

Chapter 5: Process Synchronization

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

Objectives

- To present the concept of process **synchronization**.
- To introduce the **critical-section problem**, whose solutions can be used to ensure the **consistency of shared data**
- To present both software and hardware solutions of the critical-section problem
- To examine several **classical process-synchronization problems**
- To explore several tools that are used to solve process synchronization problems

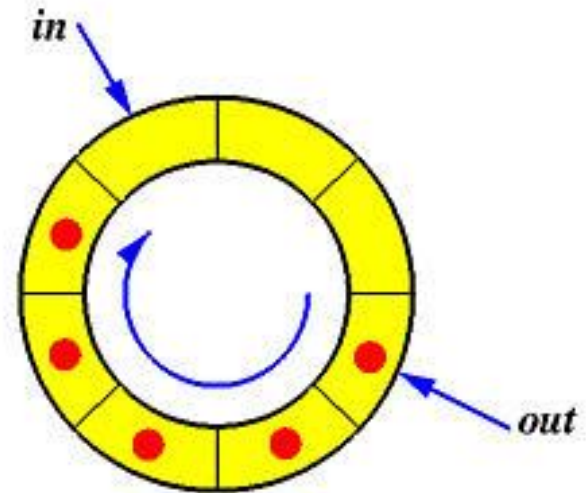
Background

- Processes can execute concurrently
 - In Chap-3: Process scheduler switches among processes; **concurrency**
 - In Chap-4: Multiprogramming distributes tasks among cores; **parallelism**
- May be interrupted at any time, partially completing execution
 - How to preserve the integrity of data shared by several processes..?
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background (Illustration of the Problem)

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers; **the original solution in Chap-3 allowed at most $\text{BUFFER_SIZE} - 1$ items in the buffer at the same time.**

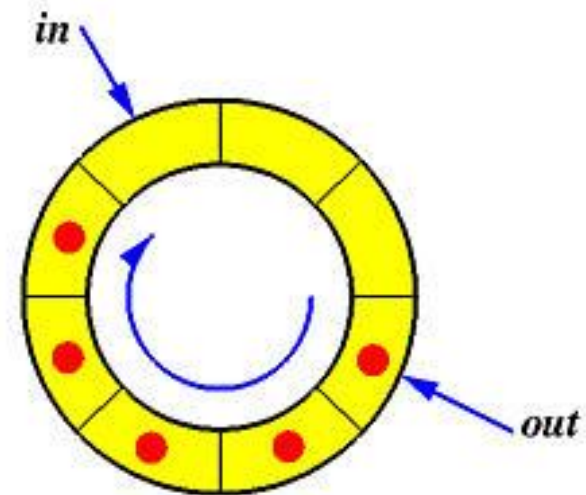
```
item next_consumed;  
while (true) {  
    while (in == out) //Buffer is empty  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consume the item in next consumed */  
}
```



Producer

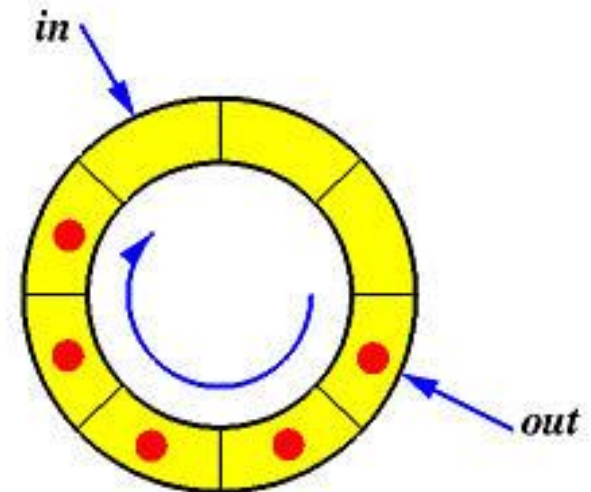
Solution: We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

```
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

- However, Producer and consumer may function incorrectly when executed concurrently

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

```
register1 = counter      //load
register1 = register1 + 1 //Increment
counter = register1      //store
```

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

```
register2 = counter      //load
register2 = register2 - 1 //decrement
counter = register2      //store
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{ counter = 4 }

- **Race condition** = concurrent access to variable + result depends on order
 - **Solution:** synchronization; only one process at a time accesses data

Race Conditions

- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called **race condition**.

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

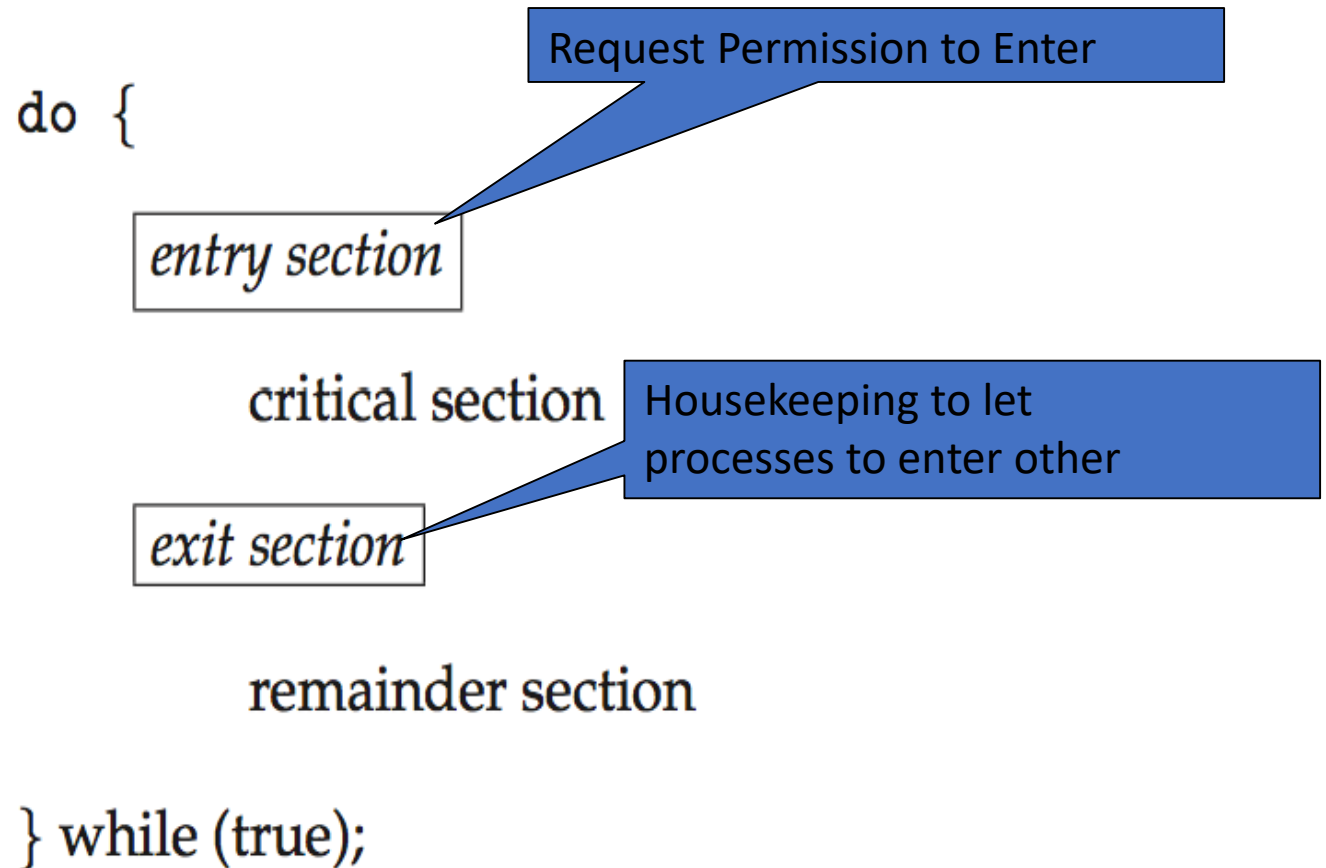
Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has a segment of code called a **critical section**, in which
 - It may be changing common variables, updating table, writing file, ... etc
 - When one process in critical section, **no other** may be in its critical section
 - No two processes are executing in their critical sections at the same time
- **Critical section problem** is to design protocol to solve this
 - That is: protocol that processes can use to cooperate

Critical Section

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

- Gene



Solution to Critical-Section Problem

A solution to the critical section problem must satisfy the following 3 requirements:

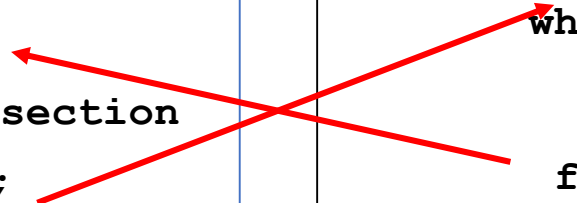
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
 2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
 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
- Speed Assumptions
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes
 - Imagine case where many active kernel processes accessing a list of all open files in the system. This data structure is prone to possible race conditions.

Algorithms for Process P_i

Algorithm 1: Any Problem?.

```
bool flag[i] = false;
//Process intends to enter CS sets the flag
true
do {
    flag[i] = True; //i is ready
    while (flag[j]);
        critical section
    flag[i] = False;
        remainder section
} while (true);
```

```
bool flag[j] = false;
//Process intends to enter CS sets the flag
true
do {
    flag[j] = True; //j is ready
    while (flag[i]);
        critical section
    flag[j] = False;
        remainder section
} while (true);
```



Algorithms for Process P_i

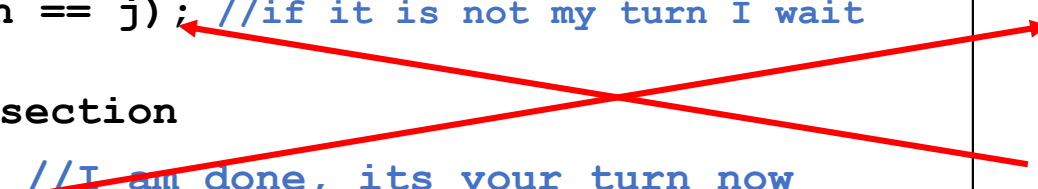
Algorithm 2:Any Problem?.

Turn = i

```
do {  
  
while (turn == j); //if it is not my turn I wait  
  
critical section  
turn = j; //I am done, its your turn now  
  
remainder section  
} while (true);
```

Turn = j

```
do {  
  
while (turn == i);  
  
critical section  
turn = i;  
  
remainder section  
} while (true);
```



Algorithms for Process P_i

Algorithm 1: Does not satisfy 'Progress'.

```
bool flag[2] = false;

do {
    flag[i] = True;
    while (flag[j]); //both flag maybe true
        critical section
    flag[i] = False;
    remainder section
} while (true);
```

Algorithm 2: Does not satisfy 'Progress'. Also, an irrelevant process blocks other processes from entering a critical section.

```
Turn = i or J

do {

    while (turn == j);

        critical section
    turn = j; //Gives turn to 'j' who is not
    interested

    remainder section
} while (true);
```


Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- **Peterson's Solution**
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assumes that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables (combines the previous two algorithms):
 - **int turn;**
 - **Boolean flag[2] //both set to FALSE initially**
- 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!

Peterson's Solution for Process P_i

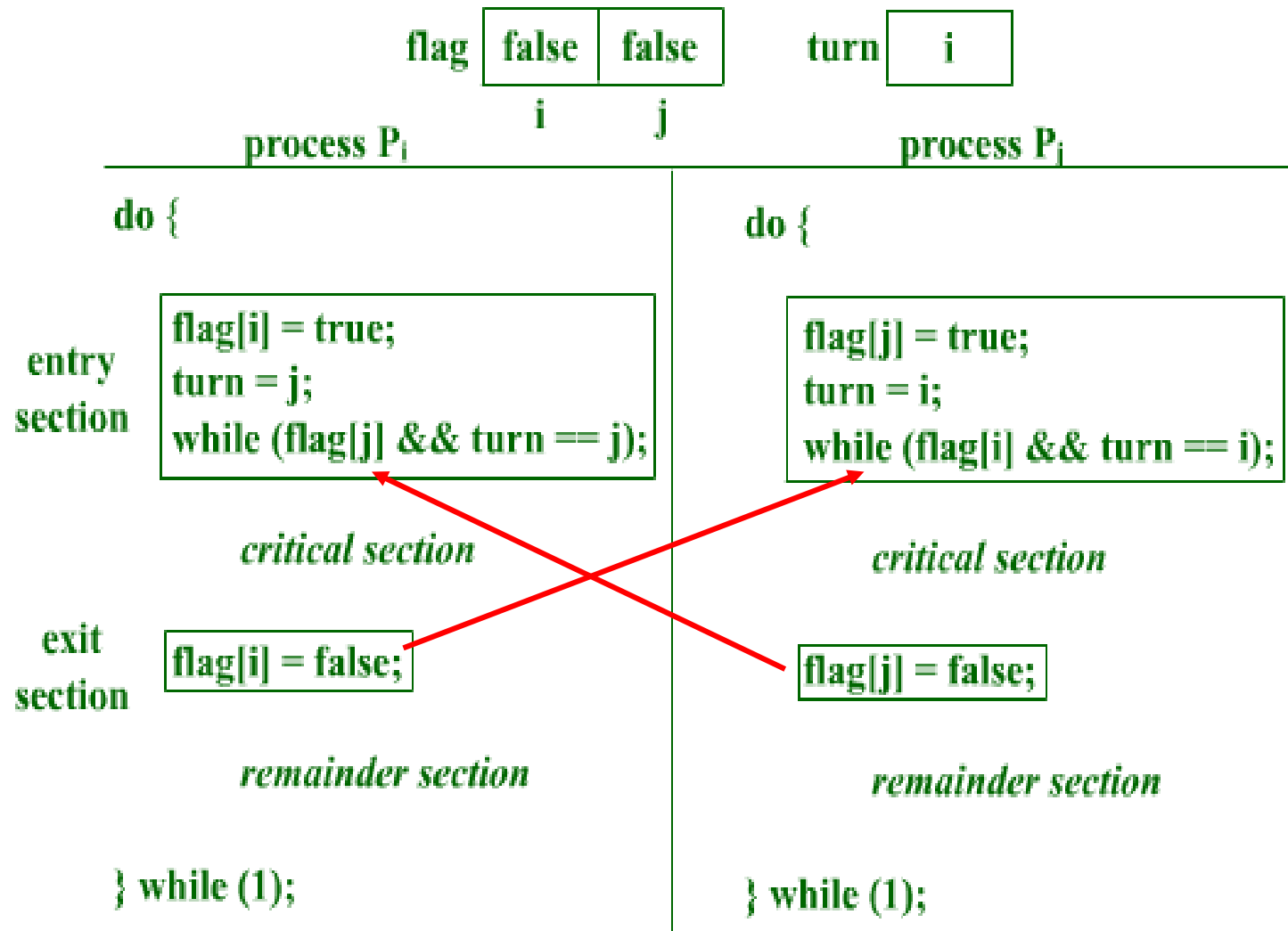
Flag[] set to false initially.

```
do {  
    flag[i] = true; //Process i is ready  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    {  
        flag[i] = false;  
        remainder section  
    }  
} while (true);
```

Entry Section

Exit Section

Peterson's Solution for Process P_i and P_j



Peterson's Solution (Cont.)

- 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 (Cont'd)

- **Software solution** to the critical section problem
 - **Restricted to two processes**
- Good algorithmic description
 - Can show how to address the 3 requirements
- No guarantees on modern architectures
 - Instruction reordering

End of Chapter 5 (part 1)