the value of `count` when both threads terminate?

What is data race?

```
global var count = 0
global var mutex = ⊥
```

Thread 1

```
for(i = 0; i < x; i++) {
   /* do sth critical */
   ......
   lock(mutex);
   count++;
   unlock(mutex);
}
```

Thread 2

```
for(i = 0; i < y; i++) {
   /* do sth critical */
   ......
   lock(mutex);
   count++;
   unlock(mutex);
}
```

Q: What is the value of `count` when both threads terminate?

Data race combined with memory errors

Data race combined with memory errors

`p` is a global pointer initialized to `NULL`

Thread 1	Thread 2

`if (!p) {`
`  p = malloc(128);`
`}`
	`if (!p) {`
	`  p = malloc(256);`
	`}`

Q: What are the possible outcomes of this execution?

Data race combined with memory errors

p is a global pointer initialized to `NULL`

```
if (!p) {
   p = malloc(128);
}

if (p) {
   free(p);
   p = NULL;
}
```

Thread 1

```
if (!p) {
   p = malloc(256);
}

if (p) {
   free(p);
   p = NULL;
}
```

Thread 2

Q: What are the possible outcomes of this execution?

Data race as heisenbug

Data race as heisenbug

Programs which contain data races usually demonstrate unexpected and even non-deterministic behavior.

Data race as heisenbug

Programs which contain data races usually demonstrate unexpected and even non-deterministic behavior.

- The outcome might depend on a specific execution order (a.k.a. thread interleaving).
- Re-running the program may not always produce the same results.

Data race as heisenbug

Programs which contain data races usually demonstrate unexpected and even non-deterministic behavior.

- The outcome might depend on a specific execution order (a.k.a. thread interleaving).
- Re-running the program may not always produce the same results.

Concurrent programs are hard to debug and even harder to ensure correctness.

Outline

An intuitive definition

Intuitively, a *data race* happens when:

1. There are two memory acceses from different threads.
2. Both acceses target the same memory location.
3. At least one of them is a write operation.

An intuitive definition

Intuitively, a *data race* happens when:

1. There are two memory acceses from different threads.
2. Both acceses target the same memory location.
3. At least one of them is a write operation.
4. Both acceses could interleave freely without restrictions such as synchronization primitives or causality relations.

Revisit the example

```
global var count = 0
```

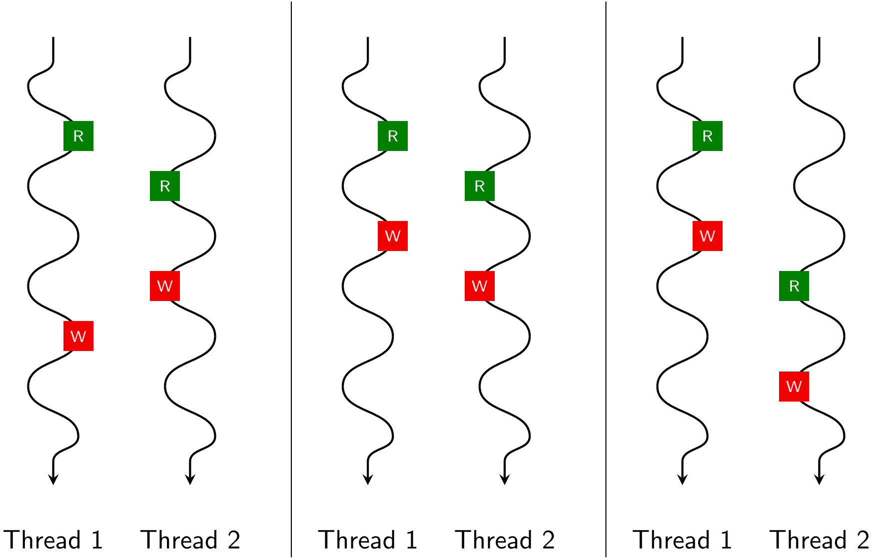
```
for(i = 0; i < x; i++) {
   count++;
}
```

Thread 1

```
for(i = 0; i < y; i++) {
   count++;
}
```

Thread 2

Free interleavings without locking



Revisit the example

```
global var count = 0
```

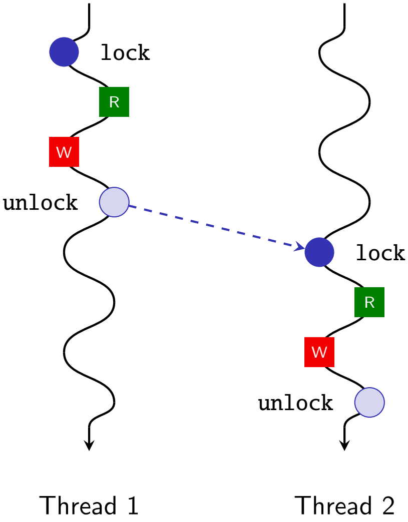
```
for(i = 0; i < x; i++) {
   lock(mutex);
   count++;
   unlock(mutex);
}
```

Thread 1

```
for(i = 0; i < y; i++) {
   lock(mutex);
   count++;
   unlock(mutex);
}
```

Thread 2

Limited interleavings with locking



Common synchronization primitives

Common synchronization primitives

- Lock / Mutex / Critical section
- Read-write lock
- Barrier
- Semaphore

Revisiting the definition

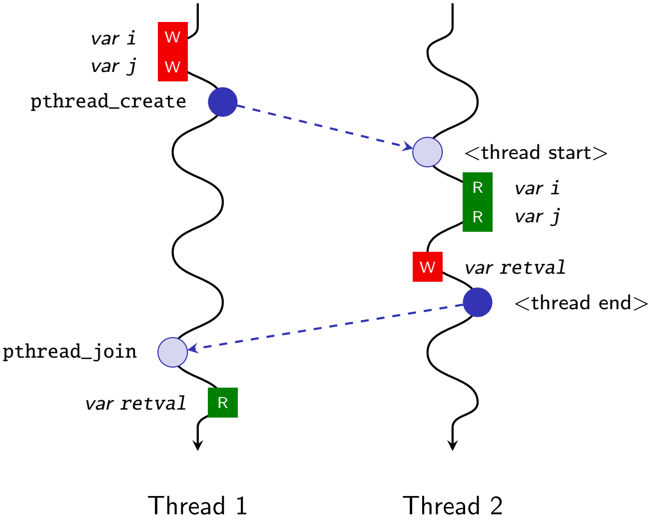
Intuitively, a *data race* happens when:

1. There are two memory acceses from different threads.
2. Both acceses target the same memory location.
3. At least one of them is a write operation.
4. Both acceses could interleave freely without restrictions such as synchronization primitives ~~**or causality relations**~~.

Causality relations: an example

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

int i;
int retval;

void* foo(void* p){
    printf("Value of i: %d\n", i);
    printf("Value of j: %d\n", *(int *)p);
    pthread_exit(&retval);
}

int main(void){
    int i = 1;
    int j = 2;

    pthread_t id;
    pthread_create(&id, NULL, foo, &j);
    pthread_join(id, NULL);

    printf("Return value from thread: %d\n", retval);
}
```

Causality relations



Wait..., how are synchronization primitives implemented?

Wait..., how are synchronization primitives implemented?

- Dekker's algorithm
- Atomic swap
- Atomic read-modify-write
    - compare-and-swap
    - test-and-set
    - fetch-and-add
    - ......

Dekker's algorithm

```
bool wants_to_enter[2] = {false, false};
int turn = 0;   /* or turn = 1 */
```

```
// lock
wants_to_enter[0] = true;
while (wants_to_enter[1]) {
    if (turn != 0) {
        wants_to_enter[0] = false;
        // busy wait
        while (turn != 0) {}
        wants_to_enter[0] = true;
    }
}

/* ... critical section ... */

// unlock
turn = 1;
wants_to_enter[0] = false;
```

Thread 1

```
// lock
wants_to_enter[1] = true;
while (wants_to_enter[0]) {
    if (turn != 1) {
        wants_to_enter[1] = false;
        // busy wait
        while (turn != 1) {}
        wants_to_enter[1] = true;
    }
}

/* ... critical section ... */

// unlock
turn = 0;
wants_to_enter[1] = false;
```

Thread 2

Bonus: Spinlock with atomic swap (`xchg`)

```
locked:                       ; The lock variable. 1 = locked, 0 = unlocked.
     dd      0

spin_lock:
     mov     eax, 1           ; Set the EAX register to 1.
     xchg    eax, [locked]    ; Atomically swap the EAX register with
                              ;  the lock variable.
                              ; This will always store 1 to the lock, leaving
                              ;  the previous value in the EAX register.
     test    eax, eax         ; Test EAX with itself. Among other things, this
                              ;  will set the processor's Zero Flag if EAX is 0.
                              ; If EAX is 0, then the lock was unlocked and
                              ;  we just locked it.
                              ; Otherwise, EAX is 1 and we didn't acquire the lock.
     jnz     spin_lock        ; Jump back to the MOV instruction if the Zero Flag is
                              ;  not set; the lock was previously locked, and so
                              ; we need to spin until it becomes unlocked.
     ret                      ; The lock has been acquired, return to the caller.

spin_unlock:
     xor     eax, eax         ; Set the EAX register to 0.
     xchg    eax, [locked]    ; Atomically swap the EAX register with
                              ;  the lock variable.
     ret                      ; The lock has been released.
```

Revisiting the definition (again)

If we can find, statically or dynamically, a pair of memory acceses (A_1, A_2) such that

- they originate from different threads,
- both A_1 and A_2 target the same memory location, AND
- at least one of them is a write operation,

then we conclude that (A_1, A_2) must be one of the following cases:

- (A_1, A_2) is part if a synchronization primitive, OR
- (A_1, A_2) is a data race.

Revisiting the definition (again)

If we can find, statically or dynamically, a pair of memory acceses (A_1, A_2) such that

- they originate from different threads,
- both A_1 and A_2 target the same memory location, AND
- at least one of them is a write operation,

then we conclude that (A_1, A_2) must be one of the following cases:

- (A_1, A_2) is part if a synchronization primitive, OR
- (A_1, A_2) is a data race.

Q: Is this definition good enough?

Is this a data race?

```
int x = 0;
bool flag = false;
lock mutex = unlocked;
```

```
x++;
lock(mutex);
flag = true;
unlock(mutex);
```

Thread 1

```
while(true) {
    lock(mutex);
    if (flag) {
        break;
    }
    unlock(mutex);
}
x--;
```

Thread 2

Is this a (bad) data race?

```
int x = 0;
bool flag = false;
```

```
x++;
flag = true;
```

Thread 1

```
while (!flag) {};
x--;
```

Thread 2

Outline

How to model concurrency mathematically?

How to model concurrency mathematically?

- Lamport clock
- Vector clock

Lamport clock algorithm

Each thread has its own clock variable t

- On initialization:
  - $t \leftarrow 0$
- On write to shared memory `*ptr = val`:
  - $t \leftarrow t + 1$
  - store t alongside `val` at memory location `ptr`
- On read from shared memory `val = *ptr`:
  - retrieve the stored clock t' at memory location `ptr`
  - $t \leftarrow \max(t, t') + 1$

Lamport clock algorithm

Each thread has its own clock variable t

- On initialization:
  - $t \leftarrow 0$
- On write to shared memory `*ptr = val`:
  - $t \leftarrow t + 1$
  - store t alongside `val` at memory location `ptr`
- On read from shared memory `val = *ptr`:
  - retrieve the stored clock t' at memory location `ptr`
  - $t \leftarrow \max(t, t') + 1$

Properties of Lamport clock:

- $a \rightarrow b \implies L(a) < L(b)$
- $L(a) < L(b) \not\Longrightarrow a \rightarrow b$

Vector clock algorithm

Each thread i has its own clock vector t

- On initialization:
  - $T \leftarrow \langle 0, 0, \ldots, 0 \rangle_N$, assuming N threads
- On write to shared memory `*ptr = val`:
  - $T[i] \leftarrow T[i] + 1$
  - store T alongside `val` at memory location `ptr`
- On read from shared memory `val = *ptr`:
  - retrieve the stored clock T' at memory location `ptr`
  - $\forall k \in [0, N) : \; T[k] = \max(T[k], T'[k])$
  - $T[i] \leftarrow T[i] + 1$

Properties of the vector clock algorithm

With the following definition on the timestamp ordering:

- $T = T' \iff \forall i \in [0, N): \; T[i] = T'[i]$
- $T \leq T' \iff \forall i \in [0, N): \; T[i] \leq T'[i]$
- $T < T' \iff T \leq T' \land T \neq T'$
- $T \parallel T' \iff T \nleq T' \land T' \nleq T$

We have:

- $a \rightarrow b \iff V(a) < V(b)$
- $a = b \iff V(a) = V(b)$
- $a \parallel b \iff V(a) \parallel V(b)$

Homework exercise

```
int x = 0;
bool flag = false;
```

```
x++;
flag = true;
```

Thread 1

```
while (!flag) {};
x--;
```

Thread 2

Prove (by hand) that the write of `x` at `x--` in thread 2 can never happen before the read of `x` in `x++` in thread 1.

Outline

A more abstract view of data race

Q: Why data race can happen in the first place?

A more abstract view of data race

Q: Why data race can happen in the first place?

A: Because two threads in the same process share memory.

A more abstract view of data race

Q: Why data race can happen in the first place?

A: Because two threads in the same process share memory.

We can further generalize this concept by asking:

Q: What else do they share?
Q: What about other entities that may run concurrently?

And the answer to these questions will help define race condition.

Example: race over the filesystem

```c
#include <...>

int main(int argc, char *argv[]) {
    FILE *fd;
    struct stat buf;

    if (stat("/some_file", &buf)) {
        exit(1); // cannot read stat message
    }

    if (buf.st_uid != getuid()) {
        exit(2);  // permission denied
    }

    fd = fopen("/some_file", "wb+");
    if (fd == NULL) {
        exit(3);  // unable to open the file
    }

    fprintf(f, "<some-secret-value>");
    fclose(fd);
    return 0;
}
```

Example: the Dirty COW exploit

CVE-2016-5195

Allows local privilege escalation: `user(1000)` → `root(0)`.

Exists in the kernel for nine years before finally patched.

Details on the Website.

$\langle$ **End** $\rangle$