Atomic Operation

The operations that get completed in a single CPU clock cycle

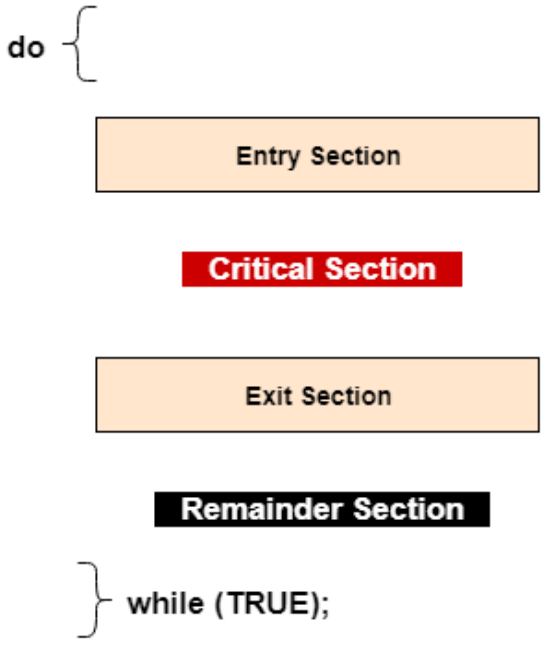
An atomic operation is a unit of code that is executed without interference from other threads or processes

Bounded buffer

A bounded buffer **lets multiple producers and multiple consumers share a single buffer**. Producers write data to the buffer and consumers read data from the buffer. Producers must block if the buffer is full. Consumers must block if the buffer is empty.

Critical section

a critical section is a piece of code that accesses shared resources and is protected by a mechanism (such as a lock) to ensure that only one thread or process can execute the code at a given time. This is used to prevent race conditions, where two or more threads try to access the same resource simultaneously, resulting in unpredictable behaviour. The critical section is typically surrounded by code that acquires and releases the lock, ensuring that other threads are not able to access the shared resource until the critical section has completed



Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion**

  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

- **Progress**

  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

- **Bounded Waiting**

  Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

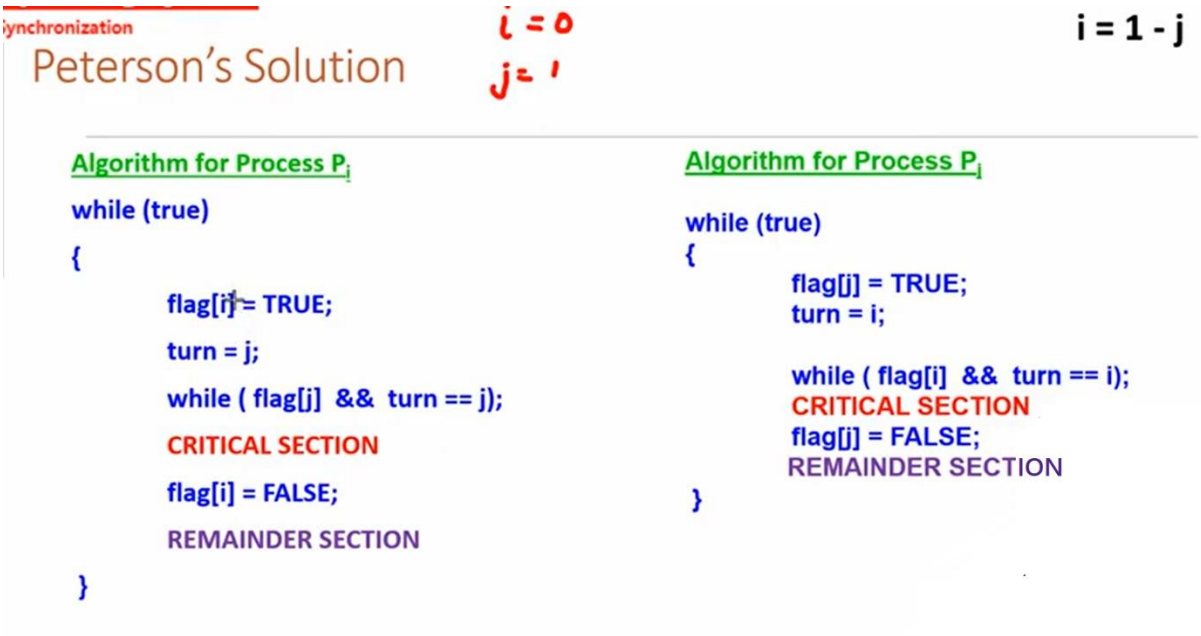
Software solution -> Peterson, bakery

Hardware solution -> TestAndSet, Swap

OS solution -> Semaphores

Peterson

It's a two process solution



Initially the flags are false. When a process wants to execute its critical section, it sets its flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished its own critical section.
After this the current process enters it's critical section. After completing the critical section, it sets its own flag to false, indication it does not wish to execute anymore.

Bakery's Algo

N process solution

The algorithm preserves the first come first serve property.

- Before entering its critical section, the process receives a number. Holder of the smallest number enters the critical section.
- If processes Pi and Pj receive the same number,
- `if i < j`
- `Pi is served first;`
- `else`
  `Pj is served first.`

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 3, 3, 3, 4, 5, …

Firstly the ticket number is compared. If same then the process ID is compared next

```
repeat
   choosing[i] := true;
   number[i] := max(number[0], number[1], ..., number[n - 1])+1;
   choosing[i] := false;
   for j := 0 to n - 1
      do begin
         while choosing[j] do no-op;
         while number[j] != 0
            and (number[j], j) < (number[i], i) do no-op;
      end;

      critical section

   number[i] := 0;

      remainder section

until false;
```

Explanation –
Firstly the process sets its "choosing" variable to be TRUE indicating its intent to enter critical section. Then it gets assigned the highest ticket number corresponding to other processes. Then the "choosing" variable is set to FALSE indicating that it now has a new ticket number. This is in-fact the most important and confusing part of the algorithm.

It is actually a small critical section in itself ! The very purpose of the first three lines is that if a process is modifying its TICKET value then at that time some other process should not be allowed to check its old ticket value which is now obsolete. This is why inside the for loop before checking ticket value we first make sure that all other processes have the "choosing" variable as FALSE.

After that we proceed to check the ticket values of processes where process with least ticket number/process id gets inside the critical section. The exit section just resets the ticket value to zero.

Semaphore

Semaphore

Synchronization tool that does not require **busy waiting**

Semaphore *S* is an integer variable

Two standard operations modify S: wait() and signal()

- Originally called P() and V()

Less complicated

Can only be accessed via two indivisible (atomic) operations

```
wait (S)
{
    while S <= 0;  // no-op
      S--;
}
```

```
signal (S)
{
  S++;
}
```

Busy waiting -> when using the loop (while loop) to wait

Spinlock -> Semaphores which involves Busy waiting is.

With no busy waiting

With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:

- value (of type integer)
- pointer to next record in the list

Two operations:

- block – place the process invoking the operation on the appropriate waiting queue
- wakeup – remove one of processes in the waiting queue and place it in the ready queue

Implementation of wait:

```
wait (S){
  value--;
  if (value < 0) {
            //add this process to waiting queue
             block();  }
}
```

Implementation of signal:

```
Signal (S){
    value++;
     if (value <= 0) {
              //remove a process P from the waiting que
               wakeup(P);  }
}
```

Dining philosopher

Dining-Philosophers Problem (Cont.)

The structure of Philosopher *i*:

```
While (true)  {
      wait ( chopstick[i] );
      wait ( chopStick[ (i + 1) % 5] );

          // eat

      signal ( chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

          // think
}
```