



Chapter 7

Synchronization Examples

Da-Wei Chang

CSIE.NCKU

*Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne,
"Operating System Concepts", 10th Edition, Wiley.*

Outline

- ✓ • Classic Problems of Synchronization
 - Synchronization within the Kernel
 - POSIX Synchronization
 - Synchronization in Java
 - Alternative Approaches
 - Transactional Memory
 - OpenMP
 - Functional Programming Languages

Classical Problems of Synchronization



- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- N buffers, each one can hold an item → 可放 N 筆 data
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0 full buffer 數
- Semaphore **empty** initialized to the value N empty buffer 數
- Players
 - Producers
 - produce full buffers, wait for **empty** buffers 找空 buffer 塞 data
 - Consumers
 - produce empty buffers, wait for **full** buffers 找 full buffer 拿 data item

Bounded Buffer Problem (Cont.)

- The structure of the **producer process**

```
do {  
    // produce an item  
    wait (empty);  
    wait (mutex);  
    // add the item to the buffer 塞 data item  
    signal (mutex);  
    signal (full); 建立一個 full buffer  
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the **consumer process**

```
do {  
    wait (full);  
    wait (mutex);  
    // remove an item from buffer 拿出 data item  
    signal (mutex);  
    signal (empty); 建立一個 empty buffer  
    // consume the removed item  
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do not perform any updates ⇒ \nearrow read \nwarrow write
 - Writers – can both read and write. ⇒ ∇ read ∇ write
- Problem – How to allow multiple readers to read at the same time? Of course, only one single writer can access the shared data at the same time.
- Shared Data
 - Data set
 - Integer **readcount** initialized to 0. // number of readers
 - Semaphore **mutex** initialized to 1. // mutex on readcount
 - Semaphore **wrt** initialized to 1. // contention with a writer

Readers-Writers Problem (Cont.)

- The structure of a **writer process**

```
do {  
    wait (wrt) ;  
    // writing is performed  
    signal (wrt) ;  
} while (true);
```


Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readercount == 1) wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount -- ;  
    if (redacount == 0) signal (wrt) ;  
    signal (mutex) ;  
} while (true)
```

⇒ mutex 用來保護 readcount

// mutex: for updating and maintaining readcount

if (readercount == 1) wait (wrt) ; // the following readers do not wait(wrt)

↳ 1st reader 和 writer 搶 wrt (其他 reader 不用, 直接進來)

critical section ⇒ 開始 read data

if (redacount == 0) signal (wrt) ; ↳ last reader 要釋放 wrt

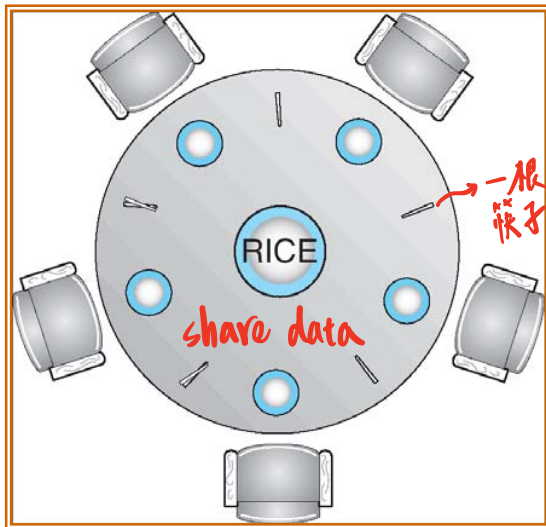
- If a writer is in the critical section and n readers are waiting, then **one reader is blocked on wrt** and **the other $n-1$ readers are blocked on mutex**
- On signal (wrt), we can wakeup a set of readers or a single writer

Reader-Writer Locks

- Some systems support reader-writer locks
- Using the reader-writer locks
 - Specify the lock mode: read or write
- Useful in the following situations 適用 reader-writer lock
 - When it is easy to identify r/w access mode
 - **More readers than writers**

Dining-Philosophers Problem

- 5 Philosophers, spending their lives **thinking** and **eating**
- Get hungry → try to get the chopsticks next to him
 - Pick up only one chopstick at a time
- Eat when he gets both
- Putdown both chopsticks when he is finished eating



Shared data

Bowl of rice (data set)

Semaphores **chopstick** [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher i :

```
do {  
    wait ( chopstick[i] );  
    wait ( chopstick[ (i + 1) % 5] ); } wait 左筷 and 右筷  
    // eat  
    signal ( chopstick[i] );  
    signal ( chopstick[ (i + 1) % 5] ); } 放下左筷 and 右筷  
    // think  
} while (true) ;
```

Dining-Philosophers Problem

- The solution above may lead to deadlock

- e.g., all philosophers pick up their left chopsticks

→ 全拿左筷

- Remedies *solution*

- (I) – At most 4 philosophers

- ✓ (II) – Pick up chopsticks when both are available 一次拿2根

- (III) – Odd philosophers: left, right; even philosophers: right, left.

→ 哲學家作編號

- We will present a solution later

Monitor Solution to Dining Philosophers

Code for each philosopher:

DP.pickup(i); // DP is the monitor

...

eat...

...

DP.putdown(i);

Solution to Dining Philosophers (cont.)

A philosopher picks up chopsticks only when both of them are available

monitor DP

```
{
    enum { THINKING; HUNGRY, EATING) state [5]; 哲學家的狀態
    condition self [5]; //delay herself when she is hungry but unable to obtain the
    chopsticks ↳ 哲學家拿不到筷子 → wait

    void pickup (int i) { 拿筷子
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait; ↳ 沒拿到一雙筷子
    }

    void putdown (int i) { 放下筷子
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5); // allow the philosopher on the left to finish her pickup
        test((i + 1) % 5); // allow the philosopher on the right to finish her pickup
    }
}
```

↳ 看左右 ⇒ 換人吃

Solution to Dining Philosophers (cont.)

```
void test (int i) {  
    if ( (state[(i + 4) % 5] != EATING) &&  
        (state[i] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING) ) {  
        state[i] = EATING ;  
        self[i].signal () ;  
    }  
}
```

} 我 hungry 且 左右都沒在吃

```
    initialization_code() {  
        for (int i = 0; i < 5; i++)  
            state[i] = THINKING;  
    }  
} // end of monitor
```

} 初始化

- **No deadlock, but starvation is possible**

↳ 可能餓死

Synchronization Examples



- Kernel level
 - Solaris
 - Windows
 - Linux
- User level
 - POSIX
 - Java

Solaris Synchronization



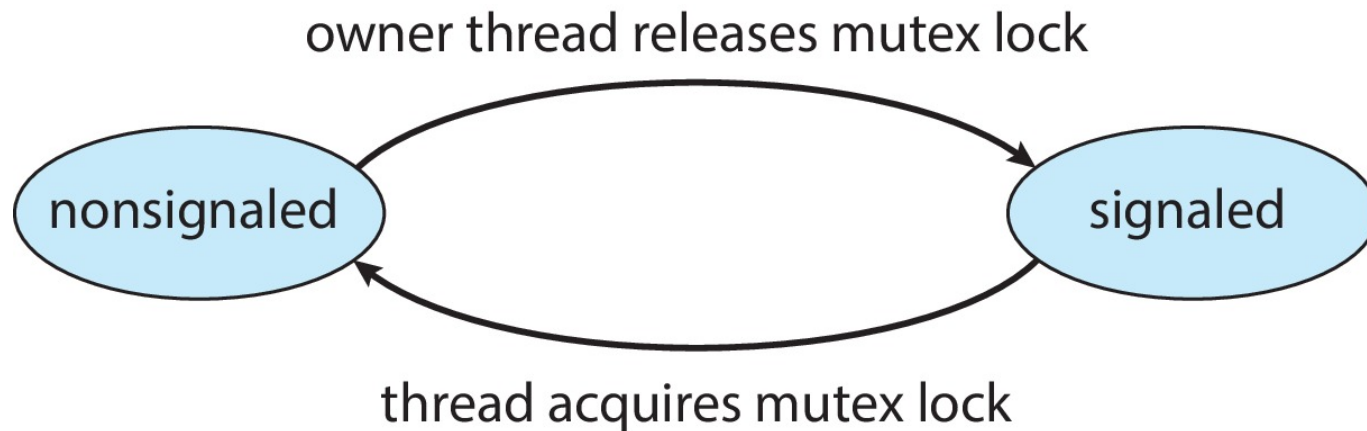
- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
 - If the lock holder is running → busy waiting
 - Otherwise → blocking
- Uses **condition variables** and **semaphores** for longer sections of code that require shared-data access
 - **readers-writers locks** are also provided

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
- Provides **dispatcher objects** which may act as either mutexes and semaphores
 - Object states: *signaled*, *non-signaled*
 - You can call the *KeWaitForSingleObject()* to wait for a dispatcher object
 - wait until another thread sets the object to the *signaled* state

Windows Synchronization

- Mutex dispatcher object



Linux Synchronization



- Linux
 - disables interrupts or uses spinlock for short critical sections
- Linux provides
 - semaphores
 - spinlocks
 - reader-writer locks/semaphores
 - atomic variables (atomic integers)

Linux Synchronization

- Atomic variables
atomic_t is the type for atomic integer
- Consider the variables
atomic_t counter;
int value;

<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&counter,5);</code>	<code>counter = 5</code>
<code>atomic_add(10,&counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4,&counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&counter);</code>	<code>value = 12</code>

POSIX Synchronization



- POSIX API is OS-independent
- It provides
 - mutex locks
 - semaphores
 - condition variables
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutexlock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutexunlock(&mutex);
```


POSIX Mutex Locks API

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

POSIX Semaphores



- POSIX provides two versions – **named** and **unnamed**
- **Named semaphores** can be used by **unrelated** processes by simply referring to the **semaphore's name**
- But, **unnamed semaphores** cannot

POSIX Unnamed Semaphores

- Creating and initializing the semaphore

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

flag Initial value

- Acquiring and releasing the semaphore

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Named Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;
```

```
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

name

flag

mode

Initial value

- Other processes can access the semaphore by referring to the name

SEM.

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);
```

```
/* critical section */
```

```
/* release the semaphore */
sem_post(sem);
```

POSIX Condition Variables

- Previously, **condition variables** are used within the context of a **monitor**
 - The monitor provides a **locking** mechanism to ensure **data integrity**
- POSIX is typically used in C/C++
 - These languages ***do not provide a monitor***
- Thus, POSIX condition variables are associated with a POSIX **mutex lock**
 - The mutex lock is used to provide mutual exclusion

POSIX Condition Variables (Cont.)

- Creating and initializing the condition variable

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

- Thread waiting for the condition **a == b** to become true:
 - The **mutex lock** is used to protect the data in the conditional clause (i.e., a and b) from a possible race condition
 - `pthread_condition_wait()` would *release the mutex lock when waiting*

```
pthread_mutex_lock(&mutex);  
while (a != b)  
    pthread_cond_wait(&cond_var, &mutex);  
  
pthread_mutex_unlock(&mutex);
```

POSIX Condition Variables (Cont.)

- Use `pthread_cond_signal()` to signal a thread waiting on the condition variable

```
pthread_mutex_lock(&mutex);  
a = b;  
pthread_cond_signal(&cond_var);  
pthread_mutex_unlock(&mutex);
```

- Once the **mutex** is released, the signaled thread can re-own the mutex and returns from **`pthread_cond_wait()`**
- The signaled thread must put the conditional clause within a **loop** to **re-check the condition**, i.e., while (a!=b), after being signaled (see the previous slide)

Java Synchronization



- Java provides rich set of synchronization features
 - Java monitors
 - Reentrant locks
 - Semaphores
 - Condition variables

Java Monitors



- Each Java object has an associated lock
- If a method is declared as **synchronized**, a calling thread must own the lock for the object
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released
- Locks are released when the owning thread exits the **synchronized** method

Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

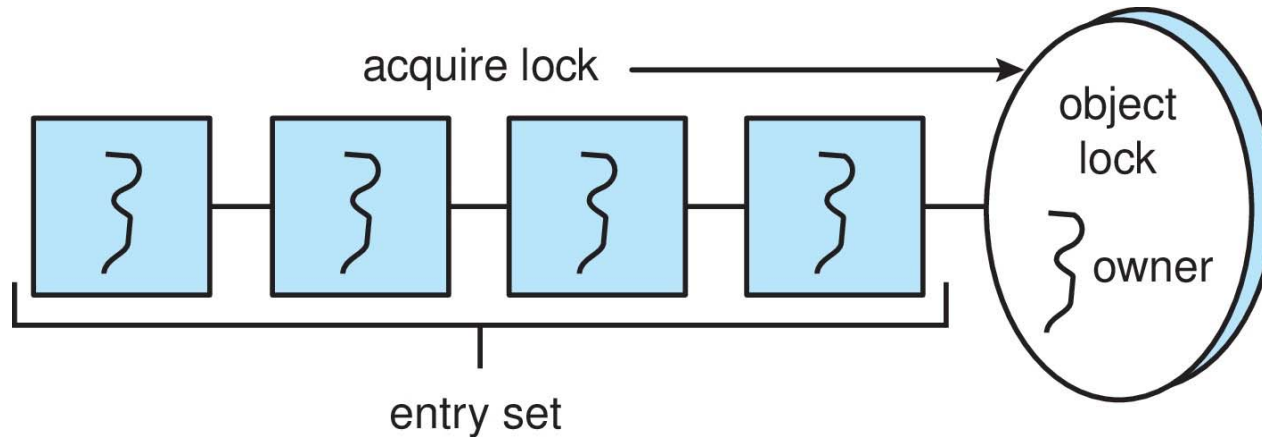
    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```

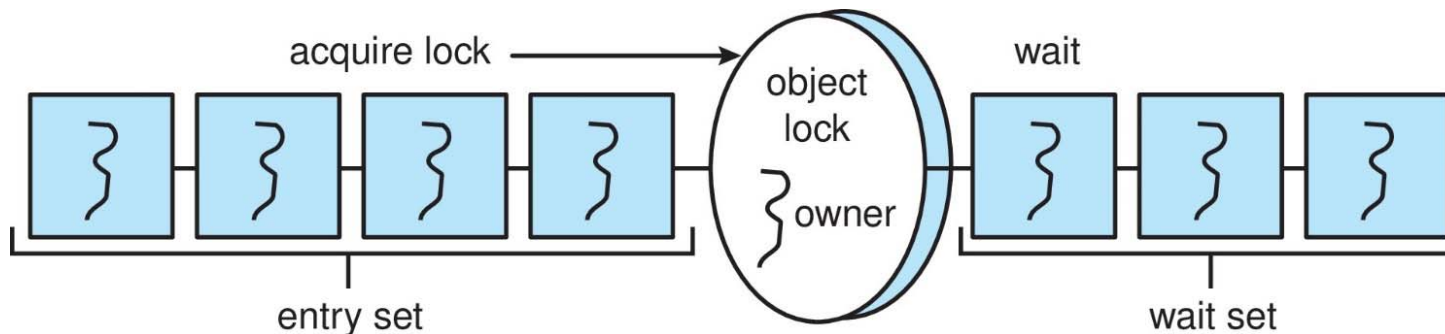
Java Synchronization

- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**



Java Synchronization

- Similarly, each object also has a **wait set**.
- When a thread calls **wait()**
 - It **releases** the lock for the object
 - The state of the thread is set to **blocked**
 - The thread is placed in the **wait set** for the object



Java Synchronization

- A thread T typically calls `wait()` when it is waiting for a condition to become true.
- How does T get notified?
 - When another thread S calls **`notify()`**
 - Thread T may be selected from the wait set, and
 - T is moved from the wait set to the **entry set**
 - The state of T is changed from **blocked** to **runnable**
 - T can compete for the object lock **after S releases the object lock**
 - Once T gets the object lock again, **the `wait()` returns** and T can check whether or not the condition it was waiting for is now true

Bounded Buffer – Java Synchronization

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    notify();
}
```

Bounded Buffer – Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

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

    notify();

    return item;
}
```

Java Reentrant Locks

- Similar to **mutex locks**
 - *Reentrant*: the lock holder can acquire the lock again

```
Lock key = new ReentrantLock();  
  
key.lock();  
try {  
    /* critical section */  
}  
finally {  
    key.unlock();  
}
```

The **finally** clause ensures the lock will be released in case an exception occurs in the **try** block.

Java Semaphores

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```

Java Condition Variables

- Condition variables are associated with an `ReentrantLock`
- Creating a condition variable using **`newCondition()`** method of `ReentrantLock`

```
Lock key = new ReentrantLock();  
Condition condVar = key.newCondition();
```

- A thread waits by calling the **`await()`** method, and signals by calling the **`signal()`** method
 - `await()` will **unlock** (the `ReentrantLock`) **and wait atomically**

Java Condition Variables

- Example
 - Five threads numbered 0 .. 4
 - Shared variable **turn** indicating which thread's turn it is.
 - A thread calls **doWork()** when it wishes to do some work.
 - But it **may only do work if it is its turn**
 - If not its turn, wait
 - If its turn, do some work for awhile
 - When completed, notify the thread whose turn is next.
 - Necessary data structures

```
Lock lock = new ReentrantLock();  
Condition[] condVars = new Condition[5];  
  
for (int i = 0; i < 5; i++)  
    condVars[i] = lock.newCondition();
```

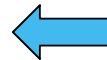
Java Condition Variables

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```



signal the next thread only

Alternative Approaches



- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- **Transaction**

- A collection of instructions that performs single logical function
- A transaction is performed either **in its entirety, or not at all**
 - **Atomic, all-or-none**
 - **Even in the presence of system failures!**
- A transaction is terminated by
 - **Commit** : transaction successful (**in its entirety**), or
 - **Abort**: transaction failed (**not at all**)
- Aborted transaction must be **rolled back** to **undo** any changes it performed
- Example: funds transfer

- **Memory Transaction**

- A sequence of memory read-write operations that are performed either **in its entirety, or not at all**

How to Rollback?

- Record **all modifications** made by a transaction
- Solution: **Write-Ahead Logging (WAL)**
 - **Log before write**
 - The **log** is maintained on **stable storage** (next slide)
 - Log must be completed before data updates
 - When a system failure occurs
 - Consult the log
 - If log contains <Ti starts> without <Ti commits>, **undo(Ti)**
 - If log contains <Ti starts> and <Ti commits>, **redo(Ti)**

Transactions

- Various types of storage
 - Volatile storage
 - Not survive system crashes
 - E.g., memory, cache
 - Non-volatile storage
 - Usually survive system crashes
 - E.g., disk, tape
 - Subject to failure
 - Stable storage
 - Never lose its data

Example

Transaction

init: a=0, b=1;

<T₁ start>

a = 10;

printf(“%d\n”,a);

if (a == 10)

b = 20;

<T₁ commit>

.....

Log

<T₁ start>

T₁, a, 0, 10

T₁, b, 1, 20

Using Transactional Memory for Race Condition Problem

- Example: a function *update()* modifies shared data that uses **mutex** locks.

```
void update() {  
    acquire();  
    /* modify shared data */  
    release();  
}
```

- Alternative solutions: uses **transactional memory**
 - **atomic**{*S*}: the operations in *S* execute as a transaction

```
void update()  
{  
    atomic {  
        /* modify shared data */  
    }  
}
```

Using Transactional Memory for Race Condition Problem

- If a **conflict** is not present // no race condition
 - commit changes
- When a **conflict** is detected // race condition
 - A transaction will roll-back to its initial state
 - And will re-run until no conflict
- Advantages
 - More easier for developer
 - No **locks** are involved, deadlock is not possible

Transactional Memory



- Implementation of transactional memory
 - **Software transactional memory:** software scheme
 - **Compiler** inserts instrumentation code inside transaction blocks
 - **Hardware transactional memory:** hardware scheme

OpenMP

- OpenMP: a set of compiler directives and an API that support parallel programming
- **#pragma omp critical**
 - Compiler directive provided by OpenMP
 - Specify the code region as a critical section

```
void update(int value)
{
    counter+=value;
}
```



```
void update(int value)
{
    #pragma omp critical
    {
        counter+=value;
    }
}
```

OpenMP (Cont.)



- Advantage of using the critical-section compiler directive in OpenMP
 - Easier to use than standard mutex locks
- Limitation
 - Deadlocks is still possible since the directive behaves like a mutex lock

Functional Programming Languages

- Most well-known languages are **imperative** (**procedural**) languages
 - **State-based**
 - Each state is represented by **variables** and **data structures**
- Instead, functional programming language **do not maintain mutable state**
 - Value of variable **cannot be changed**
 - so there are no state-change issues...
 - Thus, no race condition and deadlocks