Unit 3:Inter Process Communication

Prepared By :

Ritika Chokshi

Assistant Professor

CSPIT-CE
CHARUSAT University,
Changa

References : Operating System Concepts By Galvin

Modern Operating System By Tanenbaum

Contents of the unit

- Race Conditions, Critical Section, Co-operating Thread/ Mutual Exclusion
- Hardware Solution, Strict Alternation, Peterson's Solution
- The Producer Consumer Problem, Semaphores, Event Counters, Monitors
- Message Passing and Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem.

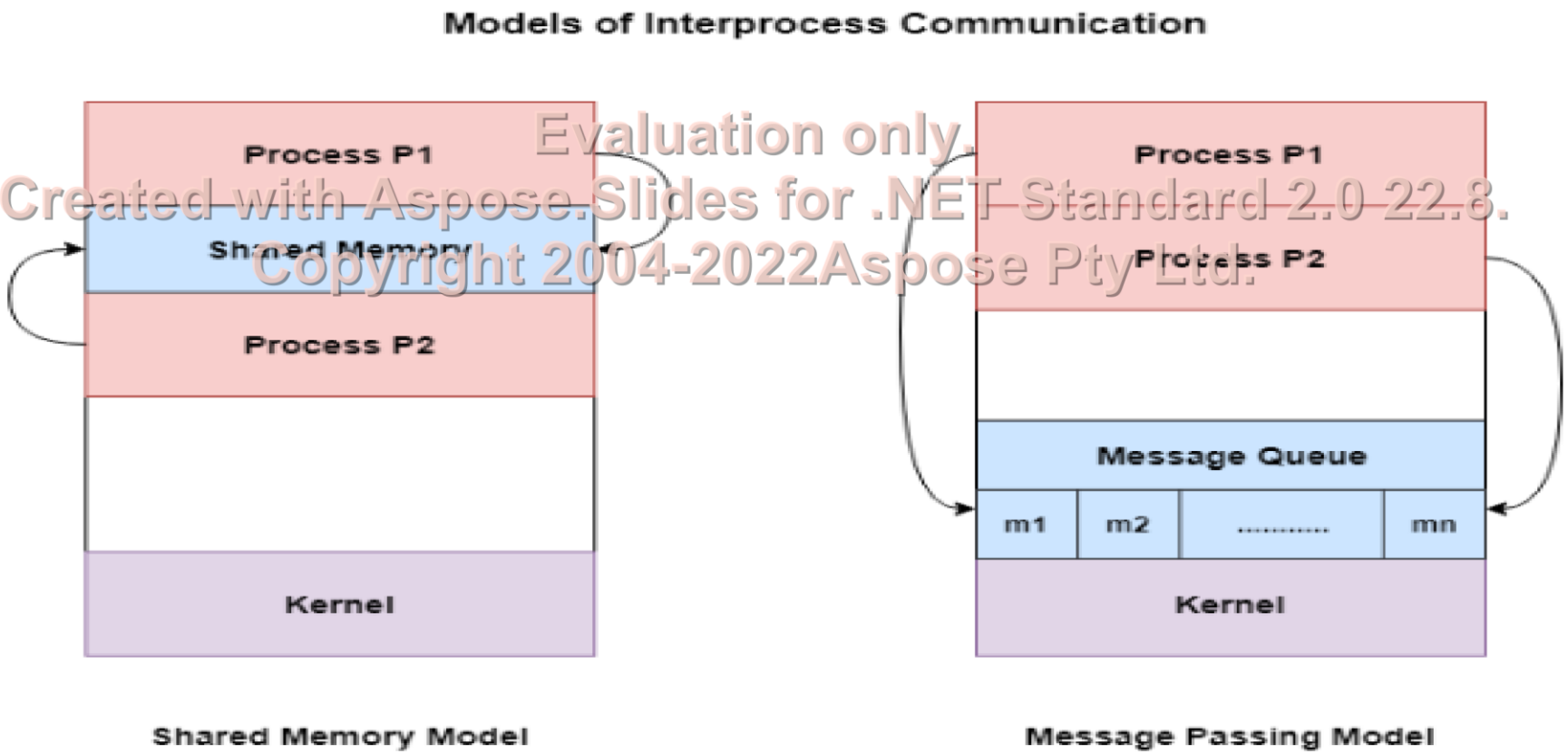
Inter Process Communication

- Types of Process: Independent & Cooperating Process

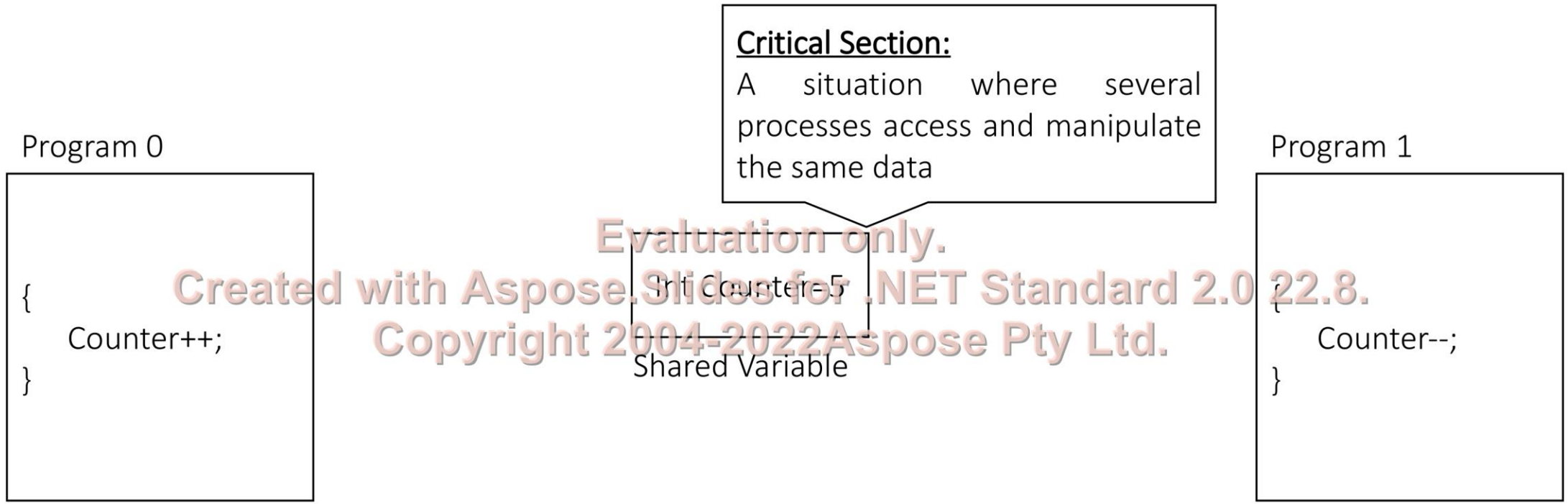
IPC

- What is IPC?
- Why IPC?/ Advantage of IPC/the reasons to provide an environment that allows process cooperation :
  - Information sharing
  - Modularity
  - Computational Speed
  - Convenience
- How IPC can be achieved?/Two Ways for providing an IPC :
  - Shared Memory
  - Message Passing

Models for IPC:



Scenario



1. What will be the output if these two program executes?
2. Possible output

Race Condition

- The outcome depends on the order in which the access take place.

- Prevent Race Condition:
  - By Synchronization
  - Ensure only once process at a time manipulates the critical data

- No more than one process should execute in critical section at a time.

- Race Condition in Multicore.

Race Conditions

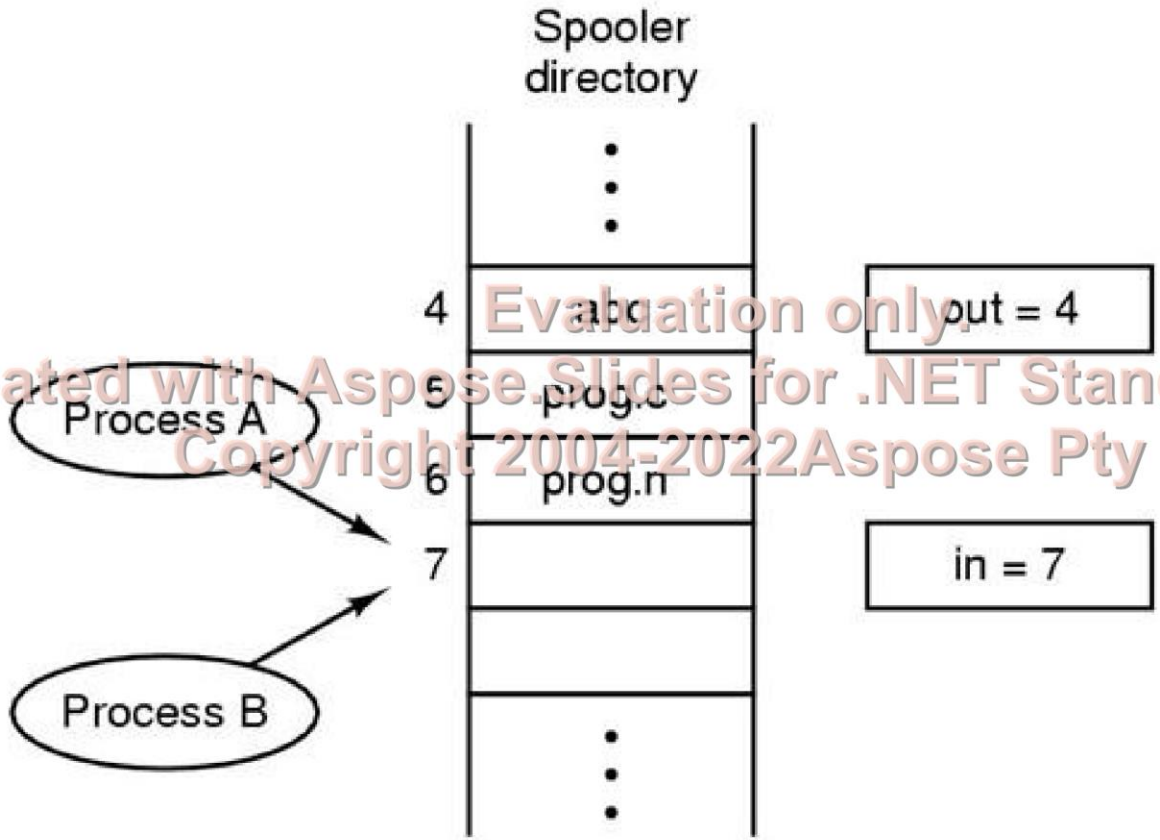
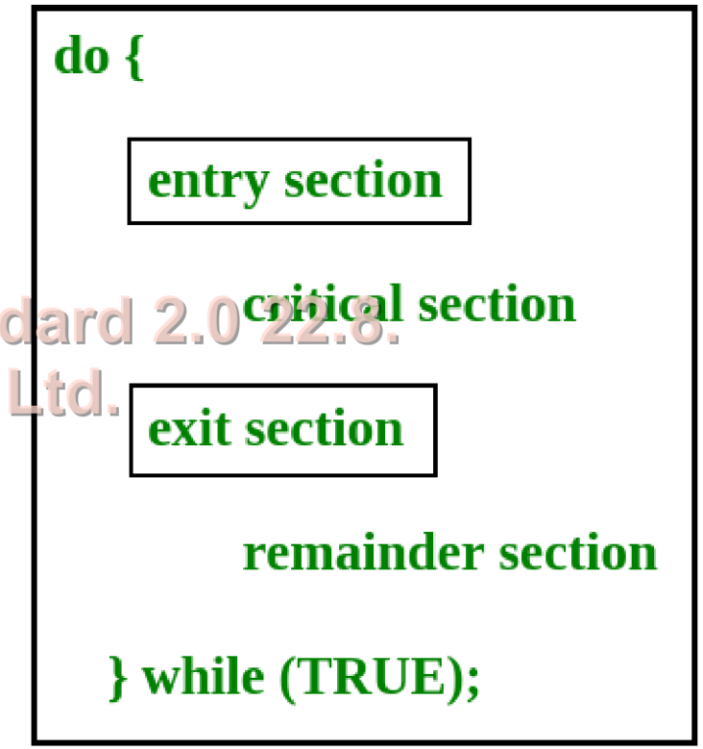


Figure 2-21. Two processes want to access shared memory at the same time.

Critical Regions (1)

Conditions required to avoid race condition:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.



Critical Regions (2)

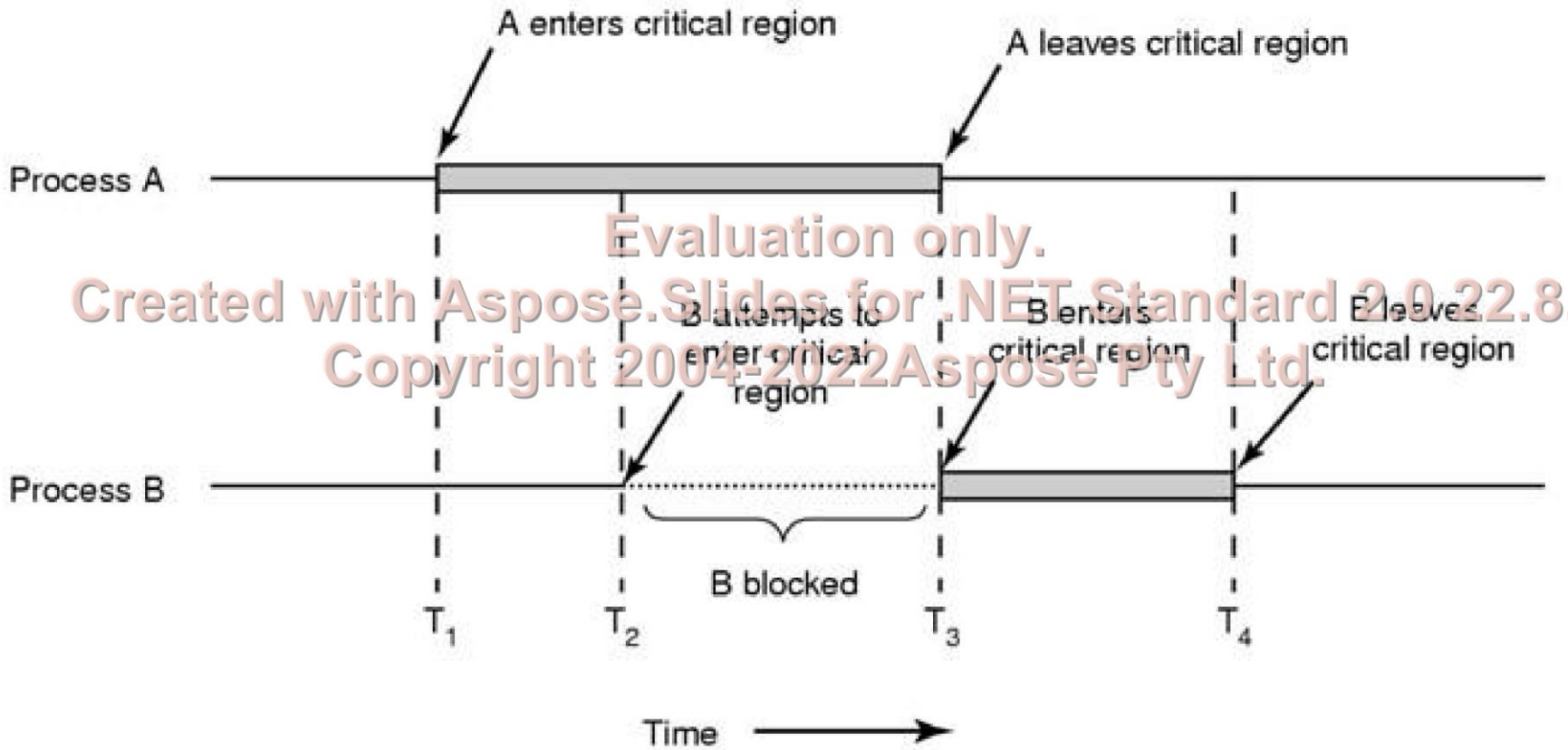


Figure 2-22. Mutual exclusion using critical regions.

Solution for Critical Section Problem

Solution should satisfy the following requirements:

1. Mutual Exclusion:
   - No two process may be simultaneously inside their critical region.

2. Progress:
   - No process running outside its critical region may block other processes.

3. No starvation(Bounded wait):
   - No process should have to wait forever to enter its critical region.

Proposals for achieving Mutual Exclusion

- ✓ Disabling Interrupts
- ✓ Lock variables
- ✓ Strict Alteration
- ✓ Peterson’s solution
- ✓ TSL instruction

1. Disabling Interrupts

- When Interrupts are disabled, context switch won't happen.
- Requires privileges:
  - User process generally cannot disable interrupts.
- Disadvantage:
  - Not suited for multicore systems
  - Process has power to disable the interrupt. (If process does not turned it on?)

Process 1

```
{
Disable Interrupts()
 Critical section
enable Interrupts()
}
```

Process 2

```
{
Disable Interrupts()
 Critical section
enable Interrupts()
}
```

2. Lock Variables

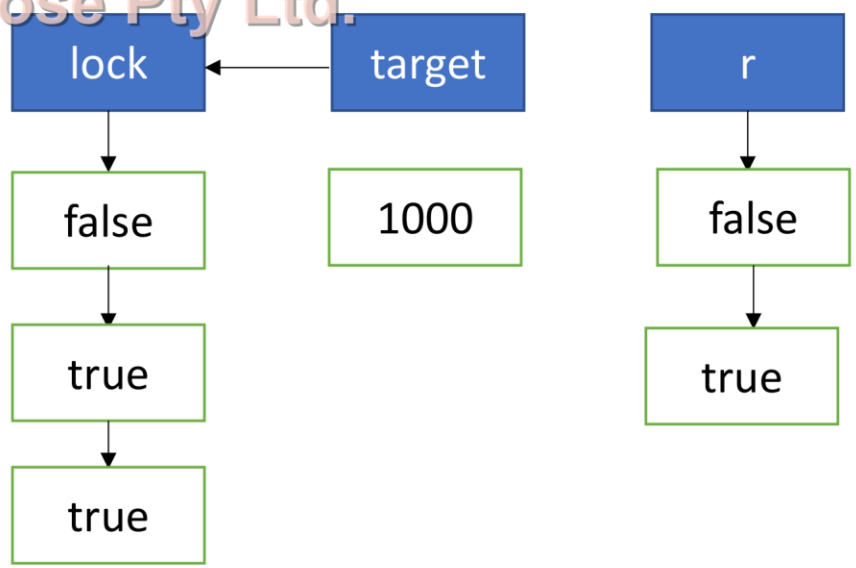
- Shared Lock variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region.

- 0 means, no process is in its critical region.
- 1 means, some process is in its critical region.

- **Violate the Mutual Exclusion.**
- Suppose process1 reads the lock as 0 and tries to set as 1.Process 2 sets the lock to 1 mean time. (2 processes end up in critical section.)

Hardware Locks: TSL(Test and Set Lock)

```
While(test_and_set (&lock));
    Critical section
lock=false;
Boolean test_and_set (Boolean *target)
{
        Boolean r=*target;
        *target=TRUE;
        return r;
}
```

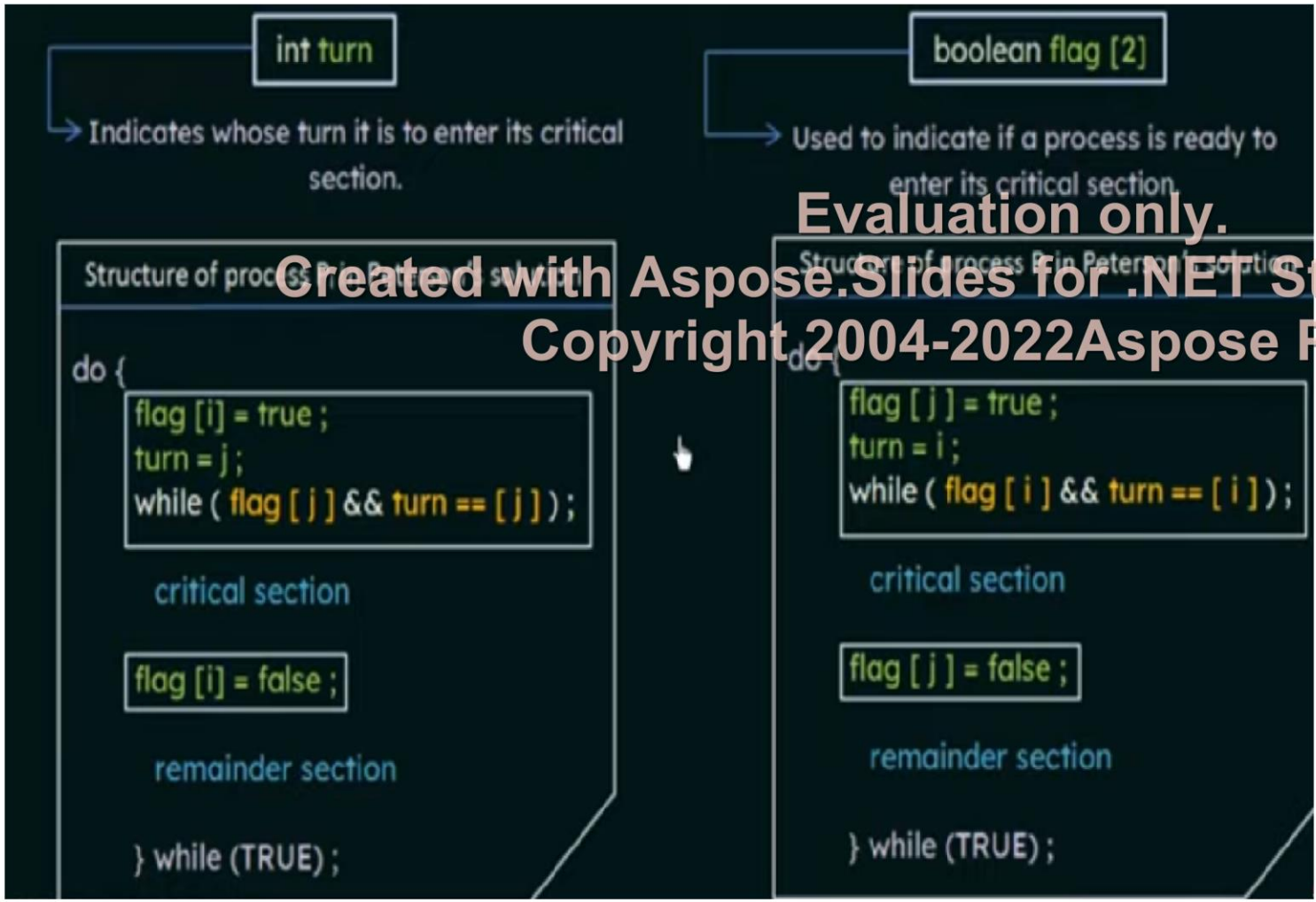
3. Strict Alternation

```
Process 0:
While(True){
While(turn!=0);   /*Loop*/
Critical_region();
turn=1;
Non-critical_region;
}
```

```
Process 1:
While(True){
While(turn!=1); /*Loop*/
Critical_region();
turn=0;
Non-critical_region;
}
```


- Software mechanism
- Implemented in user mode
- Restricted only for two processes
- Initially turn=0.
- Four cases
- **Violates Progress**
- Busy Waiting: Continuously testing a variable until some value appears.
  - Waste CPU cycle
  - Spin lock : Lock that uses busy waiting.

4. Peterson's Solution



- Software based solution
- Restricted to two processes
- Shared Variable : turn, flag

Intel x86 CPUs use XCHG instruction

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

Busy waiting

- **Busy waiting,** also known as spinning, or busy looping is a process synchronization technique **in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution**.
- In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system.

Busy waiting

- There are two general approaches to waiting in operating systems:
- firstly, a process/task can continuously check for the condition to be satisfied while consuming the processor – busy waiting.
- Secondly, a process can wait without consuming the processor. In such a case, the process/task is alerted or awakened when the condition is satisfied. This is known as sleeping, blocked waiting, or sleep waiting.

Sleep and Wakeup

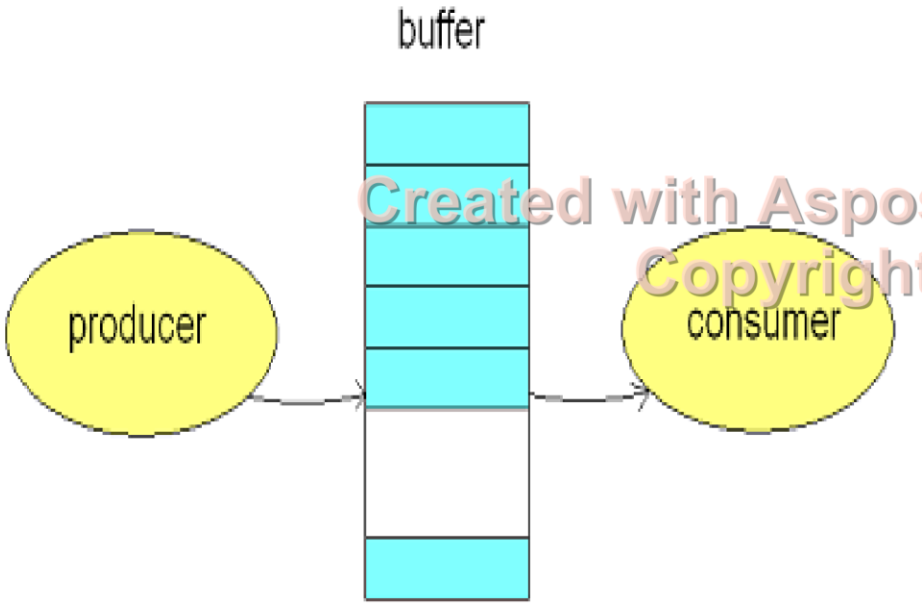
Why Sleep and Wakeup?

- Solution of Busy waiting(Waste CPU Cycle).

- Priority Inversion Problem:
  - H= High Priority process, L=Low Priority process.
  - H will run whenever it is in ready state.
  - L in its critical region, H becomes ready to run. H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, So H Loops forever.

- Sleep is system call which blocks the caller, that is, be suspended until another process wakes it up.

Producer-Consumer Problem(Bounded Buffer Problem)



- We have a buffer of fixed size.
- A producer can produce an item and can place in the buffer.
- A consumer can pick items and can consume them.
- We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item.
- In this problem, buffer is the critical section.

Producer-Consumer Problem (Bounded Buffer Problem)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Context Switch

Missing Wakeup call:

- Buffer is empty.
- Consumer checks value of count which is 0.
- Before sleep the producer generate item and increment count by 1 and execute the wakeup system call.
- Now consumer will execute sleep system and mean while producer fill up the buffer and go to sleep.
- Both will sleep forever.

Semaphore

- Semaphore is a variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment.
- The two most common kinds of semaphores
  - Counting semaphore can take non-negative integer values
  - Binary semaphore can take the value 0 & 1 only.

Semaphore

- 2 operations on Semaphores:
  - **P operation** is also called wait, sleep or down operation
  - **V operation** is also called signal, wake-up or up operation.
  - Both operations are atomic and semaphore(s) is always initialized to one.
- A critical section is surrounded by both operations to implement process synchronization.

- Uses of Semaphores:
  - Achieve Mutual Exclusion
  - Deciding order of execution
  - Managing Resource

Process P

```
// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

Semaphore Implementation

```
Void down(int *s)
        {
              while(*s<=0);
              *s--;
        }
```

```
Void up(int *s)
        {
              *s++;
         }
```


- Could now have **busy waiting** in critical section implementation

Semaphore Implementation 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

Semaphore Implementation with no Busy waiting

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to
S->list;
        block(); //Sleep
    }
}
```

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P
from S->list;
        wakeup(P);
//wakeup process
    }
}
```

Bounded buffer problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value *n*

Producer Consumer using Semaphore

```
#define N 100
typedef int semaphore;
semaphore mutex =    1;
semaphore empty =    N;
semaphore full =     0;
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
   /* controls access to critical region */
   /* counts empty buffer slots */
   /* counts full buffer slots */
```

```
Void producer (void)
{           int item;
            While(TRUE){
                        item = produce_item();
                        down(&empty);
                        down(&mutex);
                        insert_item(item);
                        up(&mutex);
                        up(&full);
            }
}
```

```
Void consumer (void)
{           int item;
            While(TRUE){
                        down(&full);
                        down(&mutex);
                        item = remove_item();
                        up(&mutex);
                        up(&empty);
                        consume_item(item);
            }
}
```

Structure of producer consumer problem

Producer

```
do {
...
  /* produce an item in
  next_produced */
   ...
 wait(empty);
 wait(mutex);
 ...
  /* add next produced to the
  buffer */
            ...
        signal(mutex);
        signal(full);
     } while (true);
```

Consumer

```
do {
            wait(full);
            wait(mutex);
               ...
  /* remove an item from buffer
  to next_consumed */
               ...
            signal(mutex);
            signal(empty);
               ...
   /* consume the item in next
consumed */
               ...
        } while (true);
```

Gate Example

- A shared variable x, initialized to zero, is operated by four process w, x, y, z. Process w and x increment x by one, while process y, z decrement x by two. Each process before reading perform ‘wait’ on semaphore ‘s’ and signal on ‘s’ after store. If semaphore ‘s’ initialized to two, find what is the maximum possible value of x if all process complete execution?

- Answer: 2

Gate Example

- A counting semaphore was initialized to 10. Then 6 P (wait) operations and 4V (signal) operations were completed on this semaphore. The resulting value of the semaphore is
- (a) 0 (b) 8 (c) 10 (d) 12

Ans: option(b)

Gate Example

- The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0=1, S1=0, S2=0.

Process P0	Process P1	Process P2

while (true)
{
wait (S0);
print (0);
release (S1);
release (S2);
}
	wait (S1);
	Release (S0);
		wait (S2);
		release (S0);

How many times will process P0 print '0'?

(a)At least twice

(b)Exactly twice

(c) Exactly thrice

(d) Exactly once

Ans: option (a)

Problems with semaphores

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

- Starvation – indefinite blocking