Synchronization Intro

- This week
  - Background
  - The Critical-Section Problem
  - Peterson’s Solution
  - Hardware Support for Synchronization
- C11 Atomic operations library
  - Atomic operations library
  - memory_order - cppreference.com
  - slides: Memory barriers in C
  - High level software solutions
    - Mutex Locks
    - Semaphores
    - Monitors

Synchronization Outline

- This week
  - Background
  - The Critical-Section Problem
  - Peterson's Solution
  - Hardware Support for Synchronization
- C11 Atomic operations library
  - Atomic operations library
  - memory_order - cppreference.com
  - slides: Memory barriers in C
  - High level software solutions
    - Mutex Locks
    - Semaphores
    - Monitors

Next week

- Implementation of locks
  - kernel space
  - user level implementation
- Cache coherence
- Lock "Free" Multithreading
  - memory barriers
- Lock free data structures
  - RCU
- transactions

Next next week

- Review and summary of synchronization

```c
void *producer(void *data){
   while (1)    {
       /* produce an item in next produced */
       while (count == BUFFER_SIZE)
           ; /* do nothing */

       buffer[in] = produced;
       in = (in + 1) % BUFFER_SIZE;
       count++;
   }
}
```

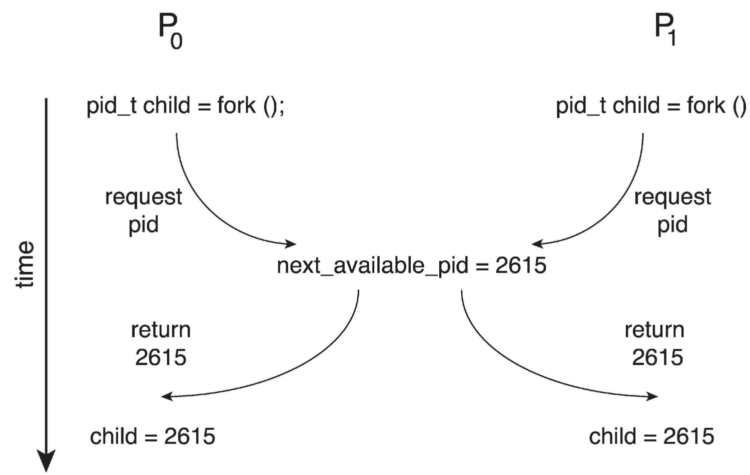
```c
void *consumer(void *data){
   while (1){

       while (count == 0)
           ; /* do nothing */

       consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
       count--;

       /* consume the item in next consumed
*/
   }
}
```

```c
#include <stdio.h> #include <pthread.h>
#define NRUN 100000
int total = 0;
void *transaction(void *data){
    for (int i = 0; i < NRUN; i++){
        total++;
    }
}
int main(int argc, char **argv){
    pthread_t thread_id[2];
    pthread_create(&thread_id[0], NULL, transaction, NULL);
    pthread_create(&thread_id[1], NULL, transaction, NULL);
    pthread_join(thread_id[0], NULL);
    pthread_join(thread_id[1], NULL);
    printf("total- expected:%d, actual:%d\n", 2 * NRUN, total);
    return 0;
}
```

In a time-shared system, the **exact instruction execution order cannot be predicted**!

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Requirements for solution to critical-section problem

1. **Mutual Exclusion**
   - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress**
   - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting**
   - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Assumptions:

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the n processes

Hardware solutions:
Interrupt-based solution

- Entry section: disable interrupts
- Exit section: enable interrupts

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

- Will this solve the problem?
  - What if the critical section-code runs for an hour?
  - Can some processes starve
    - never enter their critical section.
  - What if there are two CPUs?

Software Solutions

Try-1:

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic;
  - that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn;`**
    - indicates whose turn it is to enter the critical section
    - initialized to ***i***

Try-1: (strict alternation)

```
// P0
while (true){

   while (turn != 0)  {
        ;
   } // P1's turn

   // MY TURN
   /* critical section */

   turn = 1;

   /* remainder section */
}
```

```
// P1
while (true){

   while (turn != 1) {
        ;
   } // P0's turn

   // MY TURN
   /* critical section */

   turn = 0;

   /* remainder section */
}
```

https://phoenix.goucher.edu/~kelliher/cs42/sep27.html

Correctness of the Try-1

- Mutual exclusion is preserved
  - P_i enters critical section only if:
    - **`turn = i`**
    - and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
  - does not guarantee progress: enforces **strict alternation of processes** entering CS.
  - e.g.; P0 in remainder section,
    - P1 executes its critical section,
    - it changes the turn variable to 0.
    - P1 finishes its remainder section, now it has to wait P0’s remainder section
- What about the Bounded-waiting requirement?
  - Bounded waiting violated,
    - one process terminates while it is its turn

try-2: Remove strict alternation from try-1

```
/*flag[i] indicates that Pi is in its critical section*/
int flag[2] = {false, false};
```

```
// P0
while (true){

   while (flag[1])   {// P1 in cs
        ;
   }
   // MY TURN
   flag[0] = true;
   /* critical section */
   flag[0] = false;

   /* remainder section */
}
```

```
// P1
while (true)
{

   while (flag[0]){// P0 in cs
        ;
   }

   // MY TURN
   flag[1] = true;
   /* critical section */
   flag[1] = false;

   /* remainder section */
}
```

Correctness of try-2

- Mutual exclusion is violated
  - P0 exits while loop, then context switch.
  - P1 exits while loop,
  - both can enter critical section
- What about the Progress requirement?
  - OK
- What about the Bounded-waiting requirement?
  - OK

try-3: Restore mutual exclusion in try-2

```
/*flag[i] indicates that Pi wants to enter critical section*/
int flag[2] = {false, false};
```

```
// P0
while (true){

   // wants to enter
   flag[0] = true;

   while (flag[1]){// P1 in cs
       ;
   }

   /* critical section */
   flag[0] = false;

   /* remainder section */
}
```

```
// P1
while (true){

   // wants to enter
   flag[1] = true;

   while (flag[0]){// P0 in cs
       ;
   }

   /* critical section */
   flag[1] = false;

   /* remainder section */
}
```

Correctness of try-3

- Mutual exclusion is guaranteed.

- What about the Progress requirement?
  - violated
    - both proces can set flags, then deadlock on the while-loop
- What about the Bounded-waiting requirement?
  - violated, infinite loop.

try-4: attempt to remove deadlock

```
/*flag[i] indicates that Pi wants to enter critical section*/
int flag[2] = {false, false};
```

```
// P0
while (true){
   // wants to enter
   flag[0] = true;

   while (flag[1]) {
       flag[0] = false;
       delay();
       flag[0] = true;
   }

   /* critical section */
   flag[0] = false;

   /* remainder section */
}
```

Progress is still violated!

- both proces can "dance" in the while-loop

Bounded waiting violated

```
// P1
while (true){

   // wants to enter
   flag[1] = true;

   while (flag[0]) {
       flag[1] = false;
       delay();
       flag[1] = true;
   }

   /* critical section */
   flag[1] = false;

   /* remainder section */
}
```

Peterson’s solution

```
int flag[2] = {false, false}; /*flag[i] indicates that Pi wants to enter critical section (it’s ready)*/
int turn = 0; /*indicates which process has the priority (lock) to enter in its CS*/
```

```
// P0
while (true){

    // wants to enter
    flag[0] = true;
    turn = 1;

    while (flag[1] && turn == 1){
        ;
    }
    /* critical section */
    flag[0] = false;

    /* remainder section */
}
```

```
// P1
while (true){

    // wants to enter
    flag[1] = true;
    turn = 0;

    while (flag[0] && turn == 0) {
        ;
    }
    /* critical section */
    flag[1] = false;

    /* remainder section */
}
```

Algorithm for Process P_i

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j)
        ;

        /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

for multiple processes, see Lamport's bakery algorithm - Wikipedia, https://www.javatpoint.com/lamports-bakery-algorithm

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:

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

- Thread 1 performs

```
while (!flag)
  ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

100

Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

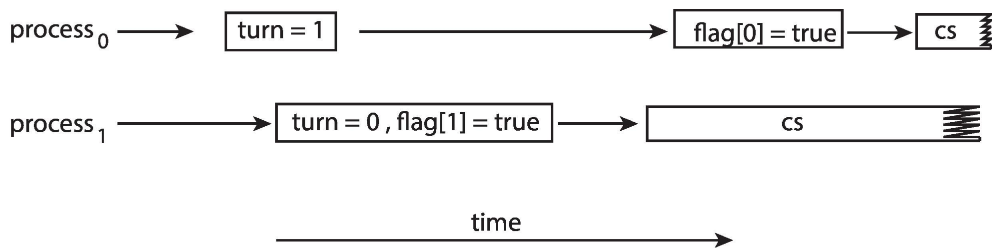
```
flag = true;
x = 100;
```

for Thread 2 may be reordered

- If this occurs, the output may be 0!

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

Hardware Support for Synchronization

Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

see linux memory barriers: Linux kernel documentation on memory barriers
An introduction to lockless algorithms [LWN.net]

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at two forms of hardware support:

1. Hardware instructions
2. Atomic variables

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words **atomically** (uninterruptedly.)
    - **Test-and-Set** instruction
    - **Compare-and-Swap** instruction

The test_and_set Instruction

- Definition

```
boolean test_and_set (boolean *lock) {
        boolean rv = *lock;
        *lock = true;
        return rv:
 }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Mutual Exclusion with test_and_set

```
volatile int lock = 0;

void critical() {
    while (test_and_set(&lock) == 1);/*spinlock*/

    /* critical section */

    lock = 0;  /* release lock when finished CS*
}
```

```
/* Spin lock: loop forever
until we get the lock;
we know the lock was
successfully obtained after
exiting this while loop because
the
test_and_set() function locks
the lock and returns the
previous lock
value.
If the previous lock value was
1 then the lock was **already**
locked by another thread or
process. Once the previous lock
value
was 0, however, then it
indicates the lock was **not**
locked before we
locked it, but now it **is**
locked because we locked it,
indicating
we own the lock.
*/
```

volatile does not guarantee r/w committed to memory(need memory barrier)

[Test-and-set - Wikipedia]

The compare_and_swap Instruction

- Definition

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **value**
  - Set the variable **value** the value of the passed parameter **new_value** but only if ***value == expected (old value)** is true.
    - That is, the swap takes place only under this condition.

may be updated between calls: ABA problem

Solution Using test_and_set()

- Shared boolean variable `lock`, initialized to `0`
- Solution:

```
while (1){
        while (test_and_set(&lock))
              ; /* do nothing */

        /* critical section */

        lock = 0;
        /* remainder section */
 }
```

- Does it solve the critical-section problem?

Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

```
while (true){
        while (compare_and_swap(&lock, 0, 1) != 0)
             ; /* do nothing */

        /* critical section */

        lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

★ This algorithm satisfies the mutual-exclusion requirement,

★ it does not satisfy the bounded-waiting requirement.
  - the same thread may get the lock infinitely

Bounded-waiting with compare-and-swap

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1) { /*enter cs if waiting[i] == false or key == 0.*/
        key = compare_and_swap(&lock, 0, 1);
    }
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j]) /*find the next waiting[j] == true*/
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

```
increment(&sequence);
```

ensures **`sequence`** is incremented without interruption:

Atomic Variables

- The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v){
 int temp;
 do {
     temp = *v;
 }
 while (temp != (compare_and_swap(v,temp,temp+1));
}
```

[6.55 Built-in Functions for Memory Model Aware Atomic Operations](#)
[Atomic operations library - cppreference.com](#)

C atomic library

6.55 Built-in Functions for Memory Model Aware Atomic Operations
Atomic operations library - cppreference.com

Memory reordering-memory barriers

Memory Barriers

Acquire and Release Semantics

The problem

Code	Compiler	CPU

`a= 1;`
`v1= b;`
`c= 2;`
`v2= d;`
	`v2= d;`
	`v1= b;`
	`a= 1;`
	`c= 2;`
		`v2= d;`
		`c= 2;`
		`a= 1;`
		`v1= b;`

https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

[Memory Reordering Caught in the Act]

```
sem_t beginSema1;
sem_t endSema;

int X, Y;
int r1, r2;

void *thread1Func(void *param){
    MersenneTwister random(1);                          // Initialize random number generator
    for (;;)                                            // Loop indefinitely
    {
        sem_wait(&beginSema1);                          // Wait for signal from main thread
        while (random.integer() % 8 != 0) {}            // Add a short, random delay

        // ----- THE TRANSACTION! -----
        X = 1;
        asm volatile("" ::: "memory");                  // Prevent compiler reordering
        r1 = Y;

        sem_post(&endSema);                             // Notify transaction complete
    }
    return NULL;  // Never returns
 };
```

```
$ gcc -O2 -c -S -masm=intel ordering.cpp
$ cat ordering.s
    ...
    mov     DWORD PTR _X, 1
    mov     eax, DWORD PTR _Y
    mov     DWORD PTR _r1, eax
     ...
```

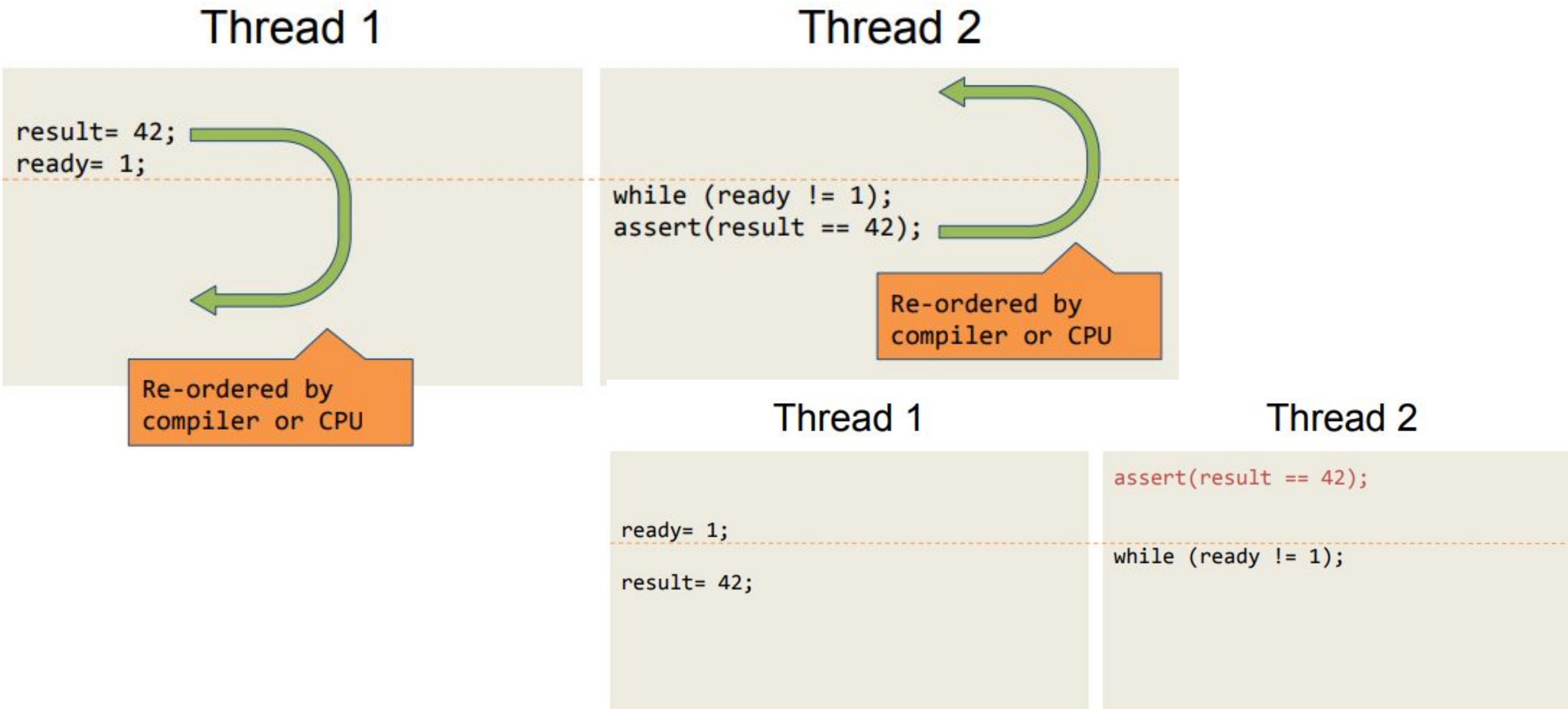
Memory Reordering Caught in the Act

preventing with store/load barrier

```
 for (;;)                                               // Loop indefinitely
  {
     sem_wait(&beginSema1);                              // Wait for signal from main thread
     while (random.integer() % 8 != 0) {}                // Add a short, random delay

     // ----- THE TRANSACTION! -----
     X = 1;
     asm volatile("mfence" ::: "memory");                // Prevent memory reordering
     r1 = Y;

     sem_post(&endSema);                                 // Notify transaction complete
   }
```

```
...
  mov     DWORD PTR _X, 1
  mfence
  mov     eax, DWORD PTR _Y
  mov     DWORD PTR _r1, eax
    ...
```

Memory Reordering Caught in the Act



Memory barriers (jointly with atomic operations) are intended to make data changes visible in concurrent threads.

https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

C API

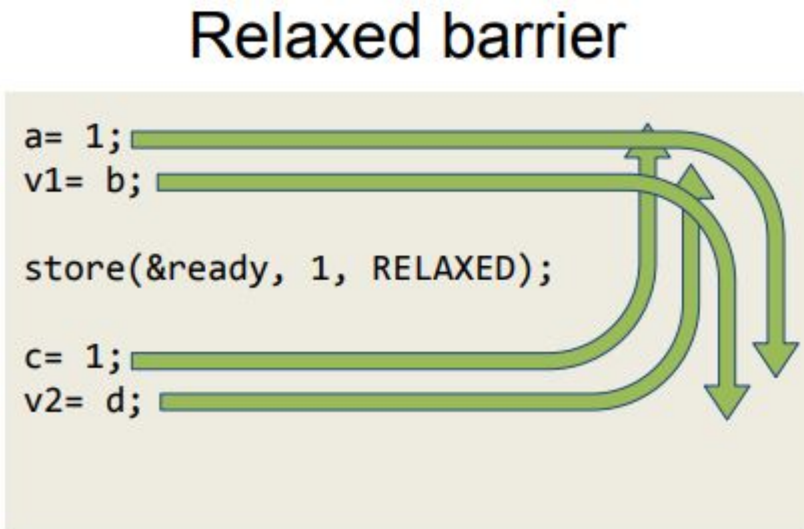
Memory barrier can be issued with atomic operations

[memory_order - cppreference.com]

```
enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

relaxed memory barrier

it guarantees atomicity but does not impose any ordering constraint.




```
// Thread 1:
r1 = atomic_load_explicit(y, memory_order_relaxed); // A
atomic_store_explicit(x, r1, memory_order_relaxed); // B
// Thread 2:
r2 = atomic_load_explicit(x, memory_order_relaxed); // C
atomic_store_explicit(y, 42, memory_order_relaxed); // D

//D can be before A
```

https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

release memory order

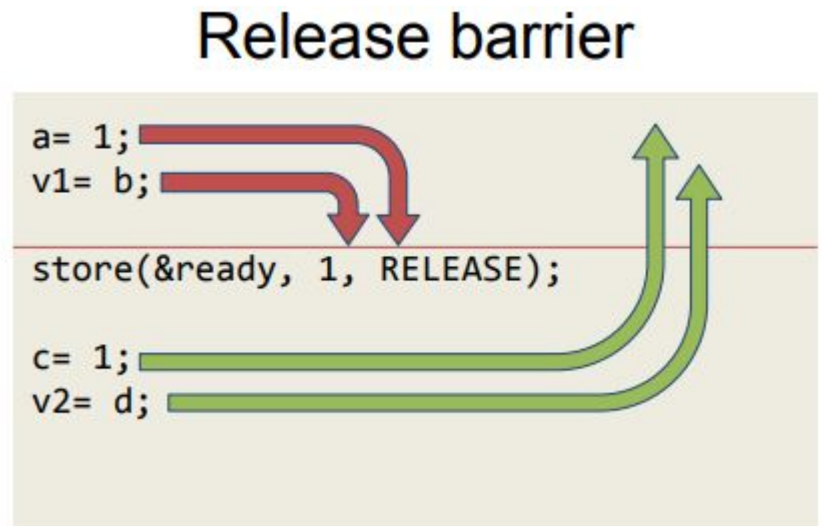
Used with a store operation

- **not valid with load**

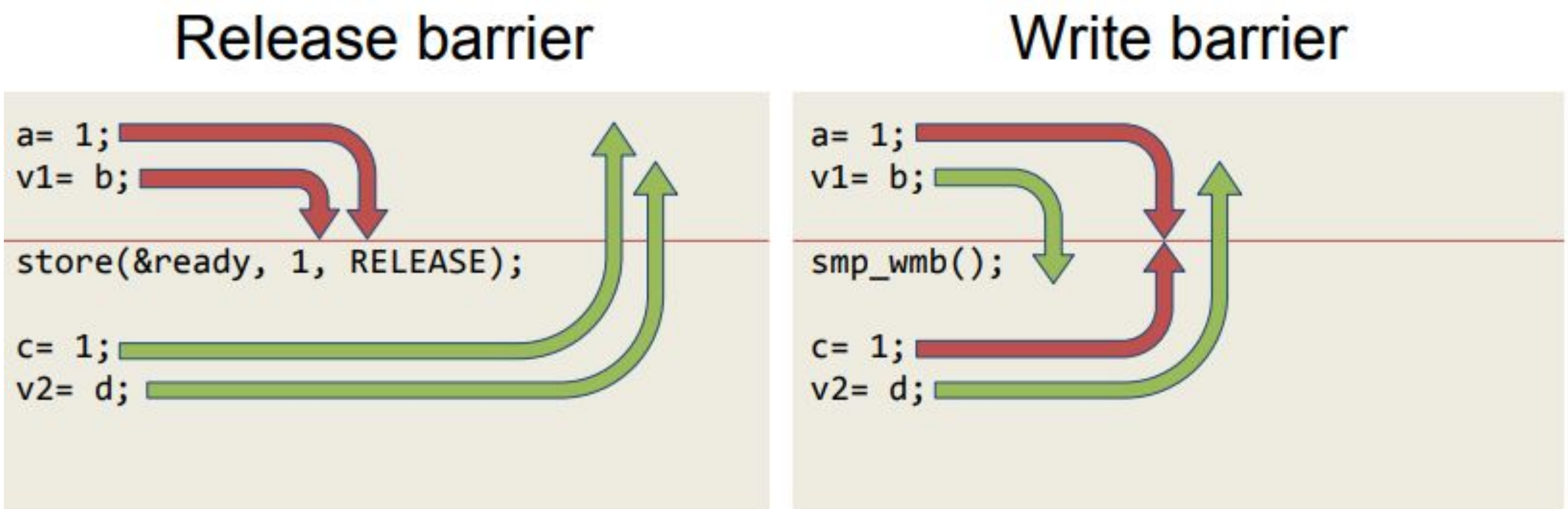
In the same thread:

Loads and stores before Release can not be reordered after **Release**.

Loads and stores after Release can be reordered before Release.



https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf



https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

release is meaningless alone

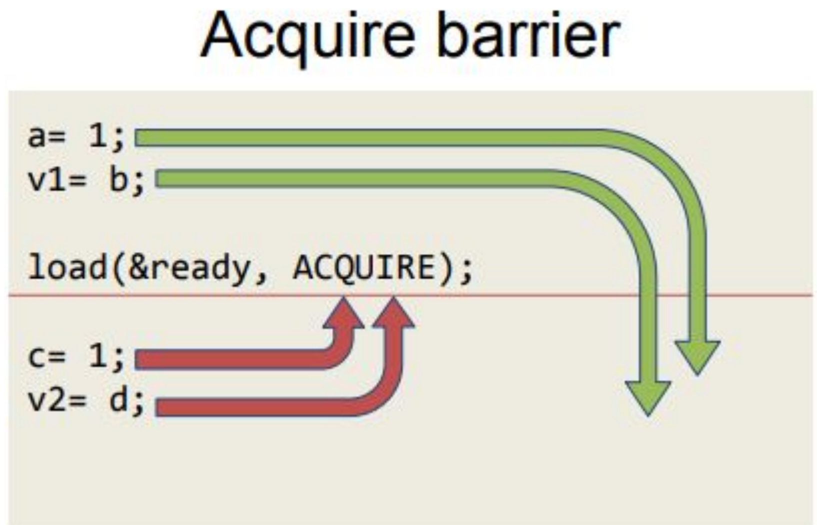
https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Acquire memory order

Used with a load operation

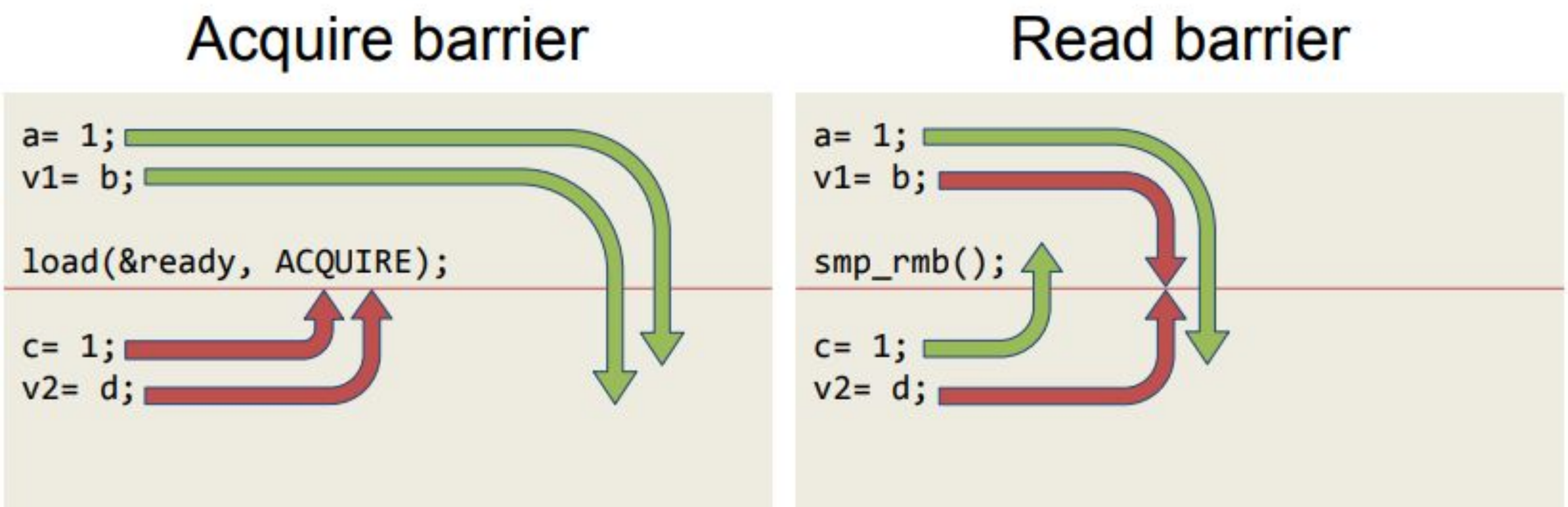
Loads and stores after Acquire can not be reordered **before Acquire.**

Loads and stores before Acquire can be reordered **after Acquire.**



https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Not same as read memory barrier



https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Meaningless alone!

Thread 1

```
result= 42;
ready= 1;
```

Thread 2

```
while (load(&ready, ACQUIRE) != 1);
assert(result == 42);
```

Thread 1

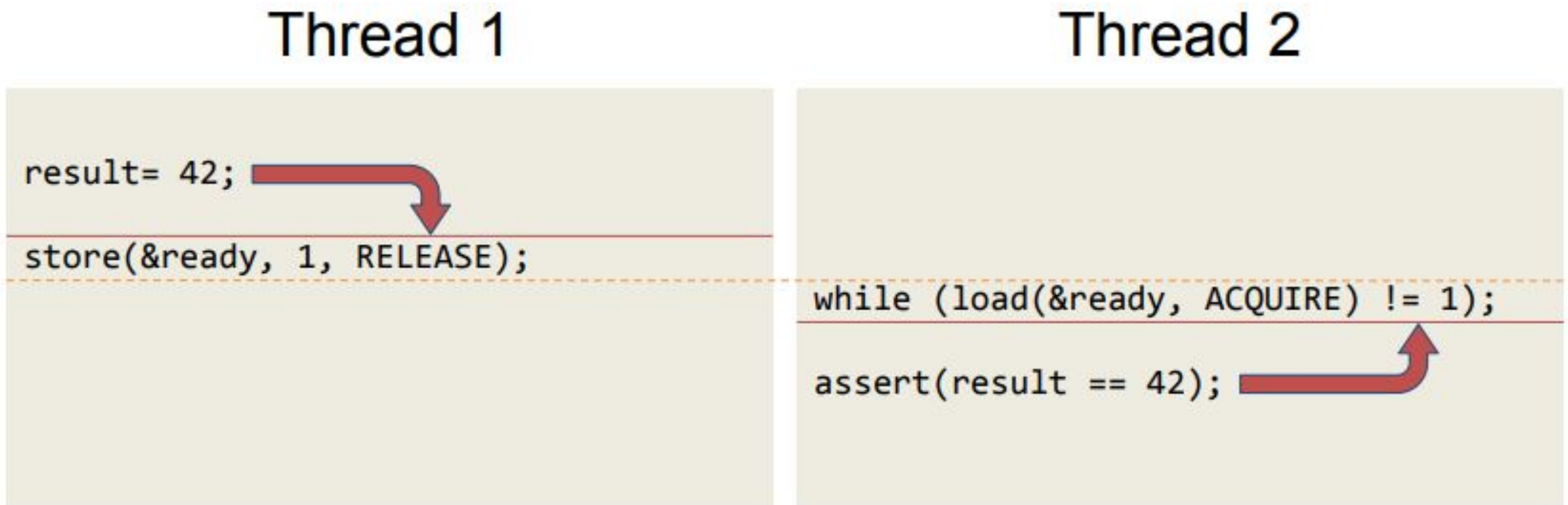
```
ready= 1;

result= 42;
```

Thread 2

```
while (load(&ready, ACQUIRE) != 1);
assert(result == 42);
```

https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Release-acquire model



Acquire must be always paired with Release (or stronger).
Only then all stores before Release in Thread 1 become visible after Acquire in Thread 2.

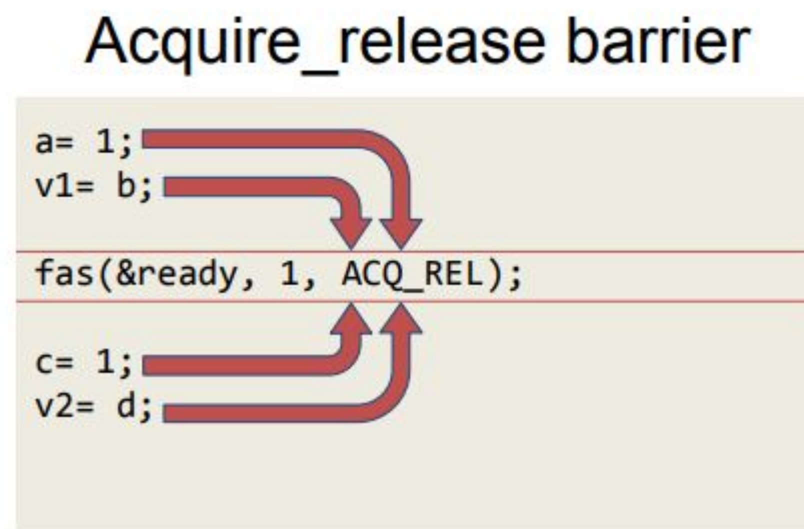
https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Acquire_release memory order

Loads and stores after Acquire_release can not be reordered before Acquire_release.
Loads and stores before Acquire_release can not be reordered after Acquire_release.

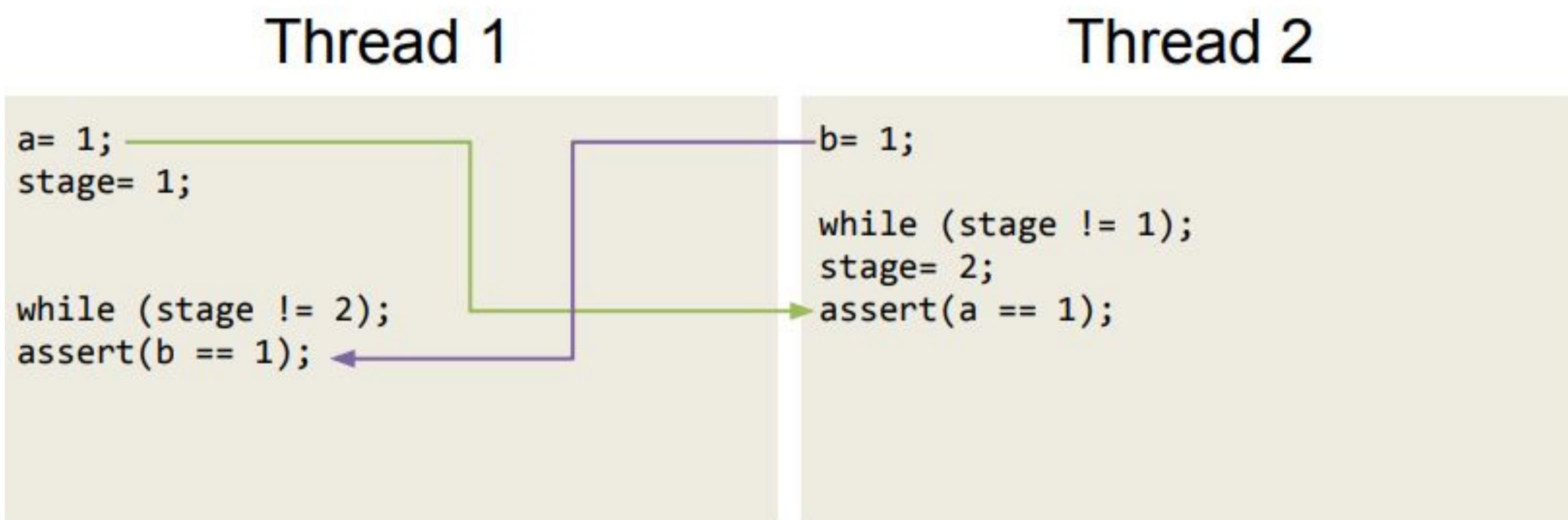
b= fas(&a, 1, ACQ_REL);
b= add(&a, 1, ACQ_REL);
b= cas(&a, &o, 1, ACQ_REL, ACQ_REL);

Not valid with atomic load and store
b= load(&a, ACQ_REL); // undefined, may become ACQUIRE
store(&a, 1, ACQ_REL); // undefined, may become RELEASE

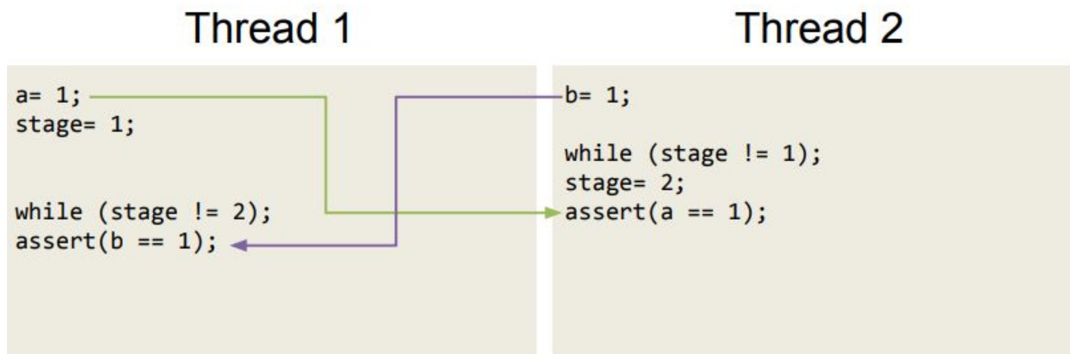


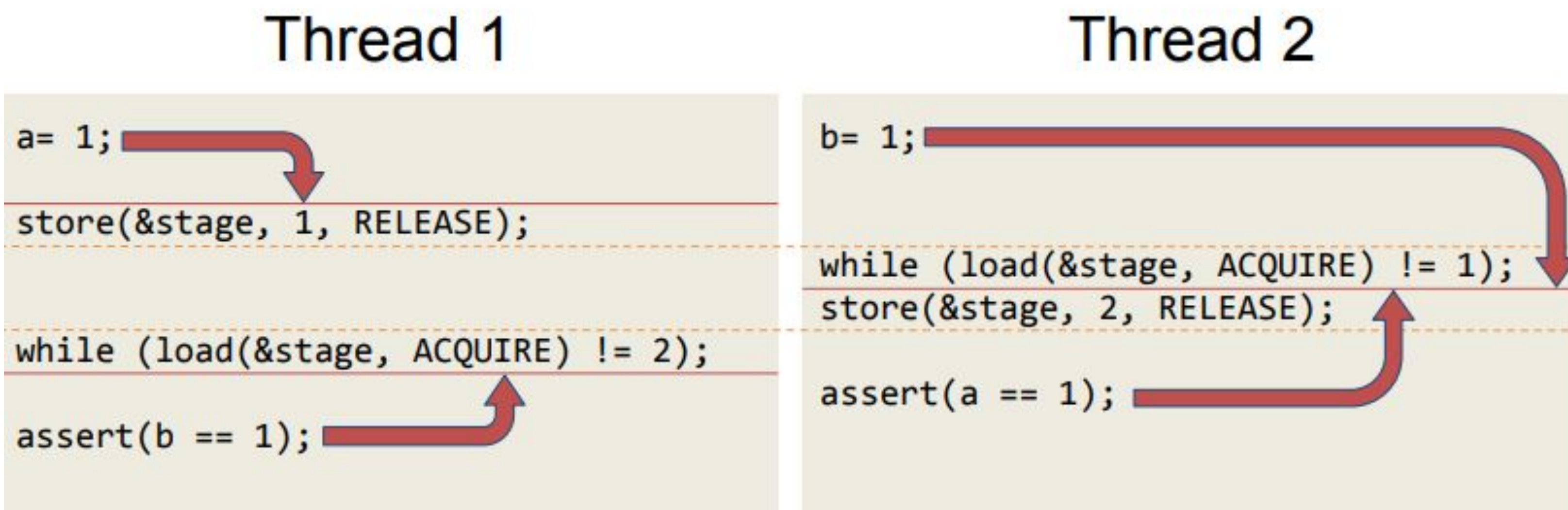
https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

acquire_release memory order-example

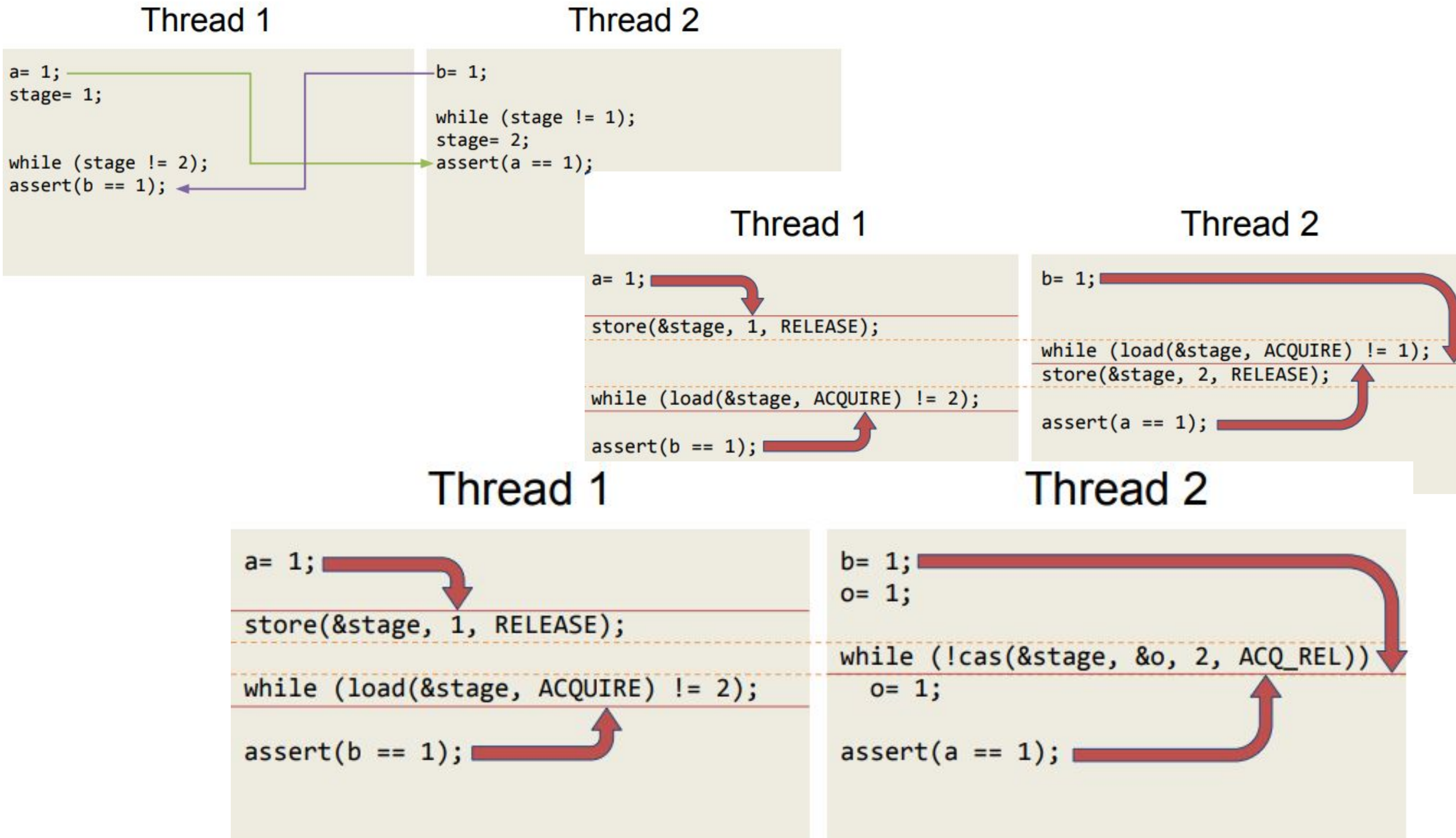


https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf





https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf



https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Consume memory order

Consume is a weaker form of Acquire:

loads and stores, **dependent on the value currently loaded**, that happen after Consume can not be reordered before Consume.

b= load(&a, CONSUME);

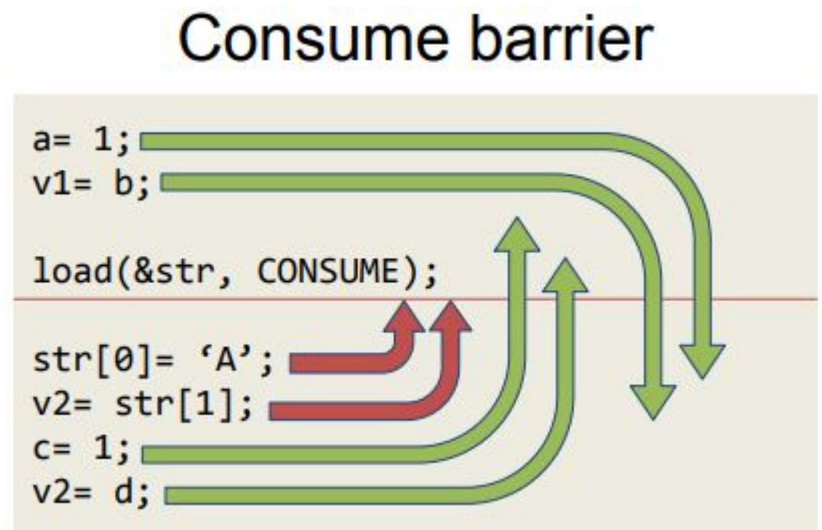
b= fas(&a, 1, CONSUME);

b= add(&a, 1, CONSUME);

b= cas(&a, &o, 1, CONSUME, CONSUME);
fence(CONSUME); // must be preceded by RELAXED atomic load or RMW

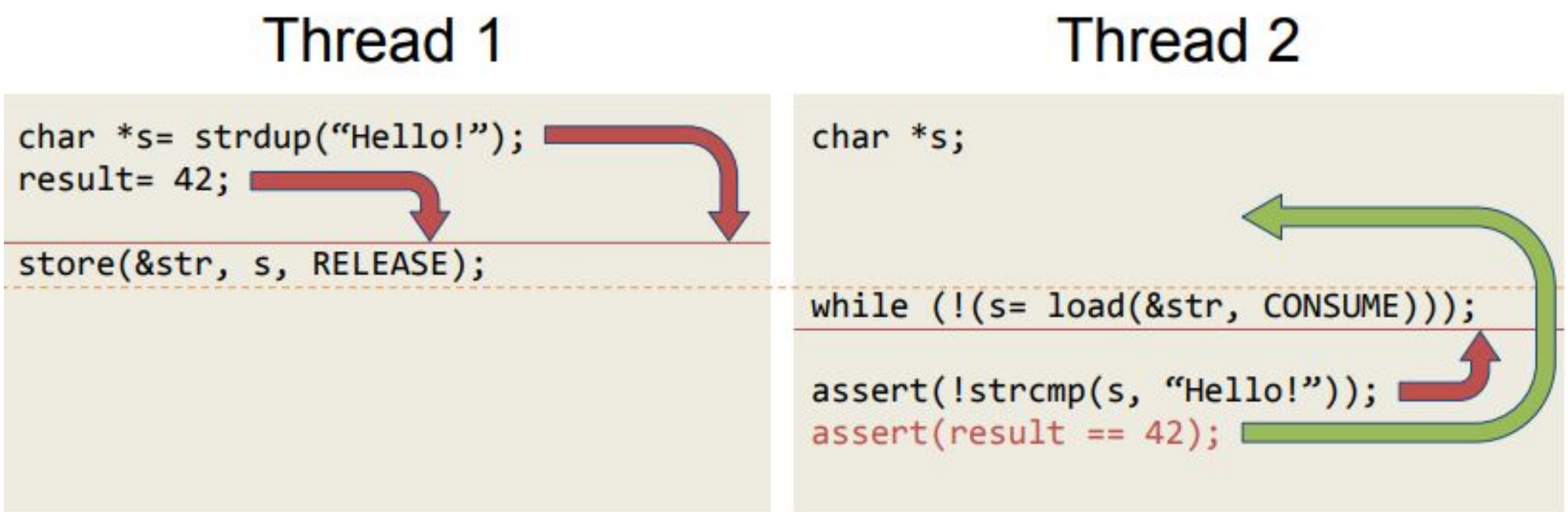
not valid with store

store(&a, 1, CONSUME); // undefined, may become RELAXED



https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Release consume model



Consume must be always paired with Release (or stronger).
Only then all dependent stores before Release in Thread 1 become visible after Consume in Thread 2.
Note that currently no known production compilers track dependency chains: consume operations are lifted to acquire operations.
`__ATOMIC_CONSUME` ([6.59 Built-in Functions for Memory Model Aware Atomic Operations](#))

This is currently implemented using the stronger `__ATOMIC_ACQUIRE` memory order because of a deficiency in C++11’s semantics for `memory_order_consume`

https://mariadb.org/wp-content/uploads/2017/11/2017-11-Memory-barriers.pdf

Sequentially consistent memory order

Loads and stores after Sequentially_consistent can not be reordered before Sequentially_consistent.

Loads and stores before Sequentially_consistent can not be reordered after Sequentially_consistent.

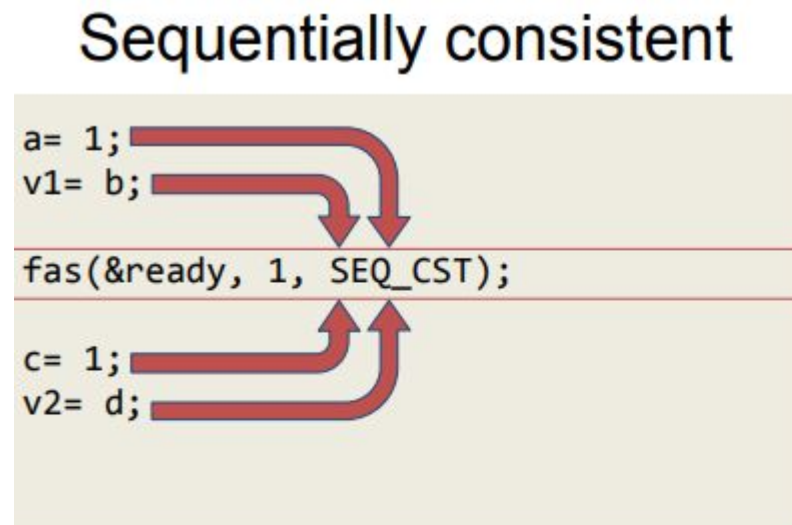
b= fas(&a, 1, SEQ_CST);

b= add(&a, 1, SEQ_CST);

b= cas(&a, &o, 1, SEQ_CST, SEQ_CST);
fence(SEQ_CST);

b= load(&a, SEQ_CST); // may become ACQUIRE + sync

store(&a, 1, SEQ_CST); // may become RELEASE + sync



High Level Software Tools and their implementations

mutex and condition variables from system programming course

the remaining part is skipped in the lecture!

Mutex Locks

- Previous solutions are complicated
  - and generally inaccessible (hardware instructions) to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {
    acquire lock

        critical section

    release lock

remainder section
}
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

- Waiting queue

```
 typedef struct {
int value;
struct process *list;
 } semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Problems with Semaphores

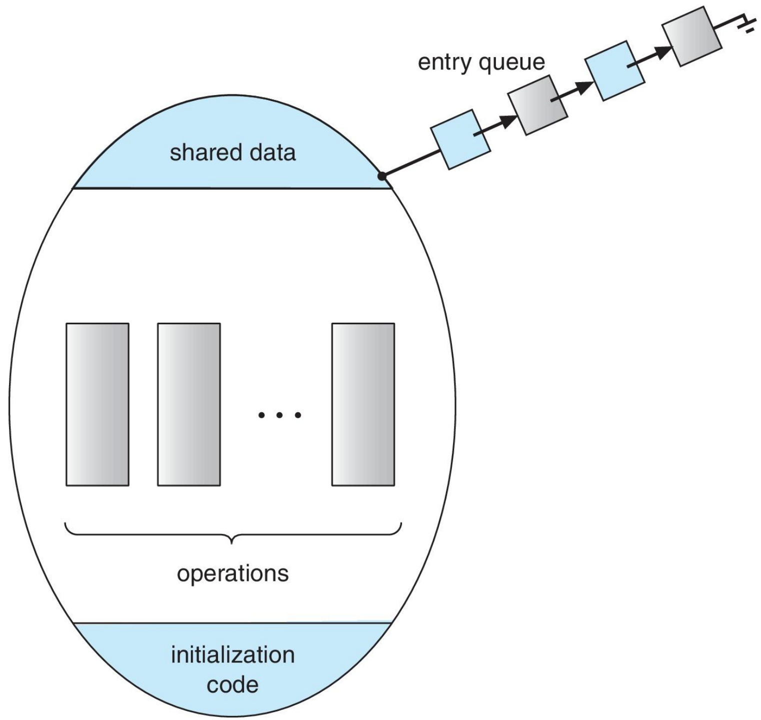
- Incorrect use of semaphore operations:
  - `signal(mutex)` …. `wait(mutex)`
  - `wait(mutex)` … `wait(mutex)`
  - Omitting of `wait (mutex)` and/or `signal (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- ***Abstract data type,*** internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

  initialization code (…) { … }
}
```

Schematic view of a Monitor



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

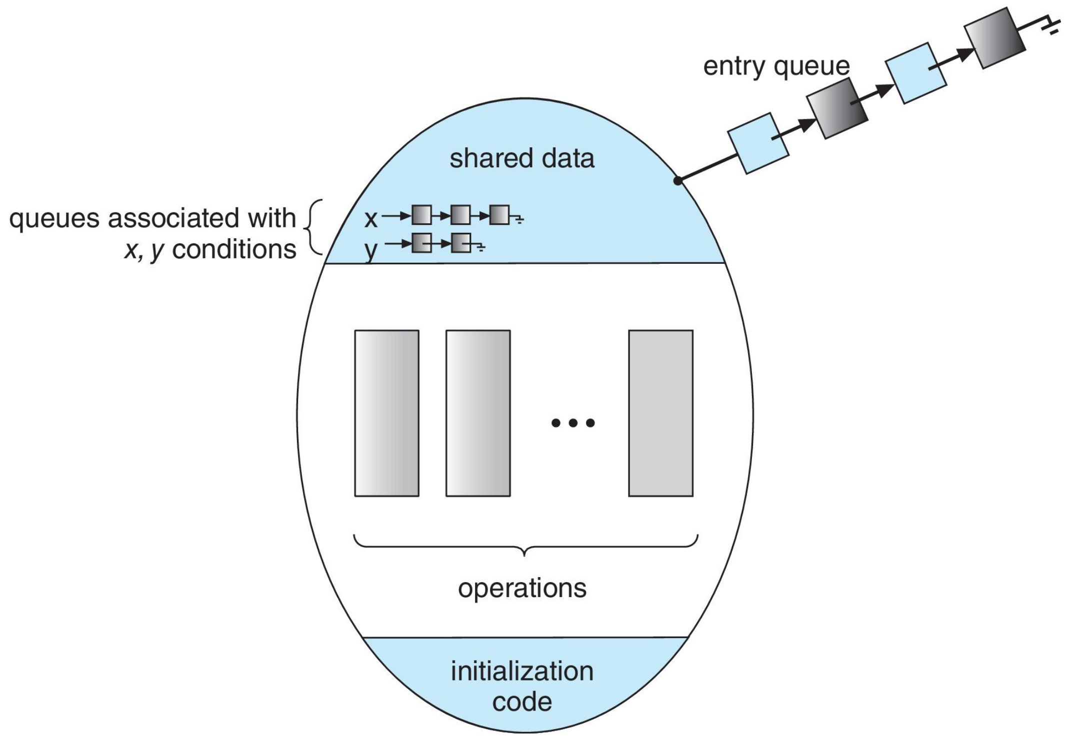
```
wait(mutex);
        …
                body of P;
        …
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
    - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
    - One condition variable “x” initialized to 0
    - One Boolean variable “done”
    - **F1:**
      ```
      S1;
      done = true;
      x.signal();
      ```
    - **F2:**
      ```
      if done = false
         x.wait()
      S2;
      ```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting
                      inside the monitor
```

- Each function ***P*** will be replaced by

```
wait(mutex);
    …
              body of P;
    …
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
     signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition variable **`x`**, and **`x.signal()`** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

  `x.wait(c)`

  where:

  - **`c`** is an integer (called the priority number)
  - The process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource

```
R.acquire(t);
      ...
  access the resurce;
      ...

R.release;
```

- Where R is an instance of type **`ResourceAllocator`**

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the shortest time is allocated the resource first
- Let R is an instance of type **`ResourceAllocator`** (next slide)
- Access to **`ResourceAllocator`** is done via:

```
R.acquire(t);
    ...
  access the resurce;
    ...
R.release;
```

- Where **`t`** is the maximum time a process plans to use the resource

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
      if (busy)
            x.wait(time);
      busy = true;
    }
    void release() {
      busy = false;
      x.signal();
    }
   initialization code() {
     busy = false;
    }
}
```

Single Resource Monitor (Cont.)

- Usage:

```
acquire
 ...
release
```

- Incorrect use of monitor operations
  - **`release()`** … **`acquire()`**
  - **`acquire()`** … **`acquire())`**
  - Omitting of **`acquire()`** and/or **`release()`**

End of Chapter 6