Chapter 6: Synchronization

6.4 | Hardware Support for Synchronization:

Why have hardware support in the first place?

The Peterson's solution is one software-based solution to the critical section problem. Software-based solution because an algorithm involves to special support from the operating system or any specific hardware instructions. But refer to 6.3 notes where we discussed why Peterson's solution does not work on modern operating systems, leads to unreliable data states.

6.4.1 | Memory Barriers:

Memory Model: How a computer architecture determines what memory guarantees it will provide to an application program.

> *Strongly Ordered:* Change the memory on one processor and it is immediately visible to other processors.
>
> *Weakly Ordered*: Change the memory on one processor and it may not be immediately visible to other processors.

Memory Barriers & Memory Fences: Memory model varies processor to processor, so you can't make assumption on the change's visibility. But you can force them any change to propagate through processor to processor with instructions.

What does a Memory Barrier do?

System makes sure that all loads and stores are completed before any other load or store is performed. Even if you re-organize the order of these loads and stores, the memory barrier makes sure that the operations are finished in memory and 'visible' to other processors before any more load and store are done.

6.4.2 | Hardware Instructions:

Systems provide hardware instructions that let us test and modify contents of a word or swap the contents atomically. You can use this same stuff to sole the critical-section problem.

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

How is mutual exclusion tackled?

Mutual exclusion using CAS can be provided as follows: A global variable (lock) is declared and is initialized to 0. The first process that invokes compare and swap() will set lock to 1. It will then enter its critical section, because the original value of lock was equal to the expected value of 0. Subsequent calls

to compare and swap() will not succeed, because lock now is not equal to the expected value of 0. When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.

But the bounded-waiting requirement is not tackled…

```
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 */
}
```

This algorithm meets all the following requirements:

Mutual Exclusion:

The value of key can become 0 only if the compare and swap() is executed. The first process to execute the compare and swap() will find key == 0; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

Progress:

Since a process exiting the critical section either sets lock to 0 or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

Bounded-waiting:

When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n − 1 turns.

6.4.3 | Atomic Variables:

Typically, you don't just use the compare_and_swap() instruction to provide mutual exclusion directly. It is just a basic building block for building other tools to solve the critical-section problem. A tool like the … **Atomic Variable**!

Atomic Variable: Affects basic data types such as integers and Booleans. It can help if there is a data race condition on a single variable being updated. But this doesn't solve race conditions under all circumstances.

> *Consider a situation in which the buffer is currently empty and two consumers are looping while waiting for count > 0. If a producer entered one item in the buffer, both consumers could exit their while loops (as count would no longer be equal to 0) and proceed to consume, even though the value of count was only set to 1.*

6.5 | Mutex Locks:

The hardware-based solutions are not accessible to application programmers. OS designers use higher-level software tools to solve the critical-section problem.

Mutex Lock: It protects critical-sections and thus prevent race conditions. A process must acquire the lock before entering a critical section, it releases the lock when it exits the critical section.

Why Mutex Lock isn't that great?

The acquire requires a busy waiting, which waste CPU cycles. If a process is it's critical section, and another process wants in, the other process that tries to enter its critical section must loop continuously in the call to acquire(). It is called a spinlock because the process spins waiting for the lock to become available.

6.6 | Semaphores:

Semaphore: An integer variable that apart from initialization is accessed only through two standard atomic operations: wait() and signal().

What does that mean, atomic operation?

That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

6.6.1 | Semaphore Usage:

OS can see a counting semaphore, which can range over an unrestricted domain, or a binary semaphore, which can be a 0 or 1.

Doesn't a binary semaphore then just act like a mutex lock? Yes.

In a counting semaphore, the semaphore is initialized to the number of resources available. Each resources has to wait(), which decreases the count. When count is 0, all resources are being used. After that, processes that wish to use a resource will block until count is greater than 0.

6.6.2 | Semaphore Implementation:

How does a semaphore overcome the busy waiting problem of a Mutex Lock?

To overcome this problem, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend

operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

Can a semaphore be negative?

If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Why are semaphores not that great?

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place, and these sequences do not always occur.