Justin Ciocoi

Oct. 20, 2023

CSCI 375 Operating Systems

Chapter 6 Synchronization Tools

Textbook Notes

Section 6.1: Background

- This chapter will largely focus on how concurrent or parallel execution can pose a threat to the integrity of data shared by several processes
- We can first return to the example of a bounded-size-buffer from chapter 3:

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

- A corresponding consumer process is also similarly defined
- The consumer process will end with the line `count--`, which can lead to a data race for the shared variable count, since the `count++` and `count--` lines are executed as three lines in machine code and can be jumbled, resulting in incorrect or inconsistent output
    - This is called a *race condition*
- Because of issues such as these, much of this chapter will focus on *process synchronization* and *coordination*

6.2: The Critical Section Problem

- Consider a system consisting of n processes, $[P_0, P_1, ..., P_{n-1}]$
- In each process, there is a section of code called the *critical section* during which a process may be accessing or updating a data member that is shared with other processes
- The *critical section problem* refers to the problem which we will attempt to find solutions to in this section:
    - How can we design a *protocol* such that processes can synchronize their activity in order to *cooperatively share data*
- Generally, this will be implemented in a section preceding the section called an *entry section*
- The *critical section* can also be followed by an exit section that allows other processes access to shared data
- The *remainder section* contains the rest of the process code that does not need access to shared data

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

- A solution to the critical section problem must satisfy the following three conditions
    - *Mutual Exclusion*
        - If a process, P_i, is executing in its critical section, then no other processes can be executing in their critical sections
    - *Progress*
        - If there is no process executing in its critical section, and some processes wish to enter the critical section, only processes not executing their remainder sections can participate in the decision making

        - The selection of the next process cannot be postponed indefinitely, thus progress is inevitable
    - *Bounded Waiting*
        - There exists a bound, or limit, 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.
- Generally speaking, there are two approaches used to handle critical sections in modern operating systems
    - Preemptive Kernel
    - Nonpreemptive kernel
- A *nonpreemptive* kernel is essentially free from any data race conditions since only one process is active in the kernel at a time
- While a nonpreemptive kernel does provide this advantage, it doesn't account for the possibility of a kernel process executing for an arbitrarily long time leading to low responsiveness on a system
- Therefore, most modern operating systems utilize preemptive kernel approaches on multi-processor systems despite the difficulty in implementation

---

6.3: Peterson's Solution

- There is one classic software-based solution to the critical section problem known as *Peterson's solution*
- Peterson's solution is limited to two processes, so in practice it is not very useful, but it presents a good look at the algorithmic fundamentals for solving the critical section problem
- For this section, we will refer to the two processes as P_0 and P_1
- Petersons solution requires processes to share *two* data items
    - ```
      int turn;
      bool flag[2];
      ```
- Below is the general structure of process P_i where $j = i - 1$
      ```

- ```
  while(true)
  {
      flag[i] = true;
      turn = j;
      while(flag[j] && turn == j)
      {
          //wait
      }

      /*Critical Section*/

      flag[i] = false;

      /*Remainder Section*/
  }
  ```

- Here, the variable `turn` indicates whose turn it is to enter the critical section
  - If `turn==i`, then process P_i is allowed to execute in its critical section
- The `flag[]` array is used to indicate if a process is *ready* to enter its critical section
  - if `flag[i]==true` then P_i is ready to enter its critical section
- In the above code snippet, process P_i first sets `flag[i] = true`, meaning process P_i is ready to enter its critical section and sets `turn = j` to allow process P_j to finish if it is in its critical section
  - `while(flag[j] == true && turn = j)` P_i will wait
- Once P_j has exited its critical section, it will set `flag[j] = false` and P_i will then proceed to its critical section
- As another example, consider the following data that is being shared between two threads
  - ```
    bool flag = false;
    int x = 0;
    ```
- Here, thread 1 performs the statements
  - ```
    while(!flag)
    {};
    print x;
    ```
    ```

- Thread 2 performs the statements
    - ```
      x = 100;
      flag = true;
      ```
- Here, the expected behavior would be that thread 1 outputs the value 100 for variable `x`
- However, machine level instructions may be broken down in an order that leads to *unresolved data dependencies* in the code leading to *inaccurate or inconsistent results* in a program

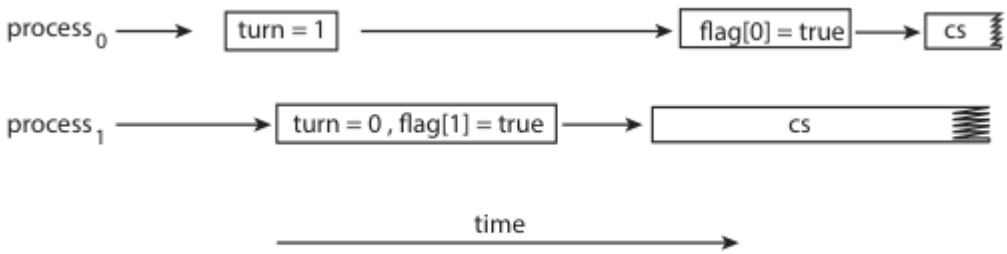


Figure 6.4 The effects of instruction reordering in Peterson's solution.

- In this case, `flag` is assigned a true value before the assignment of `x = 100`
    - Thus, thread 1 will output a value of 0 instead of the expected 100
- For a simple video explanation of Peterson's solution, click here

---

6.4: Hardware Support for Synchronization

- Peterson's solution is a software implementation, as it requires no additional hardware support to function
- This chapter will analyze hardware support for process synchronization
- **6.4.1: Memory Barriers**
    - In the previous section, we noted how a system may reorder machine level instructions, potentially leading to unreliable data states
    - The memory *guarantees* that a system will provide to an application program is determined primarily by a system's ***memory model***
    - Generally, a memory model will fall into one of two categories
        - *Strongly Ordered*, where a memory modification one on processor is immediately visible to all other processors
      ```

        - *Weakly Ordered*, where modifications to memory on one processor may not be immediately visible to other processors
    - To address issues in the communication of memory modifications, computer architectures provide instructions that can *force* any changes in memory to be propagated to all other processors, thereby ensuring that modifications are visible to threads running on other processes
    - These instructions are referred to as **memory barriers** or **memory fences**
        - Whenever a memory barrier instruction is performed, the system will ensure that all loads and stores will be completed before any subsequent load or store operations are performed

- **6.4.2: Hardware Instructions**

    - Many 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*, meaning as one uninterruptible unit

    - Rather than discussing one specific instruction on one specific machine, this section will abstract these main concepts behind the `test_and_set()` and `compare_and_swap()`

    - First, let us define the `test_and_set()` instruction

        - ```
          bool test_and_set(bool *target)
          {
              bool rv = *target;
              *target = true;

              return rv;
          }
          ```

    - Now let us illustrate an implementation of mutual exclusion using the `test_and_set()` function
          ```

```
do{
    while(test_and_set(&lock))
    {
    ; //do nothing
    }

    // critical section

    lock = false;

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

- Similar to the `test_and_set()` function, the `compare_and_swap()` function acts on a word atomically, but differs in its mechanism, which is based on swapping the content of two words
- First is the definition of the `compare_and_swap()` function

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

    if(*value==expected)
        *value = new_value;

    return temp;
}
```

- As we can see by looking at the function, the value of `*value` is replaced by `new_value` only if `*value==expected` is true
- However, the function will return the original value regardless
- Now, we look at an implementation of mutual exclusion using the `compare_and_swap()` function

```
while(true)
{
    while(compare_and_swap(&lock, 0, 1) != 0)
        ;//do nothing

    //critical section

    lock = 0;

    //remainder section
}
```

- In the above, a global variable, `lock`, was declared and initialized to 0
- The first process which invokes the `compare_and_swap()` function will set `lock` to 1
- Now, subsequent calls to the `compare_and_swap()` function will not succeed because `lock` is now not equal to the expected value of 0
- When the process exits its critical section, we can see that it sets the `lock` back to 0, thus allowing other processes to enter their critical sections
- While this code does satisfy the mutual exclusion and progress portions of the critical section problem, bounded waiting is not achieved
- Here, we will present another algorithm using the `compare_and_swap()` instruction that *does* achieve bounded waiting

```
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!=1) && !waiting[j])
        j = (j+1) % n;

    if(j == i)
        lock = 0;
    else
        waiting[j] = false;

    //remainder section
}
```

- In this algorithm, common data structures shared by processes P_i and P_j are
  - `bool waiting[n]`
  - `int lock`
- We can see that when a process exits its critical section, it scans the array waiting in the cyclic ordering, thus achieving bounded waiting

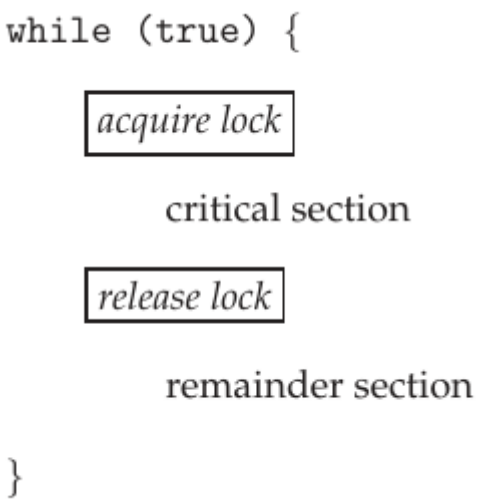
- **6.4.3: Atomic Variables**
  - Typically, the `compare_and_swap()` instruction is not used to directly provide a solution to the critical section problem, but rather to serve as a building block for other tools that aim to solve the critical section problem
  - One of these tools is known as an *atomic variable*, which provides atomic operations on basic data types such as integers and booleans
  - Atomic variables can be used to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.
  - Although atomic variables do provide atomic updates, they do not entirely solve race conditions in all circumstances

- Atomic variables are very commonly used in operating systems as well as in concurrent applications, although their use is usually only for single updates of shared data such as counters and sequence generators

---

6.5 Mutex Locks

- The hardware based solutions that were examined in section 6.4 are largely complicated and inaccessible methods that leave a lot to be desired in the world of application programming
- Instead of using these, operating-system designers use higher-level software tools to solve the critical section problem
- The simplest of these tools is the *mutex lock*, where *mutex* is short for *mut*ual *ex*clusion
- A process must first acquire a lock before entering its critical section, and then release the lock upon completion of its critical section



- If the lock is available, a call to acquire succeeds, and the lock is then considered unavailable
- A process that attempts to acquire an unavailable lock is blocked until that lock is released
- The definition of `acquire()` is as follows

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

- The definition of `release()` is as follows

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

- The calls to either acquire or release a mutex lock must be performed atomically, and are thus implemented using the `compare_and_swap()` function used in section 6.4

6.6: Semaphores

- Mutex locks are generally considered the simplest form of synchronization tools
- In this section, we can examine a more robust tool that can act in a similar fashion to the mutex lock, but can also provide even more sophisticated ways in which process may synchronize their activities
- A ***Semaphore*** `S` is an integer variable that - apart from initialization - can be accessed only through two standard atomic operations, `wait()` and `signal()`
- The definition of `wait()` is as follows

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

    S--;
}
```

- The definition of `signal()` is as follows

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

- Semaphore values can only be modified atomically, that is, by one process at any given time
- **6.6.1: Semaphore Usage**
  - Oftentimes, operating systems will distinguish between counting semaphores and binary semaphores
  - In a *counting* semaphore, the value of the semaphore can range over an unrestricted domain
  - In a *binary* semaphore, the value of the semaphore can range only between 0 and 1
    - This indicates that binary semaphores will operate in a manner fairly reminiscent of mutex locks
  - Now, let us consider two concurrently running processes, P_1 with a statement S_1 and P_2 with a statement S_2 and suppose that S_2 can be executed only *after* S_1 has completed
  - We could implement it in the following way
    - ```
      //Process P1
      {
          S1;
          signal(synch);
      }
      ```
    - ```
      //Process P2
      {
          wait(synch);
          S2;
      }
      ```
- **6.6.2: Semaphore Implementation**
  - The implementations of the mutex lock in section 6.5 suffers from busy waiting, and the above definitions of the `wait()` and `signal()` functions present the same problem

- To overcome this problem, we can redefine the `wait()` and `signal()` semaphore operations
  - ```
    wait(semaphore *S)
    {
        S->value--;
        if(S->value < 0)
        {
            add this process to S->list;
            sleep();
        }
    }
    ```
  - ```
    signal(semaphore *S)
    {
        S->value++;
        if(S->value <= 0)
        {
            remove process P from S->list;
            wakeup(P);
        }
    }
    ```