Critical Section Problem

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

- 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 count that keeps track of the number of full buffers. Initially, count 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 and put in nextProduced  */
        while (count == BUFFER_SIZE); // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

Consumer

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

Race Condition

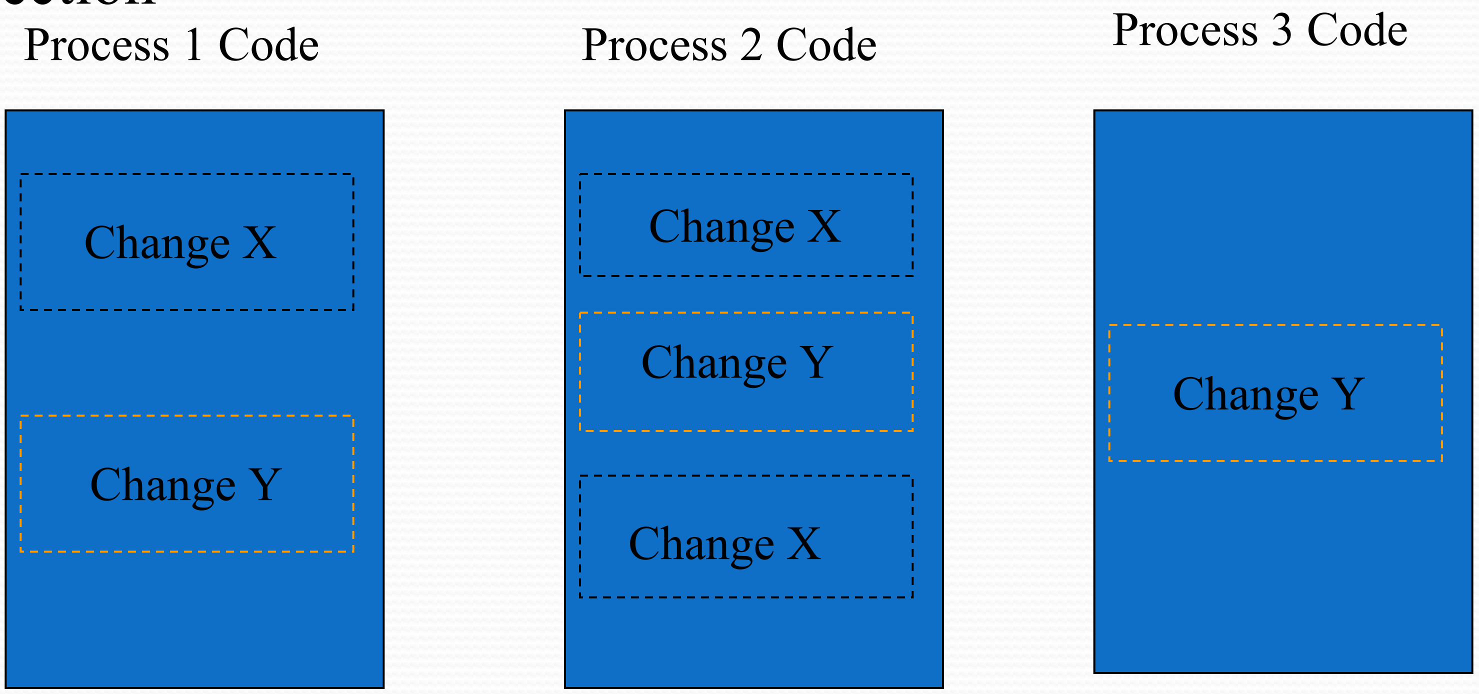
- **count++** could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- **count--** could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute **register1 = count** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = count** {register2 = 5}
  S3: consumer execute **register2 = register2 - 1** {register2 = 4}
  S4: producer execute **count = register1** {count = 6 }
  S5: consumer execute **count = register2** {count = 4}

Race Condition

- 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 a **race condition.**

Programs and critical sections

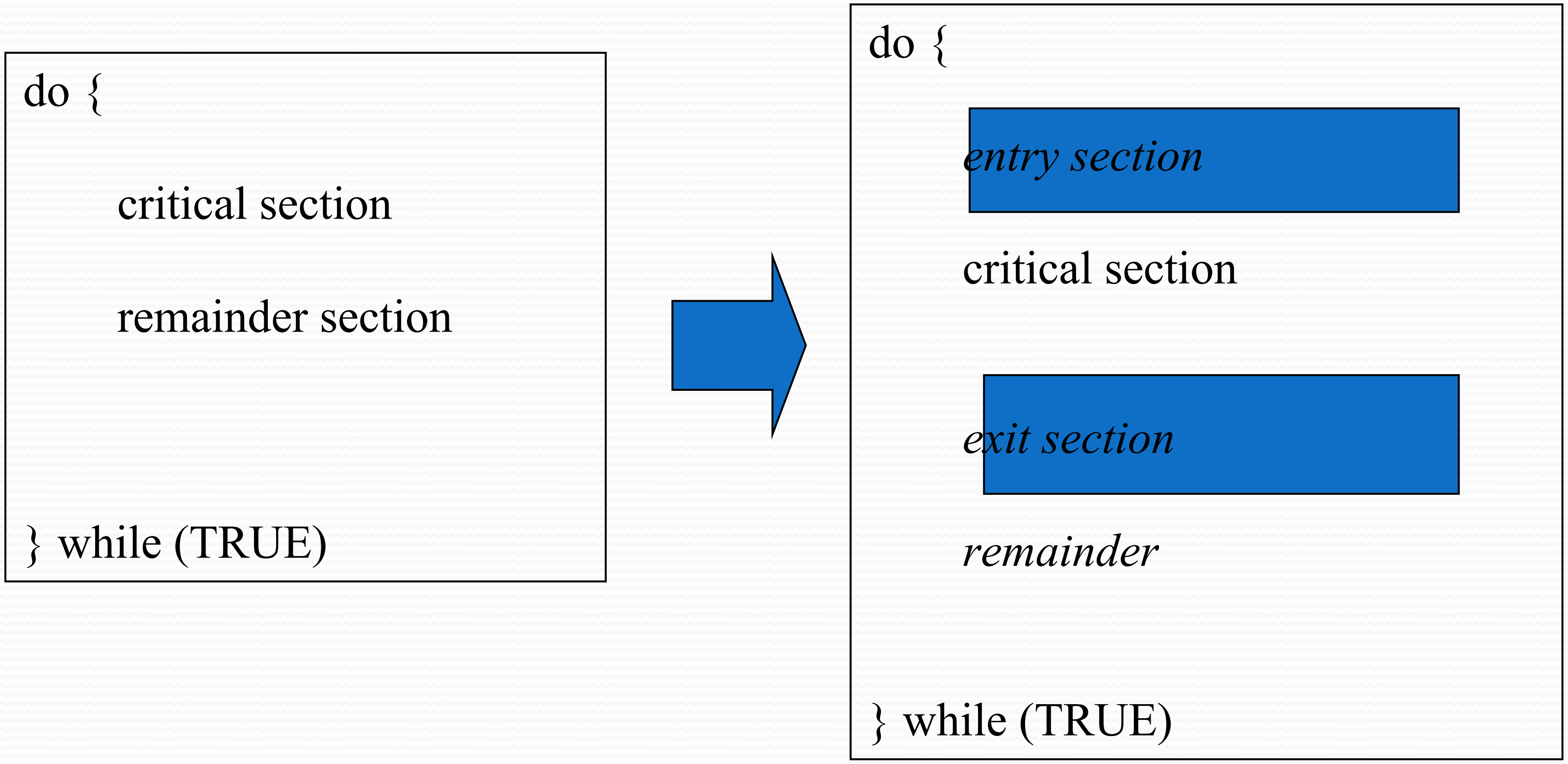
- The part of the program (process) that is accessing and changing shared data is called its critical section



Assuming X and Y are shared data.

Critical Section

- The general way to do that is:



```
do {

        critical section

        remainder section


} while (TRUE)
```

```
do {

        entry section

        critical section

        exit section

        remainder

} while (TRUE)
```

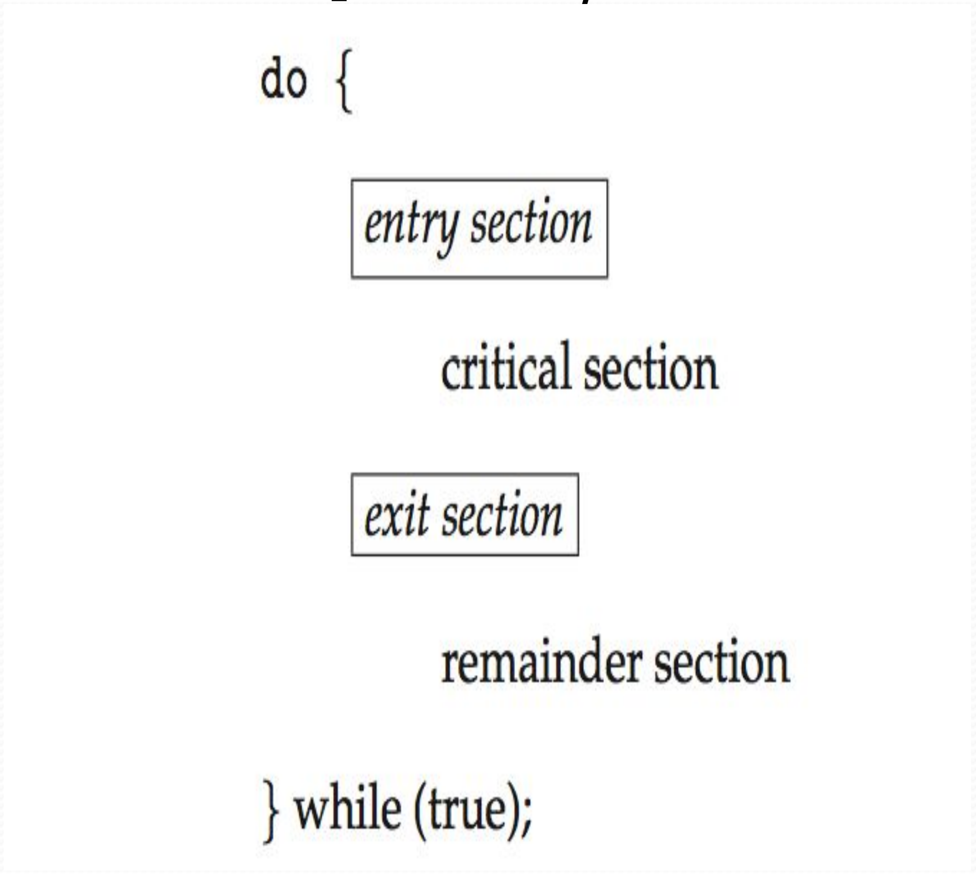
Entry section will allow only one process to enter and execute critical section code.

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**

Critical Section

- General structure of process ***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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection 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
   - 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 non peemptive kernel mode

Critical Section-Two process Solution

- Let P_0 and P_1 be two processes .For convenience let it be P_i and P_j that is j==1-i.
- Let the processes share a common integer variable turn initializes to 0(or 1).
- If turn==i ,then process P_i is allowed to execute in its critical section.
- Only one process at a time can be in its critical region.

Disadvantage

- This algorithm requires strict alternation of processes in the execution of the critical section.

  If turn==0 and P_1 is ready to enter its critical section,P_1 cannot do so ,even though P_0 may be in its remainder section.

Algorithm for Process P_i

```
do {

  while (turn != i);
        critical section
  turn = j;

              remainder section

    } while (true);
```