CS 489 / 698: Software and Systems Security

Module: Common Vulnerabilities
Lecture: race conditions

Meng Xu *(University of Waterloo)*

Fall 2024

Outline

What is a race condition?

Wikipedia's definition

A race condition is the condition of a software system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events, leading to unexpected or inconsistent results.

It becomes a bug when one or more of the possible behaviors is undesirable.

Wikipedia's definition

A race condition is the condition of a software system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events, leading to unexpected or inconsistent results.

It becomes a bug when one or more of the possible behaviors is undesirable.

What is a data race?

Data race definition in C++ standard

When

- *an evaluation of an expression writes to a memory location* **and**
- *another evaluation reads or modifies the same memory location,*

the expressions are said to conflict.

A program that has two conflicting evaluations has a data race unless:

- *both evaluations execute on the same thread,* **or**
- *both conflicting evaluations are atomic operations,* **or**
- *one of the conflicting evaluations happens-before another.*

Adapted from a community-backed C++ reference site. For the full version, please refer to the related sections in C++ working draft.

An intuitive definition

Intuitively, a *data race* happens when:

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

Test of your understanding

Q: Based on the definition of race condition and data race, what do you think are the relationship between them?

Outline

Introductory case

```
global var count = 0
```

Thread 1:

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

Thread 2:

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

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

Introductory case

global var count = 0

Thread 1:

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

Thread 2:

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

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

Introductory case

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

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

Thread 1

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

Thread 2

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

Race conditions in other settings

Race conditions are not tied to a specific programming language, instead, they are tied to data sharing in concurrent execution.

Race conditions in other settings

Race conditions are not tied to a specific programming language, instead, they are tied to data sharing in concurrent execution.

For example, in the database context:

Q: If two database clients send the following requests concurrently, what will be the result (both try to withdraw $100 from Alice)?

Client 1

```
SELECT @balance = Balance
  FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance =
  @balance - 100 WHERE Name = "Alice";
```

Client 2

```
SELECT @balance = Balance
  FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance =
  @balance - 100 WHERE Name = "Alice";
```

Race conditions in a database setting

One possible interleaving (that messes up the states)

```
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```

Race conditions in a database setting

One possible interleaving (that messes up the states)

```
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```

Q: How to prevent the race condition in this case?

Race conditions in a database setting

One possible interleaving (that messes up the states)

```
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```

Q: How to prevent the race condition in this case?

Interleavings with transactions

```
BEGIN TRANSACTION;
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
BEGIN TRANSACTION;
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
```

Revisit the example

```
global var count = 0
```

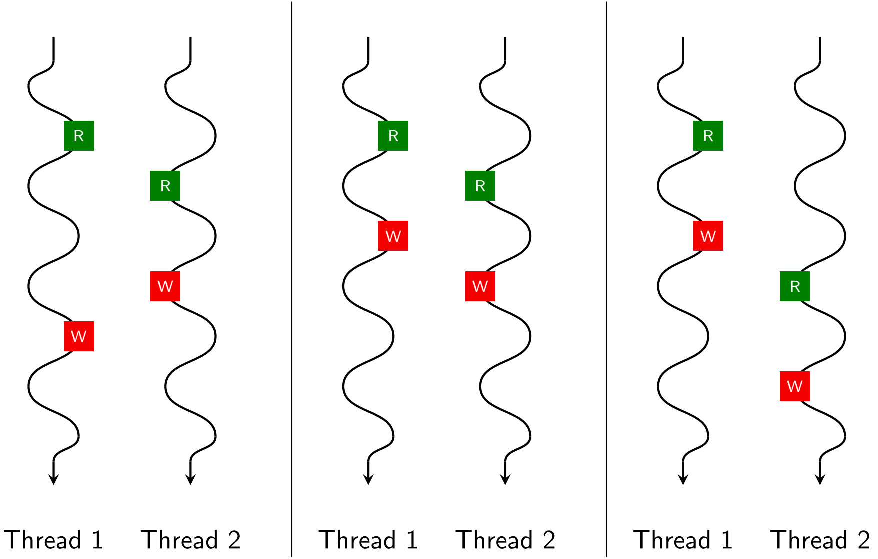
Thread 1:

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

Thread 2:

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

Free interleavings without locking



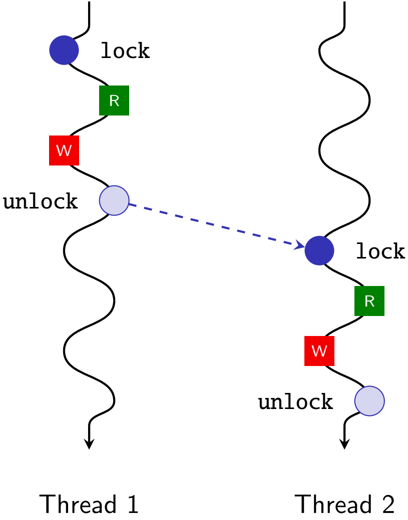
Revisit the example

```
global var count = 0
```

Thread 1	Thread 2

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

Limited interleavings with locking



Revisiting the definition

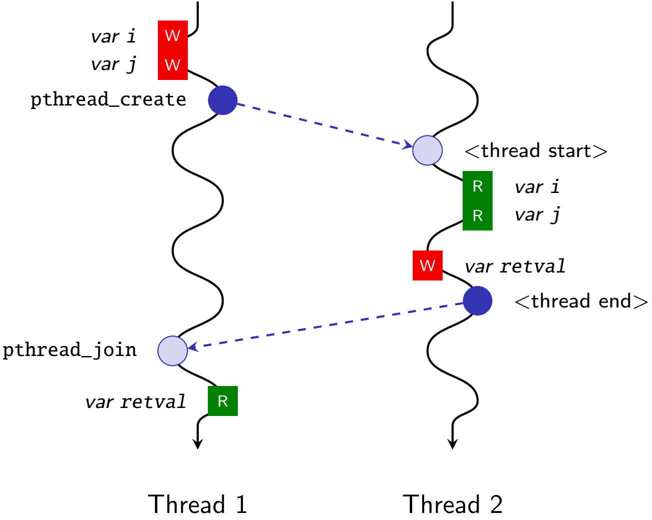
Intuitively, a *data race* happens when:

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

Causality relations: an example

```
#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



Outline

Revisiting the definition

If we can find, statically or dynamically, a pair of memory access instructions (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:

1. A_1 strictly happens before A_2 or vice versa due to causality, **OR**
2. A_1 and A_2 can only occur when a common lock is held, **OR**
3. (A_1, A_2) is a data race.

Revisiting the definition

If we can find, statically or dynamically, a pair of memory access instructions (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:

1. A_1 strictly happens before A_2 or vice versa due to causality, **OR**
2. A_1 and A_2 can only occur when a common lock is held, **OR**
3. (A_1, A_2) is a data race.

Q: Wait... how are locks implemented?

Common synchronization primitives

Common synchronization primitives

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

How are synchronization primitives implemented?

- Hardware support
  - Atomic swap
  - Atomic read-modify-write
    - compare-and-swap
    - test-and-set
    - fetch-and-add
    - ......

How are synchronization primitives implemented?

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

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.
```

Spinlock with atomic swap (`xchg`)

Q: Are there data races or race conditions in spinlock implementation?

Spinlock with atomic swap (`xchg`)

Q: Are there data races or race conditions in spinlock implementation?

A: By looking at the code

- Data race: Yes, but hardware guarantees atomicity
- Race condition: No

Dekker's algorithm

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

```c
// 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

```c
// 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

Dekker's algorithm

Q: Are there data races or race conditions in Dekker's algorithm?

Dekker's algorithm

Q: Are there data races or race conditions in Dekker's algorithm?

A: By looking at the code

- Data race: Yes, but hardware guarantees atomicity
- Race condition: No

Is this a data race or a race condition or neither?

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

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

Thread 1

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

Thread 2

Is this a data race or a race condition or neither?

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

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

Thread 1

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

Thread 2

Outline

Revisit the example

```
global var count = 0
```

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

  t++;

  lock(mutex);
  count = t;
  unlock(mutex);
}
```

Thread 1

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

  t++;

  lock(mutex);
  count = t;
  unlock(mutex);
}
```

Thread 2

Revisit the example

Q: In this modified example, is there a data race?

Revisit the example

Q: In this modified example, is there a data race?

A: No

Revisit the example

Q: In this modified example, is there a data race?

A: No

Q: But the results are the same with all locks removed?

```
global var count = 0
```

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

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

Revisit the example

Q: In this modified example, is there a data race?

A: No

Q: But the results are the same with all locks removed?

```
global var count = 0
```

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

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

A: No, depending on how hardware works (e.g., per-bit conflict)

Extract the commonalities of the two variants

Q: What is common in developers' expectations in the two variants?

Extract the commonalities of the two variants

Q: What is common in developers' expectations in the two variants?

A: States do not change for a critical section during execution.

Extract the commonalities of the two variants

Q: What is common in developers' expectations in the two variants?

A: States do not change for a critical section during execution.

A: **Generalization**: states remain integral for a critical section during execution. No change of states is just one way of remaining integral (assuming state is integral before the critical section).

State integrity example

```
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;
```

```
int add_x(v: int) {
  g.x += v;
  g.y -= v;
}
```

Thread 1

```
int add_y(v: int) {
  g.y += v;
  g.x -= v;
}
```

Thread 2

State integrity example

```
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;
lock mutex = unlocked;
```

```
int add_x(v: int) {
  lock(mutex);
  g.x += v;
  unlock(mutex);
  lock(mutex);
  g.y -= v;
  unlock(mutex);
}
```

Thread 1

```
int add_y(v: int) {
  lock(mutex);
  g.y += v;
  unlock(mutex);
  lock(mutex);
  g.x -= v;
  unlock(mutex);
}
```

Thread 2

Q: Is this the right way of adding locks?

State integrity example

```
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;
lock mutex = unlocked;
```

```
int add_x(v: int) {
  lock(mutex);
  g.x += v;
  unlock(mutex);
  lock(mutex);
  g.y -= v;
  unlock(mutex);
}
```

Thread 1

```
int add_y(v: int) {
  lock(mutex);
  g.y += v;
  unlock(mutex);
  lock(mutex);
  g.x -= v;
  unlock(mutex);
}
```

Thread 2

Q: Is this the right way of adding locks?

A: No, as the invariant is not guaranteed

State integrity example

```
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;
lock mutex = unlocked;
```

```
int add_x(v: int) {
  lock(mutex);
  g.x += v;
  g.y -= v;
  unlock(mutex);
}
```

Thread 1

```
int add_y(v: int) {
  lock(mutex);
  g.y += v;
  g.x -= v;
  unlock(mutex);
}
```

Thread 2

Q: Is this the right way of adding locks?

State integrity example

```
struct R { x: int, y: int } g;
[invariant] g.x + g.y == 100;
lock mutex = unlocked;
```

```
int add_x(v: int) {
  lock(mutex);
  g.x += v;
  g.y -= v;
  unlock(mutex);
}
```

Thread 1

```
int add_y(v: int) {
  lock(mutex);
  g.y += v;
  g.x -= v;
  unlock(mutex);
}
```

Thread 2

Q: Is this the right way of adding locks?

A: Yes, the invariant is guaranteed at each entry and exit of the critical section in both threads

State integrity is hard to capture

However, in practice, the invariant often exists in

- some architectural design documents (which no one reads)
- code comments in a different file (which no one notices)
- forklore knowledge among the dev team
- the mind of the developer who has resigned a few years ago...

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:

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

We have:

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

Practice exercise (vector clock)

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

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

Thread 1

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

Thread 2

Prove: the write of `x` at `x--` in thread 2 can never happen before the read of `x` in `x++` in thread 1.

Practice exercise (vector clock)

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

```
1 v = load(&x);
2 store(&x, v + 1);
3 store(&r, true);
```

Thread 1

```
1 loop:
2   c = load(&r);
3   jump_if_false(c, loop);
4 v = load(&x);
5 store(&x, v - 1);
```

Thread 2

Prove: line 5 at thread 2 can never happen before line 1 at thread 1.

Outline

A more abstract view of race conditions

Q: Why do race conditions happen in the first place?

A more abstract view of race conditions

Q: Why do race conditions happen in the first place?

A: Because two threads in the same process share memory

A more abstract view of race conditions

Q: Why do race conditions 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?

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)` $\rightarrow$ `root(0)`.

Existed in the kernel for nine years before finally patched.

Details on the Website.

⟨ **End** ⟩