CSC 4320/6320: Operating Systems

Chapter 06: Synchronization Tools-II

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores

Covered so far

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

Review: Software Solution 1

- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - `int turn;`
- The variable `turn` indicates whose turn it is to enter the critical section
- initially, the value of `turn` is set to *i*

Review: Algorithm for Process P_i

```
while (true){

   while (turn = = j);

   /* critical section */

   turn = j;

   /* remainder section */

}
```

Review: Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  `turn = i`

  and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Questions

1. A(n) _______ refers to where a process is accessing/updating shared data.
A) critical section ✓
B) entry section
C) mutex
D) test-and-set

2. A solution to the critical section problem does not have to satisfy which of the following requirements?
A) mutual exclusion
B) progress
C) atomicity ✓
D) bounded waiting

3. A nonpreemptive kernel is safe from race conditions on kernel data structures.
True ✓
False

Peterson’s Solution

- 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: (requirement)
  - **`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 P_i is ready!

Algorithm for Process P_i

```
while (true){

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

    /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

Correctness of Peterson's Solution

- 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 and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
   ;
print x
```

- Thread 2 performs

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

- What is the expected output?

100

Recall:

Peterson’s solution is not useful for modern computer architectures:

- processors and/or compilers may reorder operations that have no dependencies

Modern Architecture Example (Cont.)

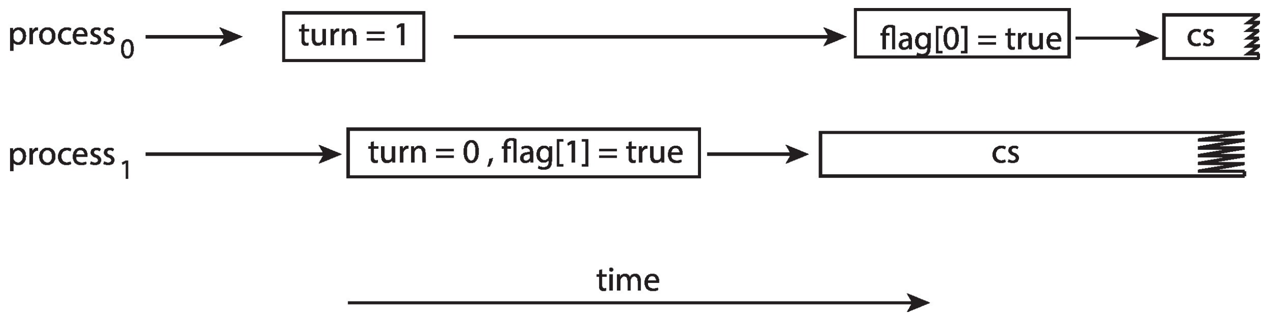
- However, since the variables `flag` and `x` are independent of each other, the instructions:

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

  for Thread 2 may be reordered
- If this occurs, the output may be 0!

Peterson's Solution Revisited

- Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson's solution are reordered;



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

Questions

1. In Peterson's solution, the _____ variable indicates if a process is ready to enter its critical section.

A) turn

B) lock

C) flag[i] ✓

D) turn[i]

2. ______ is not a technique for handling critical sections in operating systems.

A) Nonpreemptive kernels

B) Preemptive kernels

C) Spinlocks

D) Peterson's solution ✓

Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 13/14
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
   while (!flag)
 memory_barrier();
  print x
```

- Thread 2 now performs

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

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally, too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Memory Barriers
2. Hardware instructions
3. Atomic variables

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction
  - These two examples abstract the main concepts behind these types of instructions

The test_and_set Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Solution Using test_and_set()

- Shared boolean variable **`lock`**, initialized to **`false`**
- **`It ensures mutual exclusion.`**
- Solution:

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

           /* critical section */

    lock = false;
           /* remainder section */
 } while (true);
```

- Does it solve the critical-section problem?
  - **`test_and_set()`** does not directly ensure bounded waiting (fairness).

The compare_and_swap Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

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

      /* critical section */

      lock = 0;

     /* remainder section */
 }
```

- Does it solve the critical-section problem?
  - this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement
  - Modification is possible.

A solution that satisfies requirements

The common data structures are

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

- **waiting[i] = false**: This indicates that no thread or process is waiting for the lock initially.
- **lock = 0**: This means the lock is free, allowing the first thread to acquire it.

Bounded-waiting with compare-and-swap

```
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]) // if any process is in entry section
      j = (j + 1) % n; // keep iterating unless you get a process waiting
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;  // so j will no longer wait
   /* remainder section */
}
```

Questions

1. An instruction that executes atomically ____.

A) must consist of only one machine instruction

B) executes as a single, uninterruptible unit ✓

C) cannot be used to solve the critical section problem

D) All of the above

2. Race conditions are prevented by requiring that critical regions be protected by locks.

True ✓

False

3. Both the `test_ and_set()` instruction and `compare_and_swap()` instruction are executed atomically.

True ✓

False

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
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {
   acquire lock

       critical section

   release lock

remainder section
}
```

```
acquire() {

   while (!available)

     ; /* busy wait */

   available = false;

}

release() {

  available = true;

}
```

Questions

1. A mutex lock ____.

A) is exactly like a counting semaphore

[✓ B) is essentially a boolean variable]

C) is not guaranteed to be atomic

D) can be used to eliminate busy waiting

2. What is the correct order of operations for protecting a critical section using mutex locks?

A) release() followed by acquire()

[✓ B) acquire() followed by release()]

C) wait() followed by signal()

D) signal() followed by wait()

3. Busy waiting refers to the phenomenon that while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire the mutex lock.

[✓ Yes]

No