Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Synchronization Tools-Part2

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Spring 2022

Peterson’s Solution

```
//P0
while (true){
   flag[0] = true;
   turn = 1;
    while (flag[1] && turn == 1);
    /* critical section */
    flag[0] = false;
    /* remainder section */
}
```

```
//P1
while (true){
   flag[1] = true;
   turn = 0;
    while (flag[0] && turn == 0);
    /* critical section */
    flag[1] = false;
    /* remainder section */
}
```

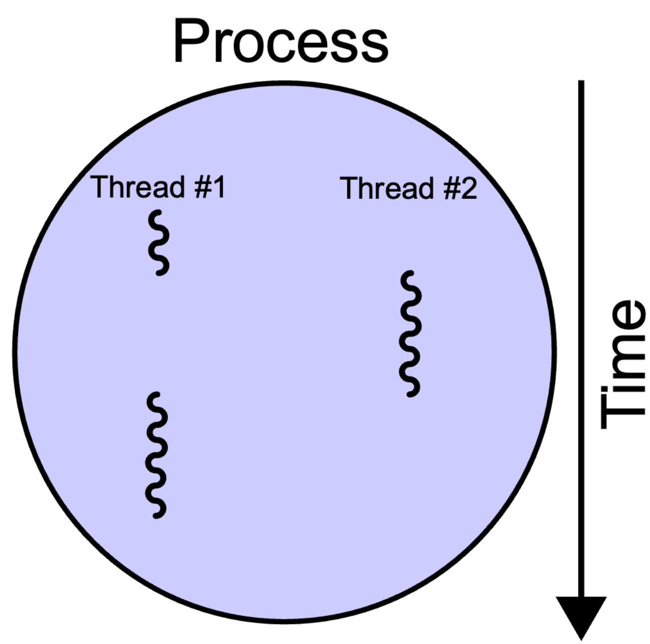
Peterson's Solution and Modern Architecture

- 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.**

Peterson's Solution and Modern Architecture

- 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
```

Modern Architecture Example (cont.)

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

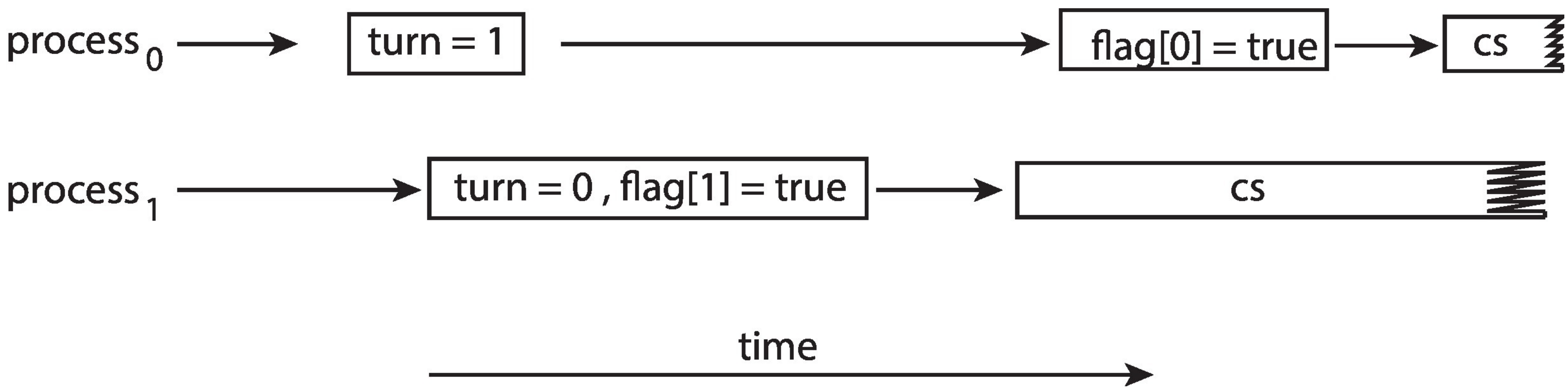
```
flag = true;
 x = 100;
```

  for Thread 2 may be reordered

- **If this occurs, the output may be 0!**

Peterson’s Solution Revisited

- The effects of instruction reordering in Peterson’s Solution

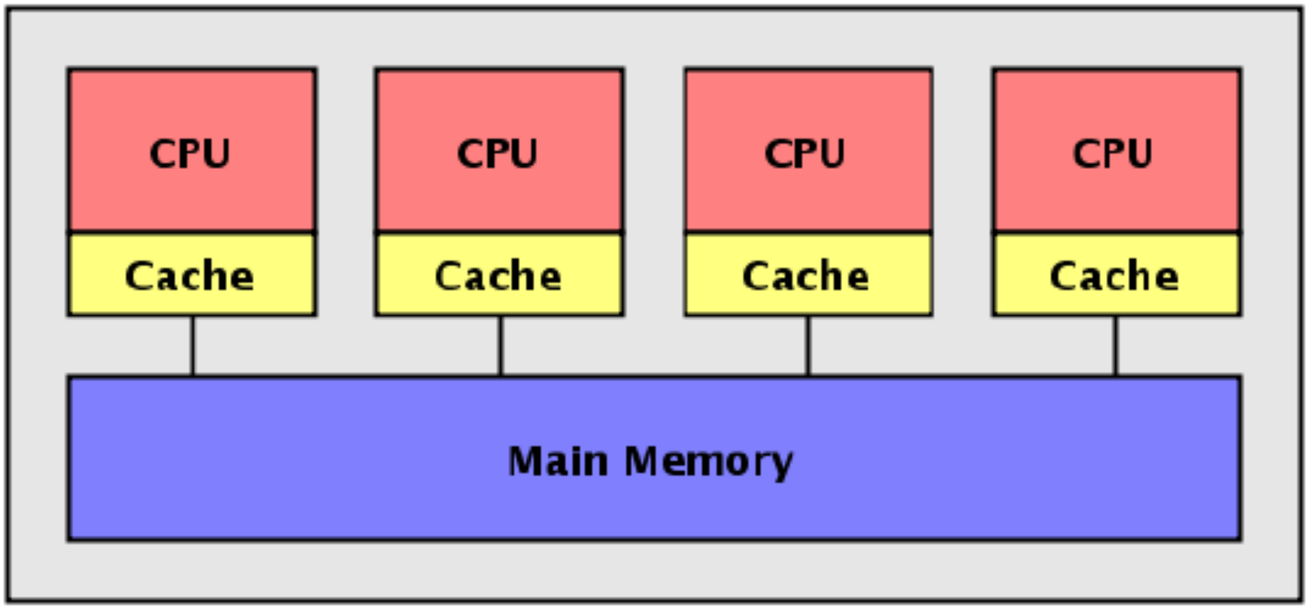


- **This allows both processes to be in their critical section at the same time!**
- To ensure that Peterson’s solution will work correctly on modern computer architecture we must use **Memory Barrier**.

Hardware Support for Synchronization

Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.



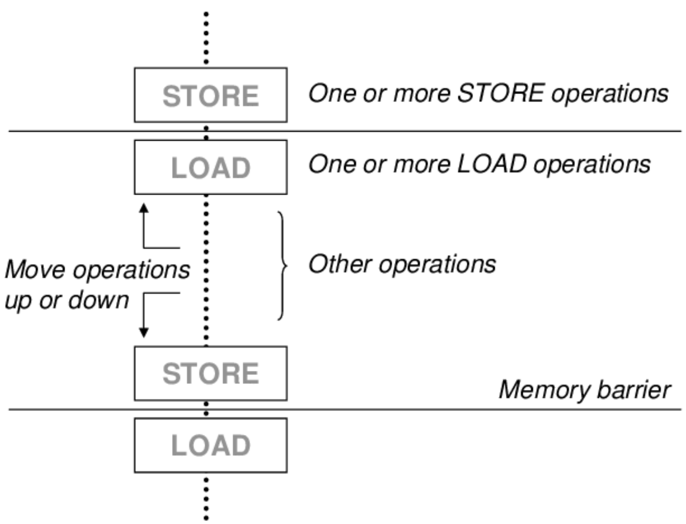
https://www.researchgate.net/figure/Shared-memory-architecture_fig1_272377248

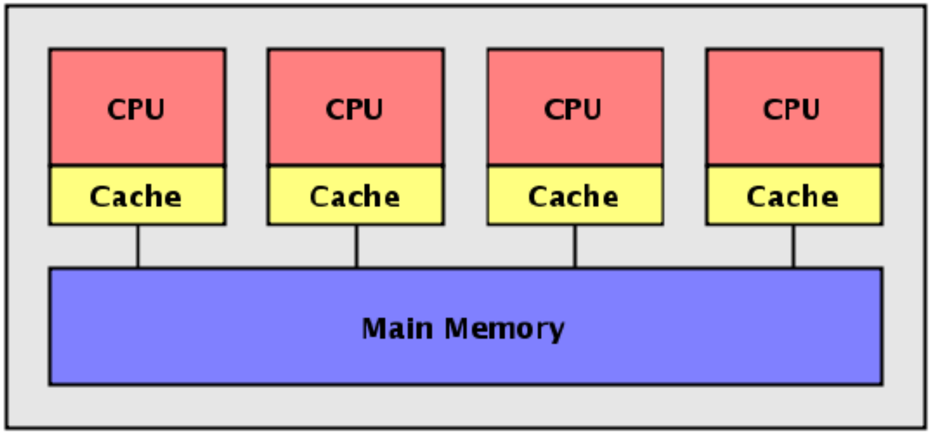
Memory Barrier (cont.)

- Memory models may be either:
  - **Strongly ordered**
    - Memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered**
    - Memory modification of one processor may not be immediately visible to all other processors.

Memory Barrier (cont.)

- 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 5-6

- We could add a memory barrier (as follows) to ensure Thread 1 outputs 100.

Memory Barrier Example (cont.)

- Thread 1 now performs

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

- Thread 2 now performs

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

- For Thread 1 → the value of flag is loaded before the value of x.
- For Thread 2 → the assignment to x occurs **before the assignment flag**.

Memory Barrier for Peterson's solution

Where should we add memory barrier?

```
//P0
1. while (true){
2.    flag[0] = true;
3.    turn = 1;
4.     while (flag[1] && turn == 1);
    /* critical section */
5.     flag[0] = false;
    /* remainder section */
}
```

```
//P1
1. while (true){
2. flag[1] = true;
3. turn = 0;
4.     while (flag[0] && turn == 0);
    /* critical section */
5.     flag[1] = false;
    /* remainder section */
}
```

Memory Barrier for Peterson's solution (Cont.)

We could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in the previous slide.

Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

Synchronization Hardware Support

- 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 two forms of hardware support:

1. **Hardware instructions**
2. **Atomic variables**

Hardware Instructions

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

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`**

```
do{
 while (test_and_set(&lock)); /*do nothing */
      /* critical section */
      lock = false;

      /* remainder section */

} while (true);
```

- **Does it solve the critical-section problem?**

Requirement	Yes/No
Mutual Exclusion	
Progress	
Bounded waiting	

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.

Solution using compare_and_swap

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

```
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?**

Requirement	Yes/No
Mutual Exclusion	
Progress	
Bounded waiting	

Second solution using compare-and-swap

- The common data structures are:

```
boolean waiting[n];

int lock;
```

- The elements in the ***waiting*** array are initialized to ***false***
- Variable ***lock*** is initialized to ***0***.

Second solution using 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 */
    …
}
```

Second solution using compare-and-swap

```
while (true) {
      …
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Requirement	Yes/No?
Mutual Exclusion	
Progress	
Bounded waiting	

Synchronization Hardware Support

- **Hardware instructions**
  - test_and_set()
  - Compare_and_swap()
- **Atomic variables**
  - We unfortunately do not have enough time to cover this
  - Please read the related section in the reference book