OPERATING SYSTEMS CS3500

PROF. SUKHENDU DAS DEPTT. OF COMPUTER SCIENCE AND ENGG., IIT MADRAS, CHENNAI – 600036.

Email: sdas@cse.iitm.ac.in
URL: http://www.cse.iitm.ac.in/~vplab/os.html

Aug. – 2022.

SYNCHRONIZATION TOOLS

OUTLINE

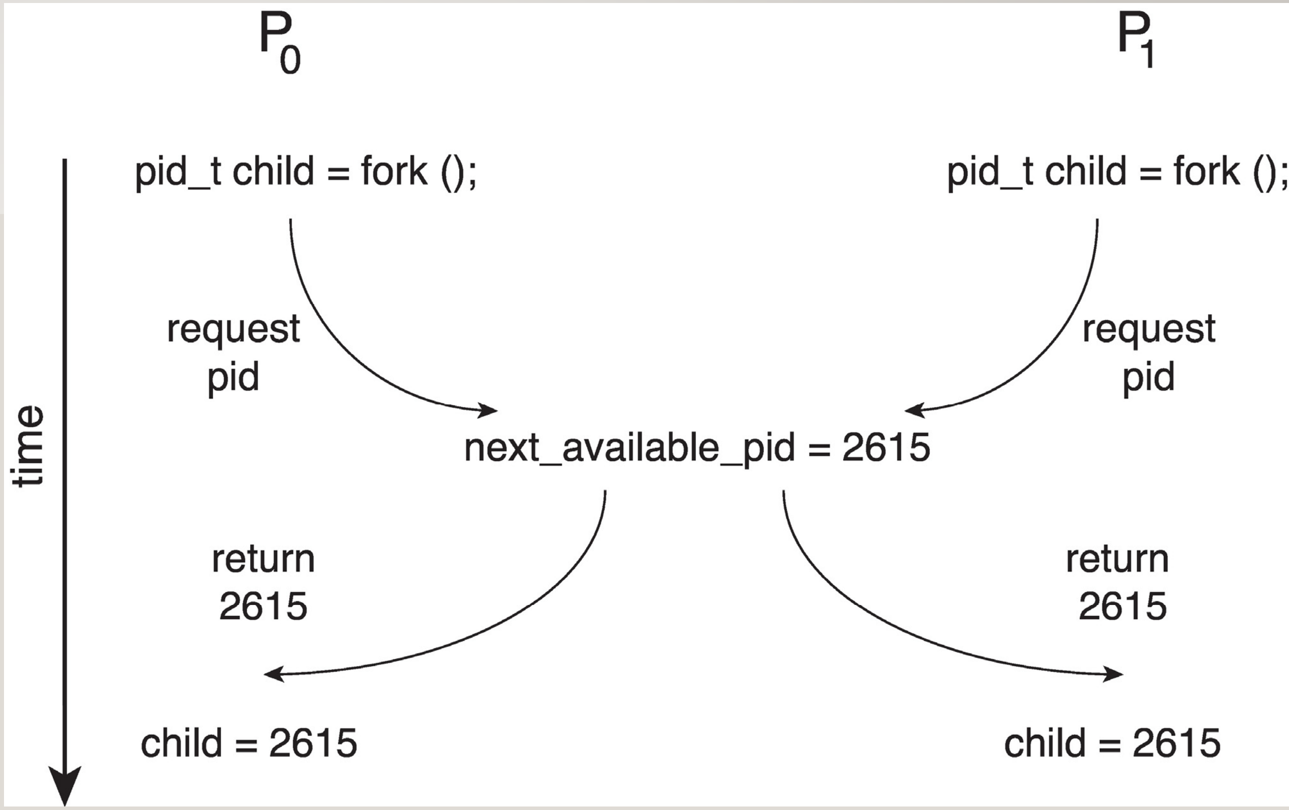
- ➢ Problem to be addressed
- ➢ Race Condition
- ➢ The Critical-Section Problem
- ➢ Interrupt Based Solution
- ➢ Peterson's Solution
- ➢ Hardware Support for Synchronization

Problem to be addressed

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- As illustrated earlier in the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer which lead to race condition.

Race Condition

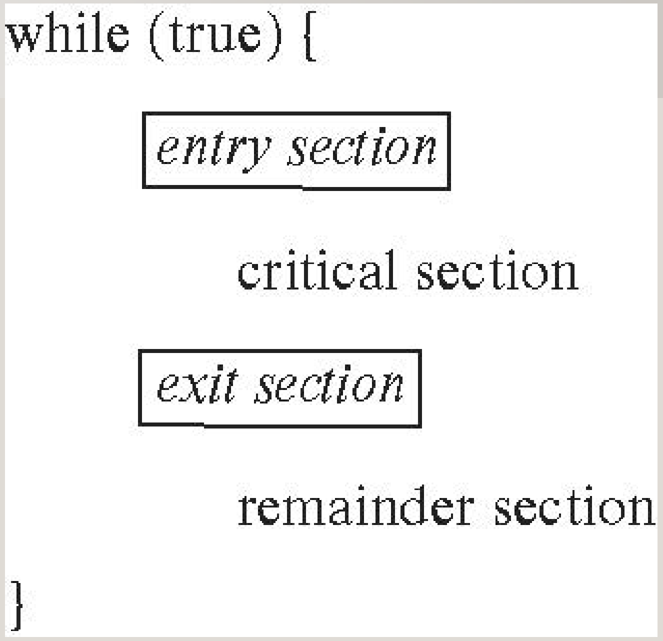
- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)
- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!



Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

General structure of process P_i



```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Critical-Section Problem

Requirements for solution to critical-section problem

- **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting** - A bound must exist 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
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the n processes

Interrupt-based Solution

- ➢ Entry section: disable interrupts
- ➢ Exit section: enable interrupts
- ➢ Will this solve the problem?
  - →What if the critical section is code that runs for an hour?
  - →Can some processes starve – never enter their critical section.
  - →What if there are two CPUs?

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*

Algorithm for Process Pi

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

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?

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:
  - **`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 Pi

```
while (true){

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

        /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

```
P0: flag[0] =  true;
P0_gate: turn = 1;

while (flag[1] == true && turn == 1)
{ // busy wait }

// critical section ...


// end of critical section

flag[0] = false;
```

```
P1: flag[1] =  true;
P1_gate: turn = 0;

while (flag[0] == true && turn == 0)
{ // busy wait }

// critical section ...


// end of critical section

flag[1] = false;
```

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

MUTEX: So if both processes are in their critical sections,
then we conclude that the state must satisfy flag[0] and flag[1] and turn = 0 and turn = 1.
No state can satisfy both turn = 0 and turn = 1, so there can be no state where both processes are in their critical sections.

P0 and P1 could not have *successfully* executed their while statements at about the same time.

Progress and Bound waiting:

A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

A process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition
flag[j] == true and turn == j, this loop is the only one possible.

Since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

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

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

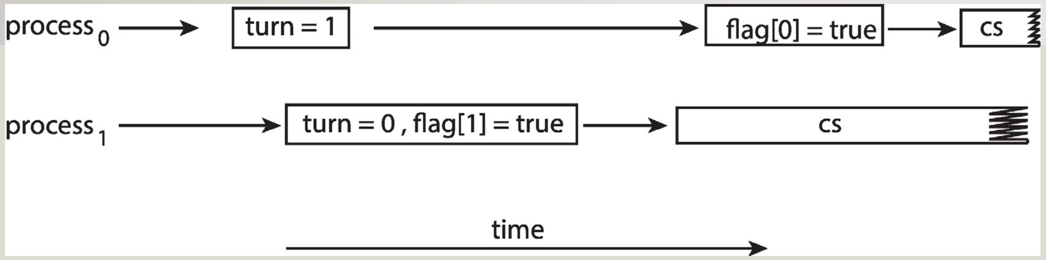
```
flag = true;
x = 100;
```

for Thread 2 may be reordered; If this occurs, the output may be

0!

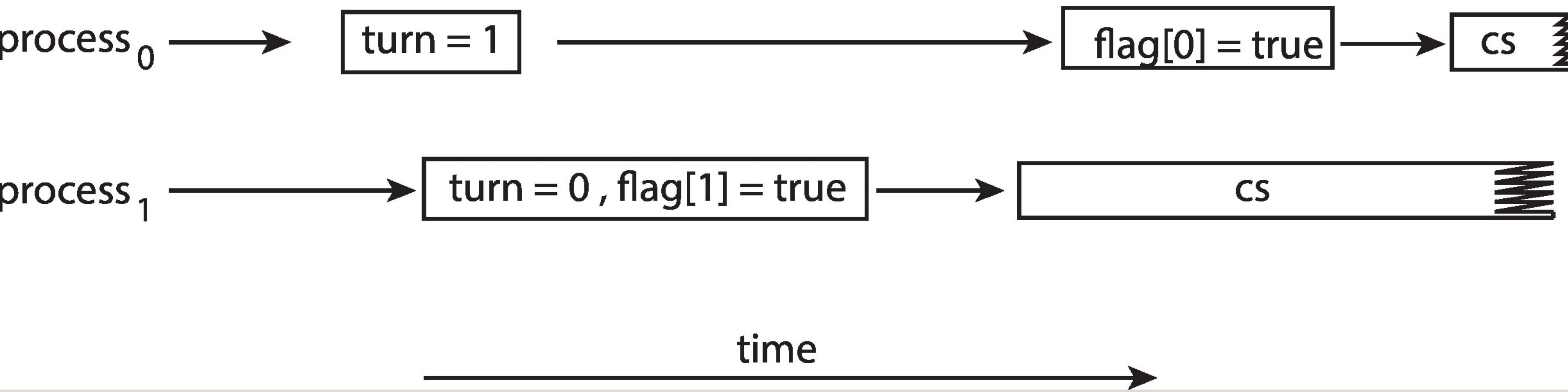
Peterson's Solution Revisited

- 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**.



```
P0_gate: turn = 1;
P0: flag[0] =  true;

while (flag[1] == true && turn == 1)
{ // busy wait }

// critical section ...

// end of critical section
flag[0] = false;
```

```
P1_gate: turn = 0;
P1: flag[1] =  true;

while (flag[0] == true && turn == 0)
{ // busy wait }

// critical section ...

// end of critical section
flag[1] = false;
```

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 two forms of hardware support:
  1. Memory Barriers
  2. 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
  3. Atomic variables

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.
- 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 slide *18*

- 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`.

boolean flag = false;
int x = 0;
where Thread 1 performs the statements
while (!flag)
;
print x;

and Thread 2 performs
x = 100;
flag = true;

The test_and_set Instruction (Hardware 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`

executed atomically -

If two test and set() instructions are executed simultaneously (each on a different core), they will be **executed sequentially in some arbitrary order**.

Thread executing an atomic instruction can’t be preempted or interrupted while it’s doing it.

Atomic operations on same memory value are serialized

Solution Using test_and_set()

- Shared boolean variable `lock`, initialized to **false**
  Solution:

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

          /* critical section */

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

```
boolean rv = *target;

          *target = true;

          return rv:
```

- Does it solve the critical-section problem?

The compare_and_swap Instruction (Hardware 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 */
}
```

```
int temp = *value;
    if (*value == expected)
       *value = new_value;
     return temp;
```

- Does it solve the critical-section problem? –
  - Check bound-waiting Condn.

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

common data structures are
boolean waiting[n];
int lock;

The elements in the waiting array are initialized to false, and
Lock is initialized to 0.

- Does it solve the critical-section problem? –
  - Check bound-waiting Condn.

To prove that the progress requirement is met, we note that the arguments presented for *mutual exclusion* also apply here, since a process exiting the critical section either sets lock to 0 or sets to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the *bounded-waiting* requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n-1, 1, 0, ..., i-1). It designates the first process in this ordering that is in the entry section (waiting[j]== true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n-1 turns.

Also - Acquire, Release;   DPRAM in H/W

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

```
increment(&sequence);
```

→ensures **`sequence`** is incremented without interruption.

- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1));
}
```