Race conditions

When several threads access and manipulate the same data concurrently, the outcome of the execution depends on the particular order in which the access takes place.

This is called a **race condition**.

Bugs in multi-threaded programs caused by race conditions are hard to debug and are called "Heisenbugs."

Examples: race.c and no_race.c on BBLearn

$\boxed{\text{Counter} = 5}$

S_0	S_1
counter++;	counter --;

counter++ : not atomic

(6)
I_1 lw \$r1, <counter>
I_2 addi \$r1, \$r1, 1
I_3 sw \$r1, <counter>

(4)
lw \$r2, <counter> I_4
subi \$r2, \$r2, 1 I_5
sw \$r2, <counter> I_6

Single core machine:

- Time shares between threads. | S_0 | S_1 | S_0 | S_1 | … (context switch; time)

I_1 I_2 I_4 I_5 I_6 I_3 (time)

counter = 4 (after I_6); counter = 6 (after I_3)

S_0 is context switched (after I_2); S_1 is context switched (after I_6)

6

- $I_1 I_2 I_4 I_5 I_6 I_3 \rightarrow 6$ ✗
- $I_4 I_5 I_1 I_2 I_3 I_6 \rightarrow 4$ ✗
- $\underbrace{I_1\ I_2\ I_3}_{S_0}\ \underbrace{I_4\ I_5\ I_6}_{S_1} \rightarrow 5$ ✓
- $\underbrace{I_4\ I_5\ I_6}_{S_1}\ \underbrace{I_1\ I_2\ I_3}_{S_0} \rightarrow 5$ ✓

Race condition:
~~4~~, 5 ✓, ~~6~~

Multi-core machine

	S_0 (core 0)	S_1 (core 1)
time ↓	lw \$r1, <counter>	lw \$r2, <counter>
	addi \$r1, \$r1, 1	subi \$r2, \$r2, 1
	sw \$r1, <counter>	sw \$r2, <counter> *

Whoever writes last wins. (4 or 6)

S_0 S_1
or
S_1 S_0 ✓

Condition

→ Atomicity → counter++ (or) counter--

↓

Indivisible

operations must be atomic

> Most machine instructions are not atomic

The critical section problem

Critical section: segment of code in which threads may be modifying shared variables

Any solution to the critical section problem must satisfy the following three conditions:

Mutual exclusion: if thread T_i is executing in its critical section, then no other threads can be executing in that critical section.

Progress: all threads must be making progress towards their overall objective. This means that the system is free of **deadlocks**.

Bounded waiting: there exists a bound or limit on the number of times other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before the request is granted.

```
while(1) {
    ...
    Critical section (CS)
    ...
}
```

set of instructions that are operating on data shared between threads

S_0

```
while(1){
    ...
CS  counter++
    ...
}
```

S_1

```
while(1){
    ...
CS  counter--
    ...
}
```

Hardware support for synchronization

- Instruction ~~Set~~ architecture (ISA)
  - ↳ exchange (XCHG)
  - ↳ atomic CAS ( )

compare and swap

atomic instructions

implement lock or mutex

Simplest way to achieve mutex:

→ Microcontrollers (MSP432 TI)

↳ single core → time sharing

↳ CPU is time shared between threads

↓ time quanta

↓ timer

↓ interrupt

↓ ISR()

Interrupt service routine → ISR()

```
While(1){
   /* Disable interrupts */
   Critical section
   /* Enable interrupts */
}
```

keep short

Will not work for multi-cores.

↳ Each core has its own interrupt set up.

atomicCAS

Behavior of atomicCAS:

shared memory

mutex

atomic

```
int atomicCAS(int *mutex, int compareVal,
              int newVal)
{
    int oldVal = *mutex;
    if (*mutex == compareVal)
        *mutex = newVal;
    return oldVal;
}
```

Lock implementation:

mutex

Shared variables

int mutex = 0

S_0

```
While(1){
    While(atomicCAS(&mutex,0,1) != 0);
    Critical section
    mutex = 0;
}
```

short

S_1

```
While(1){
    While(atomicCAS(&mutex,0,1) != 0);
    Critical section
    mutex = 0;
}
```

spin lock

Exchange:

atomic

```
void exchange (int *mutex, int registerVal)
{
    int temp;
    temp = *mutex;
    *mutex = registerVal;
    registerVal = temp;
}
```

mutex = 0

S_0

```
int key = 1;
while (key != 0)
    exchange(&mutex, key);
{
    critical section
}
exchange(&mutex, key);
```

S_1

```
int key = 1;
while (key != 0)
    exchange(&mutex,
             key);
critical section
exchange(&mutex,
         key);
```

spin lock

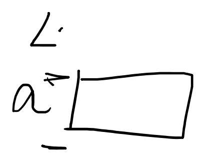
Recap:

- Atomicity → operations in CS should be indivisible
- Isolation → when multiple threads operate concurrently on a data structure, the final state should be the same as though the operations were performed sequentially.

Locks

Spinning versus context switching

"Smart" locks



S_0

```
lock(L);
/* CS */
unlock(L);
```

S_1

```
Lock(L);
/* CS */
unlock(L);
```

Uniprocessor case:

- Block the thread if it cannot acquire lock.

Multiprocessor case:

- Let threads spin on the locks
- Keep sizes of critical sections small

"Smart" locks:

↳ S_0 has lock
S_1 wants lock

If S_0 is currently executing on some core,
let S_1 spin

If S_0 is currently blocked, block S_1

Deadlocks

The use of locks serializes execution through critical sections -> loss of parallelism.

To reduce the impact on performance, the size of critical sections must be kept small (few hundreds of instructions or fewer) -> granularity of locking should be small.

Examples: non-preemptive kernel versus preemptive kernel.

Large applications (such as an OS kernel) may have thousands of locks in them.

Must be careful to avoid deadlocks.

Examples…

How to handle deadlocks? Deadlock avoidance, deadlock prevention, deadlock detection, … or the Ostrich approach

Lock ordering protocol for deadlock avoidance

Nested locks:

Locks

A → $\boxed{CS_1}$

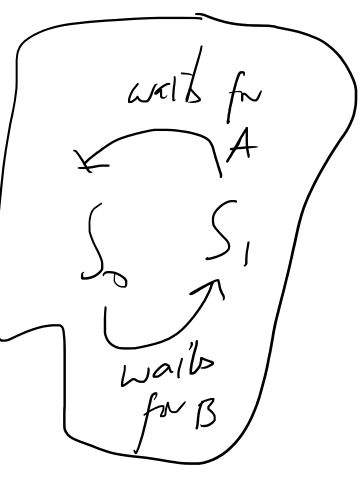
B → $\boxed{CS_2}$

time ↓

S_0	S_1
lock(A);	lock(B);
$\boxed{CS1}$	$\boxed{CS2}$
lock(B);	lock(A);
$\boxed{CS2}$	$\boxed{CS1}$
unlock(B);	unlock(A);
unlock(A);	unlock(B);

Is this code prone to deadlock?

cycle?



Hold + wait condition ⇒ circular wait

S_0 holds A; waits for B

S_1 holds B; waits for A

Deadlock prevention: lock ordering protocol

- Lock ordering
  - Given locks $\{L_1, L_2, \ldots, L_m\}$, assign to each lock a unique integer number $F(L_i)$, which allows us to compare two locks and to determine whether one precedes another in our ordering.
  - Each thread can request locks only in an increasing order of enumeration. That is, a thread can initially request any number of instances of a lock type, say L_i. After that a thread can request instances of lock type L_j if and only if $F(L_j) > F(L_i)$.
  - A thread requesting an instance of resource type L_j must have released any resource L_i such that $F(L_i) > F(L_j)$.
  - Example

Locks
A
B

$F(A) = 1$
$F(B) = 2$

S_0

```
⋮
lock(A);
⋮
lock(B);
⋮
unlock(B);
unlock(A);
```

S_1

```
⋮
lock(B);
⋮
unlock(B);
lock(A);
lock(B);
⋮
unlock(B);
⋮
unlock(A);
```