Synchronization Tools

Course Code: CSC 2209 Course Title: Operating Systems

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**08**	**Week No:**	**08**	**Semester:**	
Lecturer:	*Name & email*				

Lecture Outline

1. Background
2. The Critical-Section Problem
3. Peterson's Solution

Background

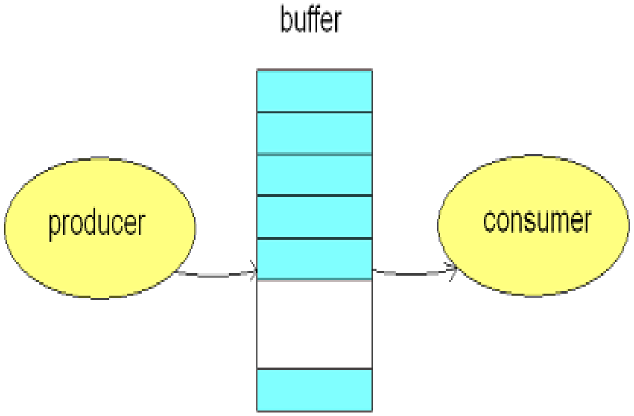
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining **data consistency** requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

Suppose that we wanted to provide a solution to the **consumer-producer problem** that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is **incremented by the producer** after it produces a new buffer and is **decremented by the consumer** after it consumes a buffer.

Producer

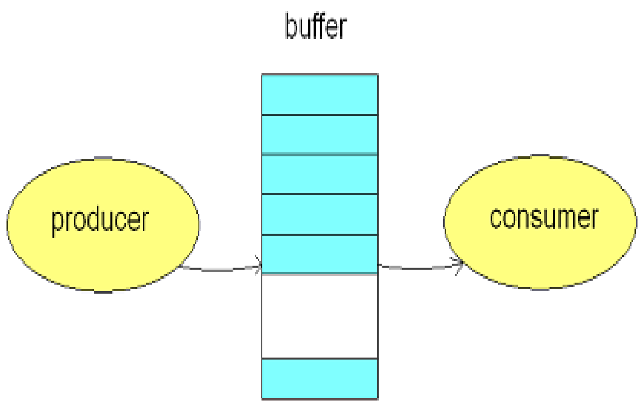
```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```



Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```



Race Condition

- A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

- A **race condition** occurs when two or more threads can access shared data and they try to change it at the same time.

Race Condition

- **counter++** could be implemented as

  register1 = counter
  register1 = register1 + 1
  counter = register1

- **counter--** could be implemented as

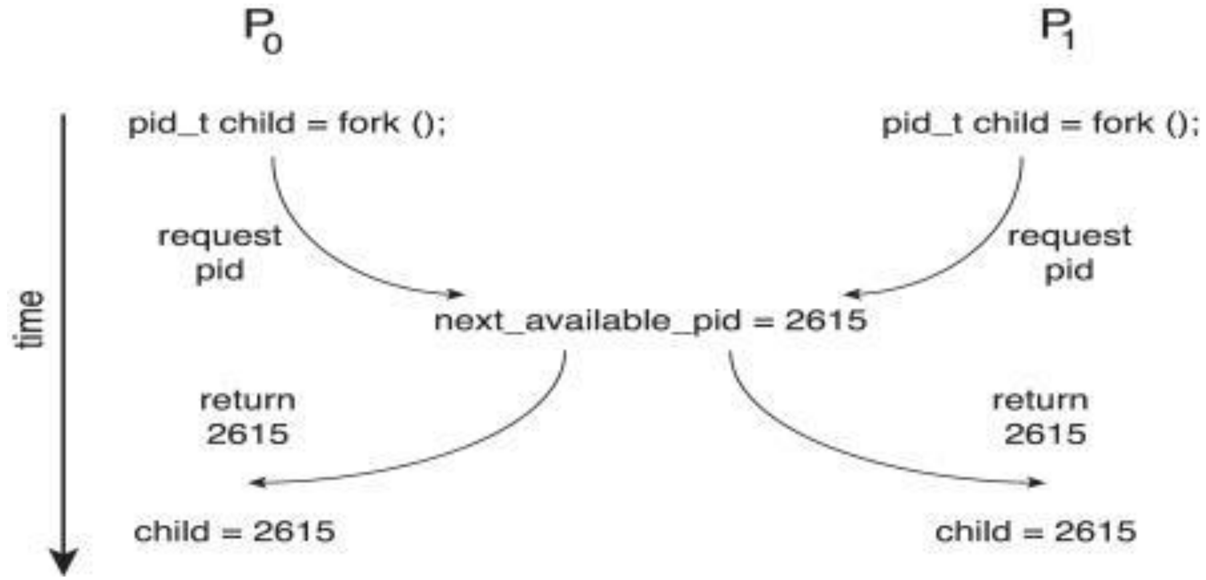
  register2 = counter
  register2 = register2 - 1
  counter = register2

- Consider this execution interleaving with "counter = 5" initially:

  S0: producer execute **register1 = counter** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = counter** {register2 = 5}
  S3: consumer execute **register2 = register2 – 1** {register2 = 4}
  S4: producer execute **counter = register1** {counter = 6 }
  S5: consumer execute **counter = register2** {counter = 4}

Race Condition

- Processes P_0 and P_1 are creating child process using the fork() **system call**
- Race condition on kernel variable next_available_pid which represents the next available **process identifier (pid)**



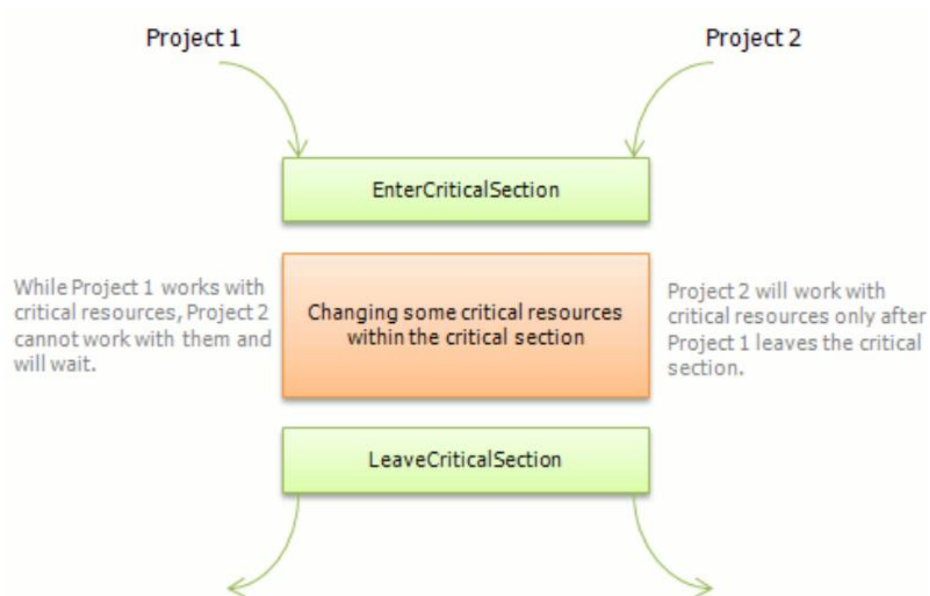
- Unless there is **mutual exclusion**, the same **pid** could be assigned to two different processes!

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 is in its critical section, no other process may be in its critical section
- ***Critical section problem*** is to design protocol to solve this

Critical Section

- 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 $\boldsymbol{P_i}$




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

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **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 processes that will enter the critical section next cannot be postponed indefinitely (stopping each other)

3. **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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- **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 (interested) to enter the critical section. **flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

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

        /* critical section */

        flag[i] = false; // Exit section

        /* remainder section */

}
```

Peterson's Solution (cont'd)

- Provable that the three CS requirement are met:

  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ 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

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

Peterson's Solution

- Two threads share the data:

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

- Thread 1 performs

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

- Thread 2 performs

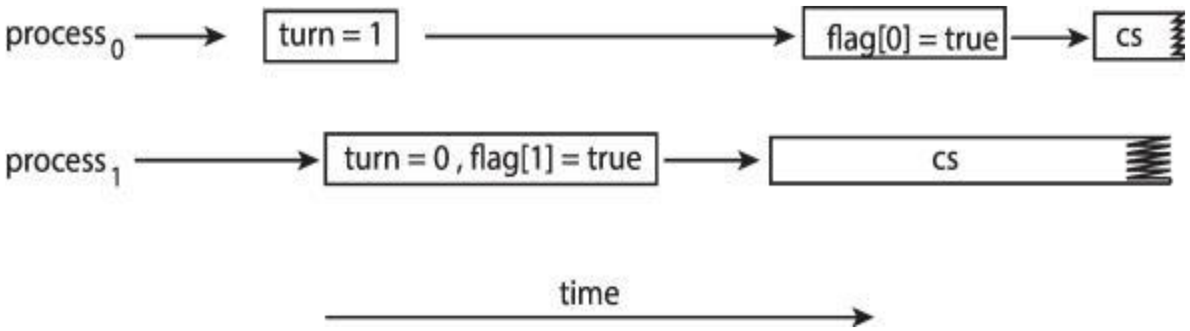
```
x = 100;
flag = true
```

- What is the expected output?

Peterson's Solution

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

  flag = true;
  x = 100;
- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!

Books

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9th

References

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9^{th}