Chapter 06 Synchronization Tools

Concepts and solutions to synchronization problems

Outline

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Objectives

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Background

- Producer

```
while (true) {
   /* produce an item in next produced */
   while (counter == BUFFER_SIZE) ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

- Consumer

```
while (true) {
   while (counter == 0) ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
   /* consume the item in next consumed */
}
```

Race Condition

- counter++ could be implemented as

  $register_1$ = counter
  $register_1$ = $register_1$ + 1
  counter = $register_1$

- counter-- could be implemented as

  $register_2$ = counter
  $register_2$ = $register_2$ – 1
  counter = $register_2$

- Consider this execution interleaving with "count = 5" initially:

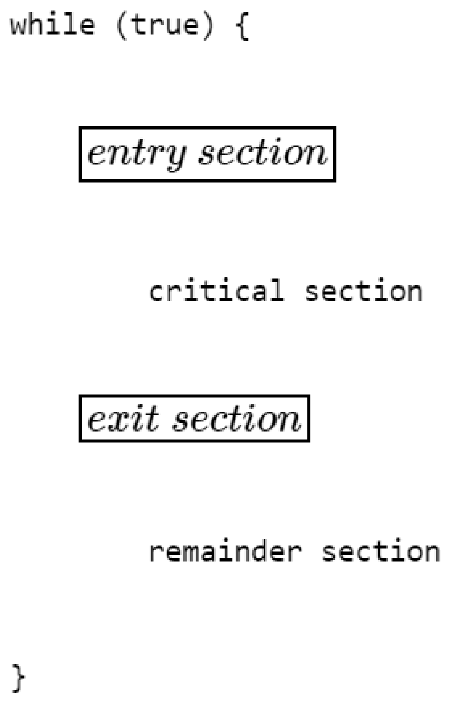
  T_0: producer execute $register_1$ = counter {$register_1$ = 5}
  T_1: producer execute $register_1$ = $register_1$ + 1 {$register_1$ = 6}
  T_2: consumer execute $register_2$ = counter {$register_2$ = 5}
  T_3: consumer execute $register_2$ = $register_2$ – 1 {$register_2$ = 4}
  T_4: producer execute counter = $register_1$ {counter = 6 }
  T_5: consumer execute counter = $register_2$ {counter = 4}

- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

- We devote a major portion of this chapter to process synchronization and coordination among cooperating 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



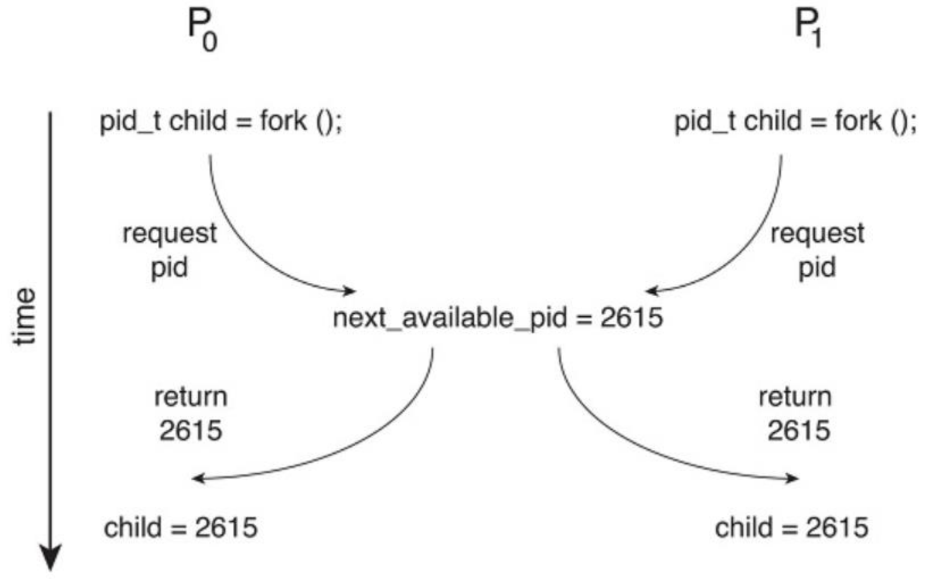
```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process Pi 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 processes 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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the n processes

Solution to Critical-Section Problem

- Race condition on the variable kernel variable **`next_available_pid`**



- No problem in single core environment
  - Achieve by prevent interrupts from occurring while a shared variable was being modified
  - Time consuming and thus infeasible

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  - Preemptive – allows preemption of process when running in kernel mode
  - Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Peterson’s Solution

- Good algorithmic description of solving the problem
- Two-process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = `*`true`*** implies that process $\mathbf{P}_i$ is ready!

Peterson's Solution

- Algorithm for Process P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn = = j);
         critical section
   flag[i] = false;
         remainder section
 } while (true);
```

- 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

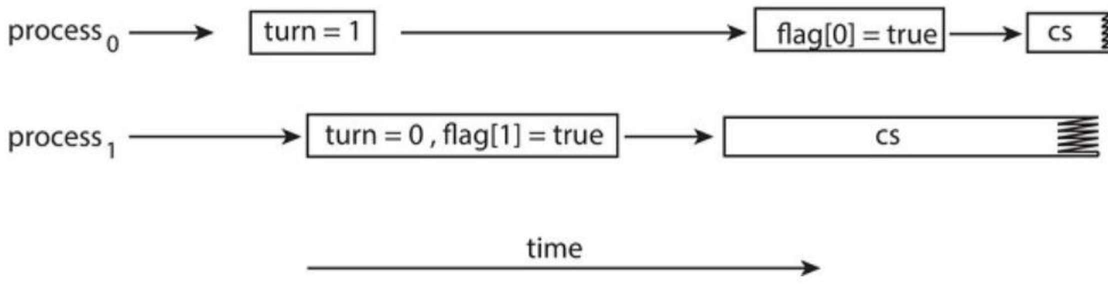
Peterson’s Solution

- Peterson's solution is not guaranteed to work on modern computer architectures.
  - Processors and/or compilers may reorder read and write operations with no dependency for performance.
- Consider the following example:
  - Shared data

    ```
    boolean flag = false;
    int x = 0;
    ```
  - Thread 1

    ```
    while (!flag);
    print x;
    ```
  - Thread 2

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



- What are the results?

Hardware Support for Synchronization

- The solution we have just described is software-based.
  - No special support from the operating system
  - No specific hardware instructions to ensure mutual exclusion
  - Not necessary work on contemporary computer architectures

Memory Barriers

- How a computer architecture determines what memory guarantees it will provide to an application program is known as its memory model, which has two categories:
  - Strongly ordered- memory modification is immediately visible
  - Weakly ordered- memory modification is **not** immediately visible
- Memory models vary- we cannot assume the type.
- Addressed by memory barriers (memory fences)- instructions that can force any changes in memory to be propagated to all other processors.

Memory Barriers

- Consider the previous example with memory barriers
  - Thread 1
    ```
    while (!flag)
       memory_barrier();
    print x;
    ```
  - Thread 2
    ```
    x = 100;
    memory_barrier();
    flag = true;
    ```
- Memory barriers are low-level operations
  - Typically only used by kernel developers

Hardware Instructions

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
- **`test_and_set()`**: test memory word and set value
- **`compare_and_swap()`**: swap contents of two words

Hardware Instructions

- **`test_and_set()`** instruction:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Hardware Instructions

- Use of **`test_and_set()`** for processor P_i:
  - Shared Boolean variable **`lock`**, initialized to **`false`**

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

Hardware Instructions

- **`compare_and_swap()`** CAS instruction:

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

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Hardware Instructions

- Use of **`compare_and_swap()`** for processor P_i:
  - Shared integer **`lock`** initialized to 0

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

Hardware Instructions

- Bounded-waiting mutual exclusion with **`test_and_set()`**:
  - Shared data:
    ```
    boolean waiting[n];
    int lock;
    ```
  - Solution
    ```
    while (true) {
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
            key = test_and_set(&lock);
        waiting[i] = false;
        /* critical section */
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
            j = (j + 1) % n;
        if (j == i)
            lock = false;
        else
            waiting[j] = false;
        /* remainder section */
    }
    ```

Hardware Instructions

- Bounded-waiting mutual exclusion with **`compare_and_swap()`**:
  - Shared data:

```
boolean waiting[n];
int lock;
```

  - Solution

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

Atomic Variables

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

- Typically, the **`compare_and_swap()`** instruction is used to build tools for solving critical-section problem such as the atomic variable.
  - Provide atomic operation
  - Ensure mutual exclusion
- For example,
  `increment(&sequence);`
  increment the atomic integer sequence as follows:

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

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock (abbr. for **mut**ual **ex**clusion)
- Protect a critical section by first acquire() a lock then release() the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (TRUE);
```

Mutex Locks

- acquire() and release()
  - Boolean variable indicating if lock is available or not

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
```

```
release() {
   available = true;
}
```

Mutex Locks

- But this solution requires busy waiting
  - This lock therefore called a spinlock
- Lock contention
  - Contended- a thread blocks while trying to acquire the lock
    - High contention decreases concurrency.
  - Uncontended- a lock is available when a thread attempts to acquire it
- What is meant by “short duration”?
  - Generally, less than 2 context switches (one to wait and one to restore)

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()`**
    - Represent proberen, and verhogen
- Definition of the **`wait()`** operation

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

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

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

Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value ranges between 0 and 1
  - Same as a mutex lock
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**`synch`**" initialized to 0

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

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- Two operations:
  - sleep– place the process invoking the **`wait()`** on the appropriate waiting queue if the semaphore value is not positive
  - wakeup – remove one process in the waiting queue and place it to the ready queue when some process executes **`signal()`**
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

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

Semaphore Implementation

- `wait()` semaphore operation:

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

link field in each PCB

system call

- `signal()` semaphore operation:

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

system call

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

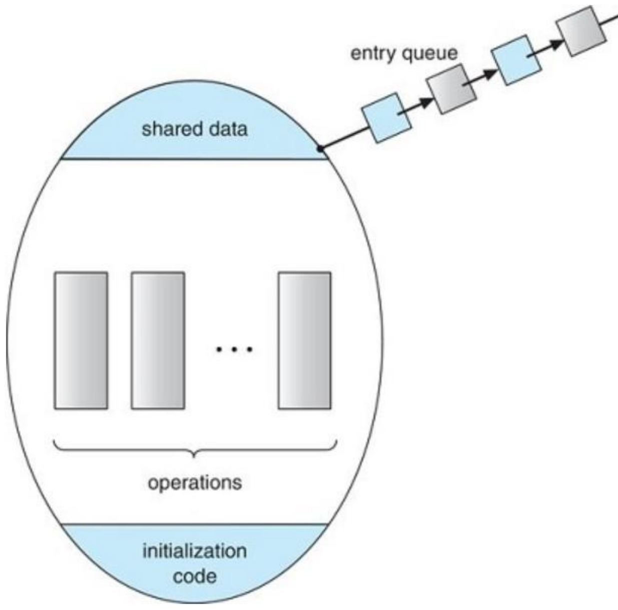
Monitors

- Incorrect use of semaphore operations:
  - signal(mutex); … critical section … wait(mutex);
  - wait(mutex); … critical section … wait(mutex);
  - Omit of wait (mutex) or signal (mutex) or both
- Address by incorporating simple synchronization tool as high-level language constructs: the monitor type.

Monitor Usage

- 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

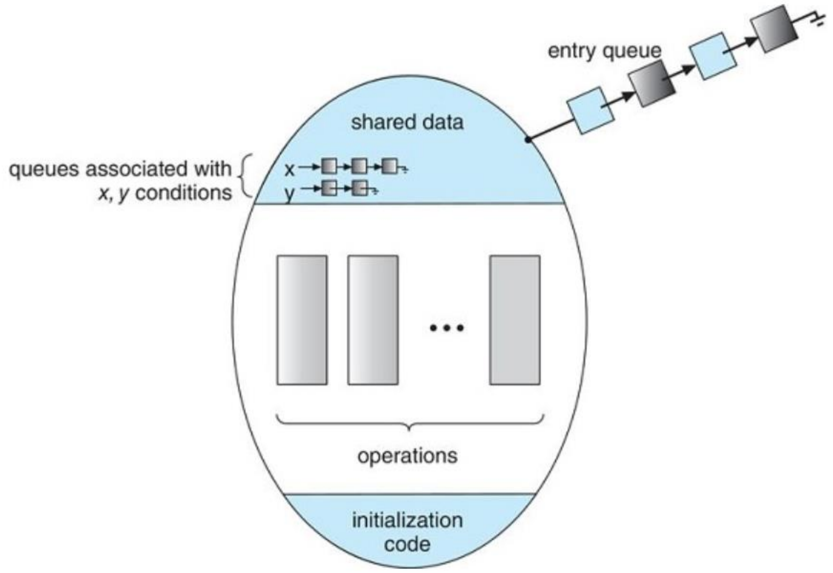
```
monitor monitor-name {
    /* shared variable declarations */
    function F1 (…) {…}

    function Fn (…) {…}

    initialization_code (…) {…}
}
```



- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

Monitor Usage

- Condition Variables
  - 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 Usage

- Condition Variables Choices
  - If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
    - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
  - Options include
    - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
    - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
    - A compromise: P executing signal immediately leaves the monitor, Q is resumed
    - Implemented in other languages including C#, Java

Implementing a Monitor Using Semaphores

- Variables
```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```
- Each function ***F*** will be replaced by
```
wait(mutex);
   …
   body of F;
   …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```
- Mutual exclusion within a monitor is ensured

Implementing a Monitor Using Semaphores

- 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--;
```

- 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 **`x`**, and **`x.signal()`** executed, which should be resumed?
  - FCFS frequently not adequate
- conditional-wait construct of the form **`x.wait(c)`**
  - Where **`c`** is priority number
  - Process with lowest number (highest priority) is scheduled next

Resuming Processes within a Monitor

- Resource Allocator

```
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;
   }
}
```

Resuming Processes within a Monitor

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

```
R.acquire(t);
   …
access the resource;
   …
R.release();
```

  Where R is an instance of type ResourceAllocator

- The monitor concept cannot guarantee that the preceding access sequence will be observed.
  - Access without first gaining access permission
  - Never release once it has been granted access
  - Attempt to release yet it never requested.
  - Request the same resource twice without first releasing it

Liveness

- Liveness refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.
  - Waiting indefinitely is an example of a “liveness failure.”
- Forms of liveness failure
  - Infinite loop
  - Busy wait presents the possibility of starvation
  - Deadlock
  - Priority inversion

Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                          P1
     wait(S);                    wait(Q);
     wait(Q);                    wait(S);
       ...                         ...
    signal(S);                  signal(Q);
    signal(Q);                  signal(S);
```

Priority Inversion

- Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- For example, consider three processes L, M, and H
  - Priorities: *L* < *M* < *H*
  - *H* requires semaphore *S* belonging to *L*
  - If *M* preempts *L*: it decides how long *H* must wait for *L*
- Solved via priority-inheritance protocol
  - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished using the resources.
  - Let *L* inherit the priority of *H* during the time when holding *L*

Evaluation

- There has been a recent focus on using the CAS instruction to construct lock-free algorithms that provide protection from race conditions without requiring the overhead of locking.
  - CAS-based approaches are considered as an *optimistic* approach
  - Mutex lock is considered a *pessimistic* strategy
- The following guidelines identify general rules concerning performance differences
  - Uncontended. CAS protection will be somewhat faster than traditional synchronization.
  - Moderate contention. CAS protection will be faster—possibly much faster—than traditional synchronization.
  - High contention. Under very highly contended loads, traditional synchronization will ultimately be faster than CAS-based synchronization.

Evaluation

- The choice of a mechanism that addresses race conditions can also greatly affect system performance.
  - Atomic integers is good at sharing counters.
  - Spinlock are held for short duration.
  - Mutex locks better binary semaphores.
  - Counting semaphore is proper for controlling access to resources than mutex locks.
- Much ongoing research toward developing scalable, efficient tools that address the demands of concurrent programming.
  - Design compilers able to generate more efficient code.
  - Develop languages which support concurrent programming.
  - Improve the performance of existing libraries and APIs.