Software Synchronization

Race Conditions

- A **race condition** is a situation where **several processes or threads** attempt to **manipulate the same data**, and the **outcome** of the execution **depends on the particular order the access takes place**.
- The **section of code** where a **process or thread** accesses a **shared resource** is referred to as the **critical section**.
- To **prevent race conditions**, we use **synchronization**. This ensures that no more than **one process** is able to execute it's **critical section** at a time.

Syncronization Solution Requirements

- Any **solution** to **synchronization** should satisfy the following requirements:
  1. The solution should have **mutual exclusion (mutex)**: Only **one process** can execute it's **critical section** at any given time.
  2. The solution should have **progress**: When **no process** is currently in a **critical section**, **any process** that **requests entry** into the **critical section** must be **permitted without delay**.
  3. The solution must **prevent starvation (bounded wait)**: There is an **upper bound** on the number of times a **process enters the critical section** while **another is waiting**.

  – Such a solution will prevent race conditions.

Synchronization Solutions

- A **simple** way to **achieve synchronization** is the use interrupts; when **interrupts are disabled, context switches will not happen**.
  – This is not a good solution, becuase user processes generally cannot disable interrupts.
  – This also does not work on a multi-core system.
- Another way to **achieve synchronization** is by using a **variable as a flag** to **determine** if any other processes are executing their **critical section**. When a process wants to **execute their critical section** and another process has already locked the resource, the process will use a **conditional loop until the lock is removed**.
  – This acheives mutual exclusion, but wastes a lot of processor time.
  – This also does not prevent starvation.
- **Modern computers** use **hardware solutions** to **synchronize processes**.

Hardware Syncronization

Hardware Solution

- The **hardware solution** always executes the **lock check, and set** in the same intruction. This avoids a **context-switch** inbetwwen, which would not allow for mutual exclusion.
- This solution is called a **Test and Set Intruction**. It works by writing to a memory location, and returning its old value.

```
// Instruction provided by hardware,
//not program code.
// You use the return value to determine
//if the lock is set (0 => lock is unset, 1 => lock is set).
int test_and_set(int *L) {
    int prev = *L;
    *L = 1;
    return prev;
}
```

- If **two processors** execute the **test-and-set instruction** at the same time, the **hardware ensures** that one **test-and-set** does **both** of it's steps before the other one starts.

Intel HD Support (xchg Instruction)

- **Intel CPUs** have an **xchg instruction**. If two CPUs execute **xchg** at the same time, the hardware ensures that only one **xchg completes**, before the second begins.
- The **xchg** instruction takes in two registers as **two operands** and **swaps their values** atomically. This behaviour acts identically to the test-and-set solution.

Hgh Level Constructs

- A **Spinlock** is a mechanism for locking a resource. It works as follows: One process will **acquire the lock**, adn the other processes will wait in a loop repeatedly checking if the lock is available, once the first process **releases the lock**, other processes will **acqire it**.

```
void acquire(int *locked) {
    int val = 1;
    while (1) {
        xchg(locked, &val)

        if(val == 0)
            break;
    }
}

void release(int *locked) {
    *locked = 0;
}
```

  - This mechanism is inefficient, becuase the CPU will waste time executing the loops while other processes wait for the lock to be released.
  - It is useful for short critical sections, but not when the period of wait in unpredicatable or will take a long time.

- A **Mutex** is a mechanism for locking a resource, if the lock is set, other processes that need accesses to the lock resource will **sleep** until it is release. When the processes that has **currently acquired the lock releases it**, it will **wake up the other sleeping processes**.

```
void lock(int *locked) {
    while (1) {
        while (1) {
            xchg(locked, &val)
```

```
                if(val == 0)
                    break;
                else
                    sleep();
            }
        }

        void release(int *locked) {
            *locked = 0;
            wakeup();
        }
    }
```

- This is more efficient, as it does not waste the CPU's time.
- There is a problem with this solution, known as the **Thundering Herd Problem**: If there are a large number of processes waiting to access the resource, and wakeup is called, all of the processes wake up at the same time. This leads to a large amount of context-switches. If more processes continue to wait, this could lead to **starvation**.
- The **Thundering Heard Problem** can be solved by placing the processes waiting to access the resource in a **queue**.

- A **Semaphore** is mechanism for managing shared memory, that is used to produce and consume data (i.e. to solve the problem of the producer, or consumer being faster).
- A **Semaphore** uses an unsigned integer with two functions: wait, and post. The wait function is used to wait until the counter goes above 0, decrements it and pulls 1 data entry. The post function is used when new data is being posted, and increments the counter.

```
void wait(int *S) {
    while (*S <= 0);
    *S--;
}

void post(int *S) {
    *S++;
}
```

- **Semaphores are atomic**, if one **process** or **thread** increments the integer, and another decrements, the two operations **cannot interrupt each other**.
- We can use **semaphores** to solve the **critical section probelm** by having a **binary semaphore**, but is it not recommended because anythread can unlock it instad of the thread executing it's critical section.