ICS431 Ch. 6 & 7

Alfaifi, Ammar

Chapter 1

Synchronization Tools

1.1 Background

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space or be allowed to share data through message passing.
- Concurrent access to shared data may result in data inconsistency, so synchronization is a must!
- Recall the **unbound buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always add new items.
- For the **bounded buffer**, it assumes a fixed buffer size. The consumer must wait if the buffer is empty, and the producer must wait if it's full.
- We consider the code of bounded buffer from Section 3.5, we add a new variable `count`, to track the number of items in the buffer.
- As you see in the consumer's and producer's codes, page 274, the producer increment `count` and the producer tries to decrement it each time. When running concurrently, there are clashing!
- When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particulate order in which the access takes place is called **race condition**.
- Since we are interleaving the lower-level statements of both codes (but the order withing each high-level is preserved), both `count++` and `count--` are executed using the same old value, resulting wrong result; this is a race condition.

1.2 The Critical-Section Problem

- Consider a system with n processes $P_0, P_1, \ldots, P_{n-1}$. Each process ha a segment of code is a **critical section**, meaning a process can modify a data that's shared with at least one other process.
- The section of code implementing this request is called **entry section**. There can be also **exit section**.
- A solution to he critical-section problem must satisfy the following

1. **Mutual exclusion** if process P_i is executing it its critical section, no other processes can be executing in their critical sections.
2. **Progress** if no process is executing in its critical section and some processes want to enter their critical sections, then only those aren't executing in their remainder section can decide which will enter its critical section next.
3. **Bounded waiting** a bound (limit) on the number of times that other processes are allowed to enter their critical section after a process made a request to enter its critical section and that request.

- Let `next_available_pid` be the pid to be assigned to a process after calling `fork()`. If `fork()` is called twice these will lead to a race condition in a kernel data structure.
- This problem can be avoided in single-core environment, by disabling interrupts while a modifying a shared variable. But this inefficiency solution, and not feasible in multiprocessor environment.
- Two approaches are used to handle this problem:
  1. **Preemptive kernels** allows a process to be preempted while it is running in kernel mode.
  2. **Nonpreemptive kernels** does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or yields control of CPU back.
- Obviously, nonpreemptive doesn't have the race condition problem on kernel data structure, since only one process at time can run.

1.3 Peterson's Solution

- This is a software-based solution to the critical-section problem.
- It requires the two processes to share two data items as, written in `C++`

```
int turn;
boolean flags[2];
```

- `turn` indicates whose turn it is to enter its critical section. if `turn == i`, then process P_i is allowed to execute its critical section.
- `flag` array indicates if a process is ready to enter its critical section; if `flag[i]` is `true`, P_i is ready to enter its critical section.
- The process P_i code should be

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

/* Here is the critical section */

flag[i] = false;

/* remainder section */
}
```

- A downside of the solution is that, it is affected by compilation process where reordering for the read and write operation may happen.

1.4 Hardware Support for Synchronization

1. **Memory Barriers**

   - We saw that system may reorder instructions, leading to unreliable data states.
   - **memory model** is how a computer determines what memory should provide to an application program.
     (a) *Strongly ordered*: A memory modification in one process is visible immediately to all other processes.
     (b) *Weakly ordered*: Modifications to memory on one processor may not be visible to others.
   - Memory models vary by process*or*, so kernel developers cannot make assumptions if the visibility of modifications to memory on a shared-memory multiprocessor.
   - computer architectures provide instructions that can *force* any changes in memory to be propagated to all other possessors, they're called **memory barriers**
   - The idea is that when a memory barrier instruction is performed, the system ensures all `load` & `store` are completed before any coming `load` and `store` are executed.
   - Consider this code

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

     we ensure the assignment to `x` occurs before the assignment to `flag`.

2. **Hardware instructions**

   - Modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**, as one uninterruptible unit.
   - Then we use these special instructions to solve the critical-section problem.
   - In abstract, consider this code

     ```
     boolean test_and_set(boolean *target) {
         boolean rv = *target;
         *target = true;

         return rv;
     }
     ```

   - This instruction is a low-level atomic operation used in operating systems and multi-threaded programming. It is used to set a flag or a lock on a shared resource while checking if it was previously set. The instruction performs the following steps atomically:
     (a) Read the current value of the flag or lock.
     (b) Set the flag or lock to the desired value.
     (c) Return the previous value of the flag or lock.

- This operation is used to implement synchronization primitives such as mutexes, semaphores, and spin locks. It ensures that multiple threads or processes do not access the shared resource simultaneously, preventing race conditions and other synchronization issues.
- The main characteristic of this instruction is that it is executed atomically; if two `test_and_set()` instructions are executed simultaneously, they will be executed sequentially in arbitrary order.
- If the machine support this instruction, then we can implement *mutual exclusion* by declaring a boolean variable `lock`, initialized to `false`. The structure of process P_i becomes

```
do {
    while (test_and_set(&lock))
        ;

        /* critical section */

        lock = false;

        /* remainder section */
} while(true);
```

- `compare_and_swap()` (CAS) is another low-level atomic operation It is used to update the value of a memory location atomically, while also checking if the current value of the memory location matches an expected value.
- It takes three arguments: a pointer to the memory location to be updated, the expected value of that memory location, and the new value to be written to that memory location. The operation will only update the memory location if its current value matches the expected value. It is defined as

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

    return temp;
}
```

- And the process structure becomes

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ;

        /* critical section */

        lock = 0;

        /* remainder section */
}
```

- A global variable, `lock`, is declared and initialized to 0. The first process invokes `compare_and_swap()` will set `lock = 1` and enters its critical section.
- Any next calls to `compare_and_swap()` will fail, because `lock` now is not equal to the expected value of 0.

- This algorithm does not satisfy the bounded-waiting requirement.

3. **Atomic variables**
   - They provide atomic operations on basic data types such as integers and booleans.
   - Atomic variables can be implemented using low-level hardware instructions such as test-and-set, compare-and-swap, and fetch-and-add. In modern programming languages, atomic variables are typically provided as language-specific constructs, such as `std::atomic` in C++ or `Atomic` in Java.
   - Note that they don't solve the problem of bounded-buffer problem, where to consumer could be triggered at the same time even if there is only one item in the buffer.

1.5 Mutex Locks

- Since the hardware-based solution to the critical-section problem are quite complicated, OS designers build a software-level tools to solve it; such as **mutex lock** (from **mu**tual **ex**clusion)
- A process must acquire a lock before entering critical section and releases it back once it exits.
- A boolean variable `available`. If the lock is available, a call to `acquire()` succeeds.
- Call to either `acquire()` or `release()` must be performed atomically. Meaning mutex locks should use CAS operation.

1.6 Semaphores

- A **semaphore** `S` is an integer variable that, excluding the initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`.
- Again all modifications to the integer value of the semaphore in the `wait()` and `signal()` must be done atomically. Meaning, no other process can simultaneously modify the same semaphore value.
- When a process wishes to use a resource it calls `wait()`. When it releases a resource, it calls `signal()`. If all resources are in use, `S = 0`, the process gets block.

1.7 Liveness

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely.
- A process is in a **deadlock** state when every process in the set is waiting for an event that can be caused only by another process in the set.