



## Operating Systems ICS 431

# Ch. 6 Process Synchronization

**Dr. Tarek Helmy El-Basuny**

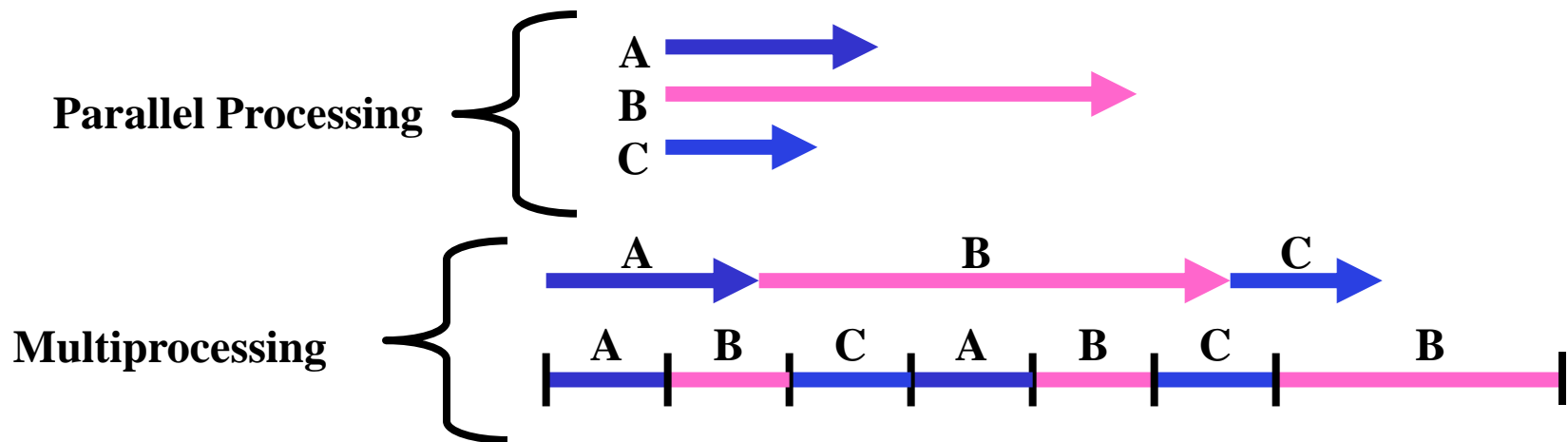
## Ch. 6 Process Synchronization

### This chapter discuss:

- Concurrent Processing.
- Why Process Synchronization?
- Producer-Consumer Problem
- How do processes work with resources that must be shared between them?
- Atomic Operation
- The Critical-Section
- Critical section Implementation
- Evaluating synchronization algorithms of a critical section
- Different algorithms to Synchronize two Processes enter of Critical Section.
- Dangers of handling the Critical Section Problem
- Synchronization Tools
- Semaphores
- Incorrect usages of Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization in different OSs

# Concurrent Processing

- In a single-processor with multiprocessing system, processes are interleaved in time to yield the appearance of simultaneous execution. Even though actual parallelism is not achieved and there is overhead in switching between processes, interleaved executions provide major benefits in processing efficiency and in program structures.
- In a multi-processor systems, it is possible not only to interleave the execution of processes but also to overlap them. Although it might seem that interleaving and overlapping present different problems, both techniques can be viewed as examples of concurrent processing, and both present the same problems.



## Background- Concurrency

- In order to cooperate, processes must be able to:
  - Communicate with one another
    - Passing information between two or more processes
  - Synchronize their actions
    - Coordinating access to shared resources
      - Hardware (e.g., printers, drives), Software (e.g., shared code)
      - Files (e.g., data), Variables (e.g., shared memory locations)
- Concurrent access to shared data may result in **data inconsistency**.
- Maintaining **data consistency requires synchronization mechanisms** to ensure the orderly execution of cooperating processes.
- Synchronization itself requires some form of communication

# Why Process Synchronization?

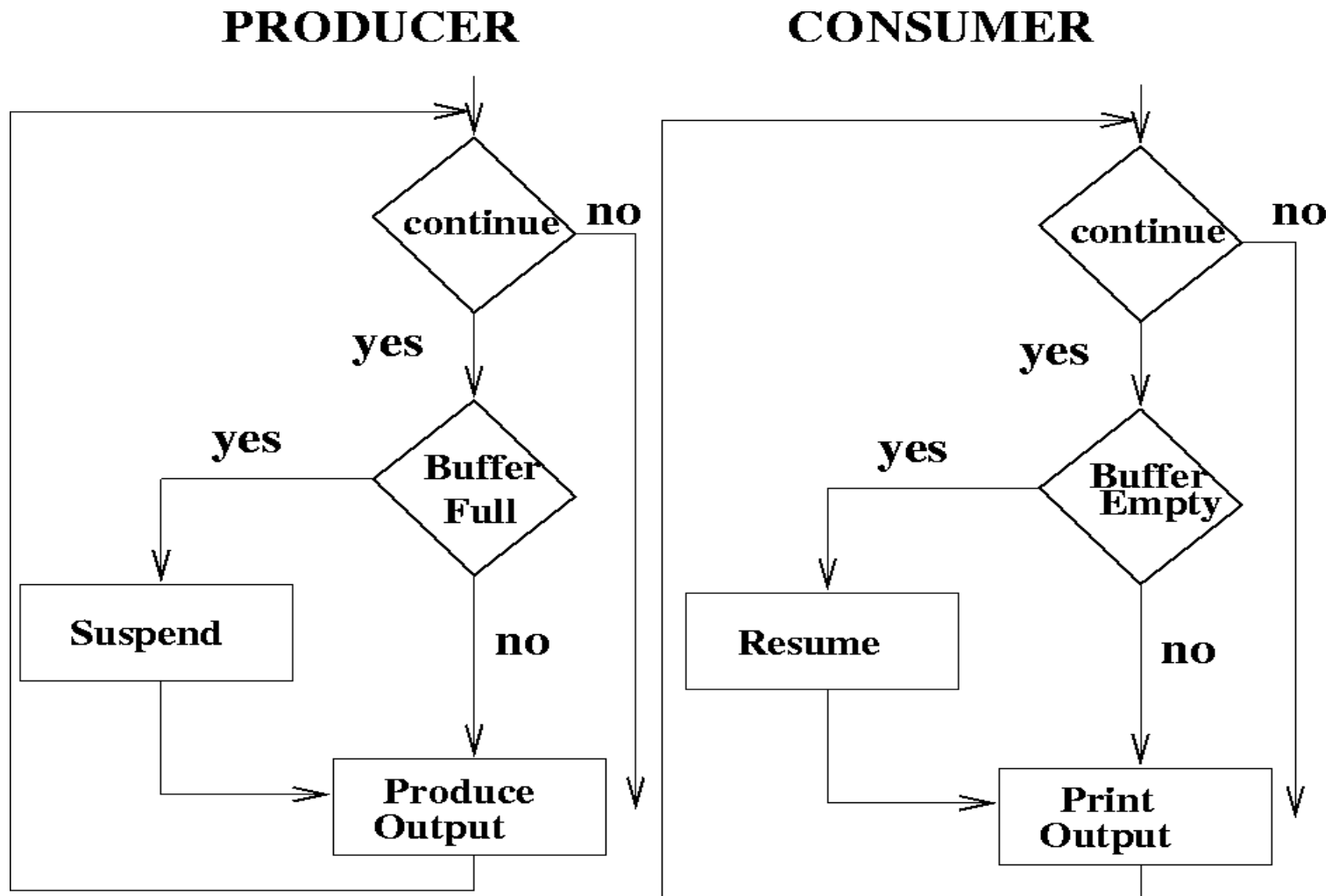
- **Concurrent execution without Synchronization can cause two major problems:**
- **Inconsistency due to Race Condition**
  - When two or more cooperating processes access and manipulate the same data concurrently, and
  - The outcome of the execution depends on the order in which the access takes place.
    - Counter = 0; // global variable
    - Thread 1 does Counter++
    - Thread 2 does Counter-- // “at the same time”
    - What is the order of values of Counter ?
      - 0 : 1 : 0?
      - 0 : -1 : 0?
- **Deadlock**
  - When two or more waiting processes require shared resource for their continued execution,
  - But the required resources are held by other waiting processes.
- **Let us return to the bounded buffer [producer-consumer] problem we presented before.**

```
shared int x = 3;
process_1 () {
    x = x + 1;
    print x; }
process_2 () {
    x = x - 1;
    print x; }
```

## Producer-Consumer Problem

- Example of Cooperating processes that need to be synchronized:
  - **Producer** process produces information that is consumed by a **Consumer** process.
- We need buffer of items that can be filled by producer and emptied by consumer.
  - **Unbounded-buffer** places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
  - **Bounded-buffer** assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
  - **Producer and Consumer must be synchronized. How?**

# Producer-Consumer Problem



# The Producer Consumer Problem

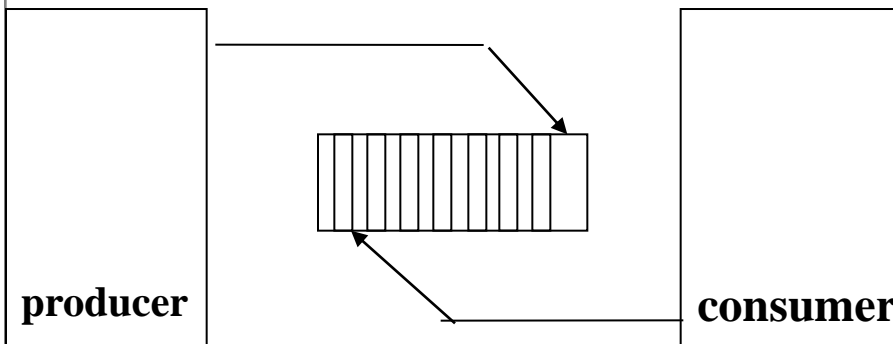
A **Producer** process "produces" information "consumed" by a **Consumer** process.

```
item    nextProduced;
while (1) {                                PRODUCER
    while (counter ==
        BUFFER_SIZE);
    /*do nothing*/
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    /*incremented every time we
    add element*/
    counter++;}
```

```
#define BUFFER_SIZE 10
typedef struct {
    DATA          data;
} item;
item    buffer[BUFFER_SIZE];
int     in = 0;
int     out = 0;
```

```
item    nextConsumed;

while (1) {                                CONSUMER
    while (counter == 0);
    /*do nothing*/
    nextConsumed = buffer[out];
    out = (out + 1) %
        BUFFER_SIZE;
    /*decremented every time we
    remove element*/
    counter--;
}
```





# Atomic Operation

- **Definition:** An atomic operation is the one that executes to completion without any interruption or failure.
- **Operations are often not “atomic”**
  - Example:  $x = x + 1$  is not atomic!
    - Read/**Load**  $x$  from memory into a register
    - Increment register (**x**)
    - Store register (**x**) back to memory
- An atomic operation has “**an all or nothing**” flavor:
  - Either it executes to completion, or
  - It does not execute at all, and
  - It executes without interruptions.

# The Producer Consumer Problem

Although both the producer and consumer function well separately, they may not function well when executed concurrently!!

we can show that by implementing the statement “**counter++**” in machine language as follows:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

The statement “**counter--**” implemented in machine language as follows:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

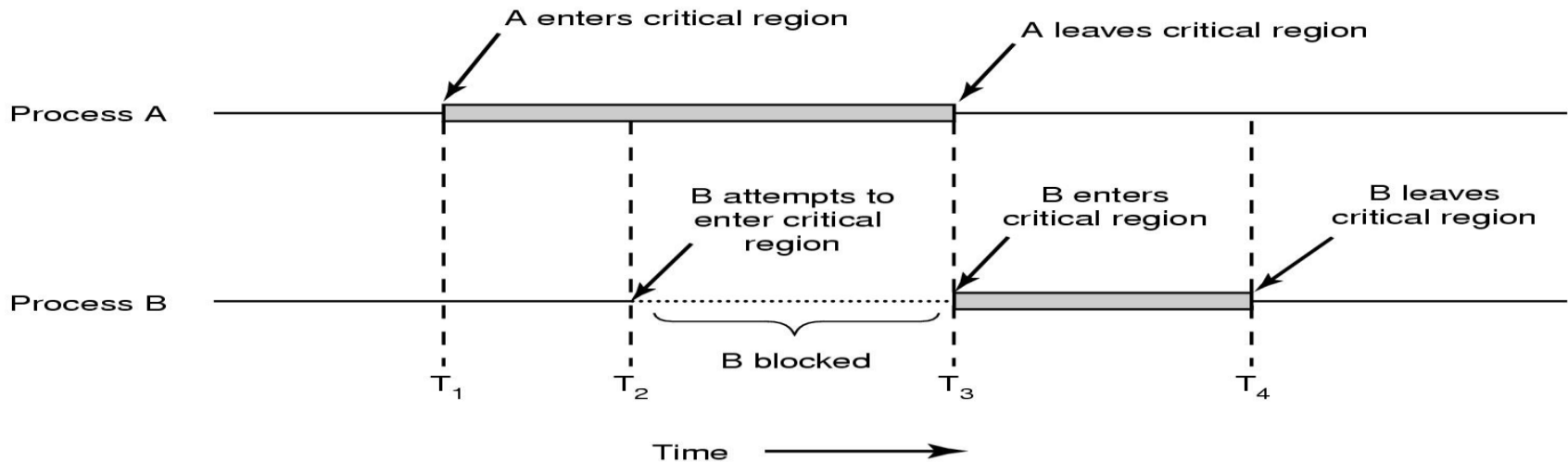
Where **register 1, 2** are local CPU registers.

At a micro level, the following scenario could occur using this code. **Assume counter is initially 5.**

T0;	Producer	Execute	register1 = counter	register1 = 5
T1;	Producer	Execute	register1 = register1 + 1	register1 = 6
T2;	Consumer	Execute	register2 = counter	register2 = 5
T3;	Consumer	Execute	register2 = register2 - 1	register2 = 4
T4;	Producer	Execute	counter = register1	counter = 6
T5;	Consumer	Execute	counter = register2	counter = 4

# The Critical Section

- The previous example demonstrates a **critical section**: a memory location, or part of the process code, or disk space which is shared by **n** processes where all processes may be able to change stored values.
- Critical sections are used frequently in an OS to protect data structures (e.g., queues, shared variables, lists, ...)
- **Problem**: how to ensure that only 1 process can change the value at a time such that all processes know the current value.
- Critical sections must be protected so that they are **mutually exclusive** of processor access. **All code within the section executes atomically**



## Critical Sections

- **A critical section implementation must be:**
  - **Correct/Mutual Execution:** only **1** thread/process can execute in the critical section at any given time.
  - **Efficient:** Getting into and out of critical section must be fast. Critical sections should be as short as possible.
  - **Concurrency control:** A good implementation allows maximum concurrency while preserving correctness.
  - **Flexible:** A good implementation must have as few restrictions as practically possible.

## Mutual Exclusion: Disabling Interrupts

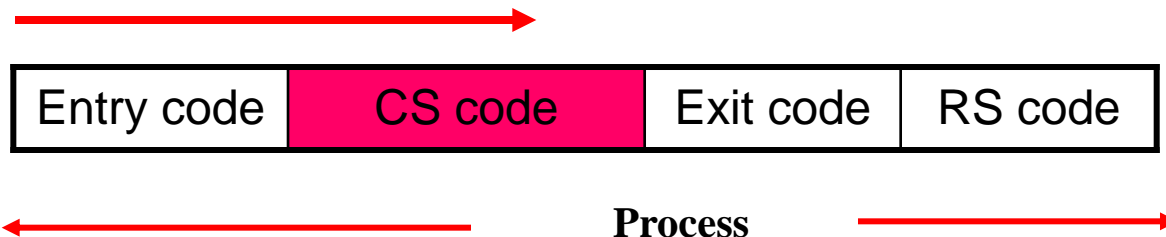
- The critical section problem could be solved simply in uni-processor environment if:
- Interrupts are disabled while a shared variable is being accessed.
- Without interrupts no process context switching can occur and this supports mutual exclusion.
- Somewhat dangerous: one process can hold up the CPU forever
  - Endless loop
  - Waiting for resources
  - Efficiency of execution may be degraded
  - Processor has limited ability to interleave programs
- Used in special-purpose systems with limited hardware
- In Multiprocessor system
  - Disabling interrupts on one processor will not guarantee that other processors can be interrupted and that means mutual exclusion can not be supported.
  - Can we disable interrupts in all processors?

# Critical Section

- **do{**
- **entry section**
- **critical section**
- **exit section**
- **remainder section**
- **} while (TRUE)**

A Critical Section environment should contain:

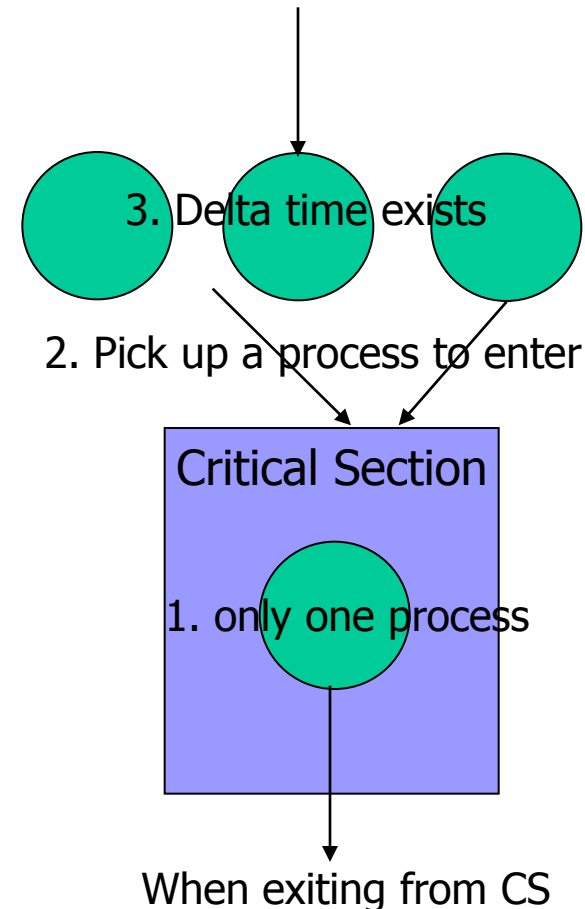
<b>Entry Section</b>	Code requesting entry into the critical section.
<b>Critical Section</b>	Code in which only one process can execute at any one time.
<b>Exit Section</b>	The end of the critical section, releasing or allowing others to in.
<b>Remainder Section</b>	Rest of the code after the critical section.



# Evaluating Critical Section Solution

A solution to the Critical Sections (CS) must satisfy: When coming and entering CS

1. **Mutual Exclusion:** At most, one process/thread is allowed to execute in a shared CS.
2. **Progress:** If a process/thread wishes to execute its CS (and no other processes/threads are currently executing in it) then the processes which are allowed to decide if this process/thread gains entrance are those not currently executing their remainder section.
3. **Bounded waiting:** If a process/thread  $i$  is in entry section, then there is a bound on the number of times that other processes/threads are allowed to enter the critical section before process/thread  $i$ 's request is granted.
  1.  $P_1$  wants to enter CS
  2. Any  $P_i$  enters its CS at most  $N$  times after  $P_1$ 's request
  3. After  $N$  entries,  $P_1$  must enter its CS
  4.  $N$  must be bounded



## Initial Attempts to Solve the CS Problem

- We restrict our attention to the algorithms that are applicable to only **two processes** at a time:
- Two processes **P0** and **P1** (also noted as **Pj** and **Pi**)  
    **do** {  
        *entry section*  

critical section

  
        *exit section*  

reminder section

  
    **} while (True);**
- Processes may share some common variables to synchronize their actions.



# Algorithm 1

- Uses **turn** variable to alternate entry into critical sections between two processes.

**int turn;**

initially **turn = 0**

**turn = i**  $\Rightarrow P_i$  can enter its critical section

Process  $P_i$

**do** {

**while (turn = j) ;**

critical section

**turn = j;**

remainder section

} **while (True);**

## Algorithm 1

- This algorithm ensures that only one process can access the CS at a time (**mutual exclusion**)
- The process of entering the critical section only tests the shared turn variable.
- Missing the other processes set just delays its entry
- **It does not satisfy the progress requirement.**
- The algorithm does not retain the state of each process in order to get around this problem. It remembers only which process is allowed to enter the CS.
  - If  $turn=0$ , P0 is executing its remainder section and P1 is ready to enter the CS, **it is denied since  $turn \neq 1$**
  - However, P0 cannot agree to turn over the CS to P1 since it is currently in its remainder section
  - $P_i, P_j, P_i, P_j \dots$  strict alternation of processes
  - $P_i$  leaves,  $P_j$  busy with long I/O,  $P_i$  comes back to CS-entry;
  - No one in the CS, but  $P_i$  has to wait until  $P_j$  to come to the CS.
  - What if  $P_j$  never comes back to CS ????

## Algorithm 2 (saying I'm using)

- Shared Variables
    - **var** flag: **array** (0..1) **of** Boolean;  
initially flag[0] = flag[1] = false;
    - flag[i] = true ☒ P<sub>i</sub> ready to enter its critical section
  - Process P<sub>i</sub>
    - repeat**
      - flag[i] := true;
      - while** flag[j] **do** no-op;  
critical section
      - flag[i] := false;  
remainder section
    - until** false
- Can block indefinitely.... Progress requirement not met.

### Algorithm 3: Combining Alg.1 & Alg. 2

- By combining the ideas behind Alg.1 and Alg. 2, we can ensure all 3 properties
- Two process solution
- Processes share both flag and turn variables
- Two shared variables:
  - `int turn;`
    - `turn` indicates whose turn it is to enter the critical section
  - `Boolean flag[2]`
    - `flag[i] = true` implies that process  $P_i$  is ready
- Each processes sets a flag to request entry. Then each process toggles a bit to allow the other in first.

### Algorithm 3: Combining Alg.1 & Alg. 2

- Uses **flag** variables to show requests by processes wishing to enter their critical sections.
  - Process checks the flag of another process and doesn't enter the critical section if that process wants to get in.
  - Flag array[0..1] of Boolean;
  - Flag[0] and Flag[1] are initialized to false
  - If Flag[i] is true then **P<sub>i</sub>** is ready to enter the CS
    - Do {  
    flag[i] = true;  
    turn=j;
- /\*check to see that P<sub>j</sub> is false [not ready]\*/*
- while (flag[j] && turn==j);  
    [critical section]
- /\*if P<sub>j</sub> is true then P<sub>i</sub> should wait\*/*
- flag[i] = false;  
    [remainder section]  
} while (True);
- Still accomplishes mutual exclusion
  - But does not accomplish progress
- T0: P0 sets flag [0] = true
- T1: P1 sets flag [0] = true
- If both flag[0] and flag[1] are true simultaneously (which could occur if both processes decide to access the CS at about the same time) then they both wait forever.

## Algorithm 3: Combining Alg.1 & Alg. 2

```
while (true) {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);
```

CRITICAL SECTION

```
flag[0] = FALSE;
```

REMAINDER SECTION

```
}
```

```
while (true) {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);
```

CRITICAL SECTION

```
flag[1] = FALSE;
```

REMAINDER SECTION

```
}
```

## Algorithm 3 Evaluation

- **Mutual Exclusion** is maintained because turn will allow only 1 process access at a time
- **Progress and bounded-waiting** are maintained because a process  $i$ , is only denied access to the CS while both  $\text{turn} \neq i$  (I.e.  $\text{turn} = j$ ) and  $\text{flag}[i]$  is false.
- $\text{Flag}[i]$  will be true when the process wants to enter the CS and if  $\text{turn} \neq i$ , process  $i$  must only wait until process  $j$  terminates with the CS (which must happen in a finite amount of time)

# Lock

- Suppose we have some sort of implementation of a lock.
  - **Lock.Acquire()** – wait until lock is free, then grab
  - **Lock.Release()** – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our critical section problem is easy:  
    **lock.Acquire();**  
Critical section  
    **lock.Release();**
- Once again, section of code between **Acquire()** and **Release()** called a “**Critical Section**”



## Mutual Exclusion: HW Instructions

- Special machine (**atomic**) instructions that are available on many systems can be used to solve the critical section problem.
  - Performed in a single instruction cycle
  - Not subject to intrusion from other instructions
  - i.e. Reading and writing
  - i.e. Reading and testing
- These instructions can do two steps indivisibly [**atomically**]
  - **Test\_and\_set**: Test a value; if it is **false** set it to **true**, else leave it **True**
  - **Exchange**: Swap the values of two variables
- Often in combination with interrupts
- Sometimes used as basis for OS synchronization mechanisms

## Mutual Exclusion: Test & Set Instruction

- The important characteristic of the T&S instruction is that it is executed **atomically**. i.e. it is un-interruptible; no other process can access the memory location until the instruction is finished. **It is used to put a “lock” on a memory word.**
- **Implementation of Mutual Exclusion with Test-and-Set**
- **A physical entity** often called a **lock byte** **must be used to represent the resource.**
- There should be a **lock byte** associated with each shared database/device.
- **lock byte = 0** means the resource is available, **lock byte = 1** means the resource is in use.
- Before operating on a shared resource, a process must perform the following actions:
  1. Examine the value of the lock byte, (**Test**)
  2. Set the lock byte to 1, (**Set**).
  3. If the original value was 1, go back to step 1.
- T & S can be used to implement mutual exclusion as follows:

# Mutual Exclusion with Test-and-Set

```
Boolean TestAndSet (Boolean &lock) {  
    Boolean tmp = lock;  
    lock = true;  
    return tmp;    }
```

When calling *TestAndSet*(lock)

- if lock = False before calling *TestAndSet*
  - It is set to True and False is returned
- if lock = True before calling *TestAndSet*
  - It is set to True and True is returned

Shared data:

```
Boolean lock = False;
```

Process  $P_i$  do {

```
    while (TestAndSet(lock) = True); // recourse is in use (do nothing)
```

```
        -- critical section --
```

```
        lock = False; // means the resource became available
```

```
        -- remainder section--
```

```
    } while (1);
```

- This algorithm satisfies the mutual exclusion and progress requirements, but not the bounded-waiting requirement.
- It may be sufficient for synchronization, but wasteful of processor resources.
- The blocked process doesn't really stop executing, instead it continually loops, testing the lock byte and waiting for it to change to 0 (**Busy Waiting**).

# Mutual Exclusion with Swap

*(Atomically swap two variables)*

```
void Swap(Boolean a, Boolean b) {  
    Boolean temp = a;  
    a = b;  
    b = temp; };
```

- After calling swap,
  - a = original value of b
  - b = original value of a
- Shared data (initialized to false):  
**Boolean lock = false; /\* shared variable - global \*/**  
**// if lock = 0, door open, if lock = 1, door locked**
- Private data  
**Boolean key\_i = true;**
- Process  $P_i$   
do {  
key\_i = true; **/\* not needed if swap used after CS exit \*/**  
while (key\_i = true)  
Swap(lock, key\_i );  
**critical section /\* remember key\_i is now false \*/**  
lock = false; **/\* can also use: swap(lock, key\_i );\*/**  
**remainder section**  
}

- Declaring a **global** Boolean variable **lock**, initialized to **false**.
- In addition, each process has a **local** Boolean variable **key**.
- **Mutual Exclusion:** Pass if key = T or waiting[i] = F
- **Progress achieved** because exit process sends a new process in.
- **Bounded Waiting** achieved because each process wait at most n-1 times

- **Advantages**
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory.
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections
- **Disadvantages**
  - An explicit flush of the write to main memory
  - **Busy-waiting** consumes processor time, while a process is in CS, others should loop in the entry sections.
  - **Starvation** is possible when a process leaves a critical section and more than one process is waiting. *Who is next?*
  - **Deadlock** - If a low priority process has the critical region and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical region.

## Busy Waiting/**Spin-Lock**

- Main disadvantage of the mutual exclusion solutions shown in the previous algorithms is **BUSY WAITING** or **SPIN-LOCK**.
- **Busy Waiting**
  - Process is continuously looping in the entry section to see if it can enter the critical section.
  - Process can do nothing productive until it gets permission to enter its critical section – wastes CPU cycles.
- Most mutual exclusion solutions result in “busy waiting/Spin-lock”
- **To overcome this we can use a *wait and signal* mechanism.**
- Spin-Lock in multiprocessor system is useful as it minimizes the context switch process.

## Mutual Exclusion with Wait and Signal

We can modify the lock mechanism as follows to avoid busy waiting:

- (I) Examine the value of the lock byte.
- (ii) Set the lock byte to 1
- (iii) If the original value was 1, call Wait (X).

To unlock:

- (I) Set lock byte to 0
- (ii) Call Signal (X).

Wait and Signal are primitives of the traffic controller.

- A Wait (X) sets the process' PCB to the blocked state and links it to the lock byte X.
- Another process is then selected to run by the scheduler.
- A Signal (X) checks the blocked list associated with the lock byte X;
- If there are any processes blocked, waiting for X, one is selected and its PCB is set to the ready state.
- Eventually, the scheduler will select this newly “awakened” process for execution.

# Semaphore

- Synchronization tool that does not require busy waiting.
- An abstract data type, **non-negative** integer
- Synchronization operations are atomic.
  - Operation **P** = wait
  - Operation **V** = signal
- Use
  - *N*-process critical section problems
  - Synchronization problems
- Semaphore **S** – non-negative integer variable

Semaphore **S**; // initialized to 1

```
acquire(S);  
criticalSection();  
release(S);
```

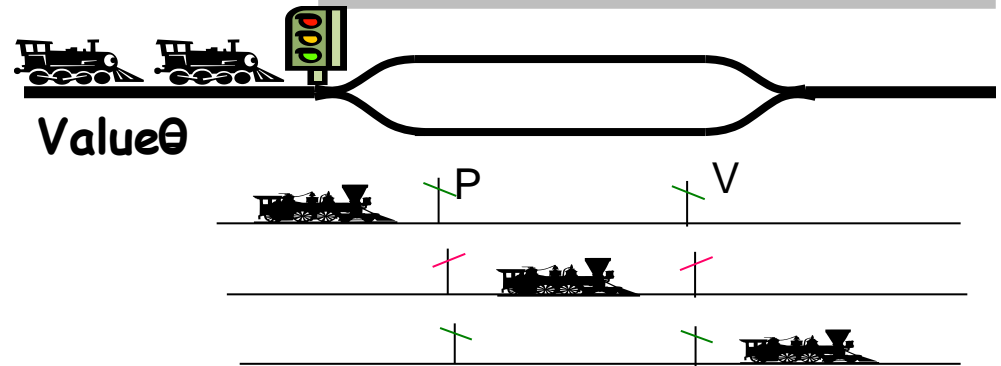
- When one process modifies the semaphore value, no other process can simultaneously modify it.
- The testing of a semaphore value and either increment or decrement it should be done atomically.

**Semaphore** → **P()** (wait)

If *sem* > 0, then  
decrement *sem* by 1  
Otherwise “wait” until  
*sem* > 0 and then  
decrement

**Semaphore** → **V()** (signal)

Increment *sem* by 1  
Wake up a thread waiting  
in P()





# Semaphore Implementation

- Each semaphore has an **integer value** and a list of associated processes
- When a process blocks/waits itself on a semaphore, it is added to a queue of waiting processes.
- The **signal** operation [from other processes] on a semaphore restarts a process from the queue and wakes the process up by the wakeup operation.

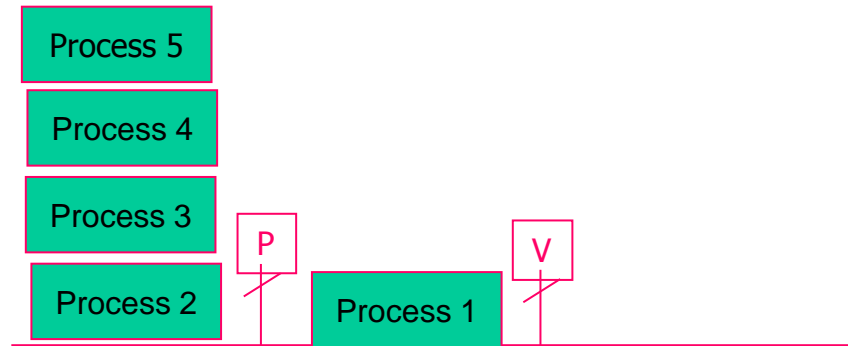
- Define a semaphore as a record

**type semaphore = record**

**value**: integer

**L**: list of process;

**end;**



## Critical Section of N Processes

- We can use semaphores to deal with the n-process critical-section problem.
- N processes share a semaphore mutex; // stands for mutual exclusion) initially  $mutex = 1$
- Process  $P_i$ :  
do {  
    wait (mutex);  
    critical section  
    signal (mutex);  
    remainder section  
} while (1);
- Shows use of semaphore to implement mutual exclusion.
- Semaphores can be used to solve various synchronization problems,

Wait (mutex);

Critical section

Signal (mutex);

## Implementation

- When a process executes **wait** and finds the semaphore value  $< 0$ , it **blocks** itself rather than **looping**.

*wait (S):*

```
S.value--;  
if (S.value < 0) {  
    add this process to the list;  
    block;  
}
```

- Signal** restarts a process using **wakeup** which **moves** a process from the wait queue to the ready queue.

*signal (S):*

```
S.value++;  
if (S.value <= 0) {  
    remove a process P from the list;  
    wakeup(P);  
}
```

- block** suspends the process that invokes it.
- wakeup(P)** resumes the execution of a blocked process P.
- Block** and **wakeup** are provided by the OS as basic system calls.

## Busy Waiting/**Spin-Lock** Solution

- **Solution:** When a process executes **wait** and finds the semaphore value  $< 0$ , it **blocks** itself rather than **looping** which transfers control to the scheduler that selects another process to execute.

## Semaphore Usage - Example

- **Goal:** Force P2 to execute after P1
- Using a common semaphore **synch** to synchronize the operations of the two concurrent processes:
  - **Wait**, signal utilized to delay P2 until P1 is done
  - **Synch** initialized to 0

**P1**

process 1 statements

**signal(synch);**

done executing

**P2**

since  $\text{synch} = 0$ ,  
must stay idle until  
signal from P1  
**wait(synch);**

received signal from P1

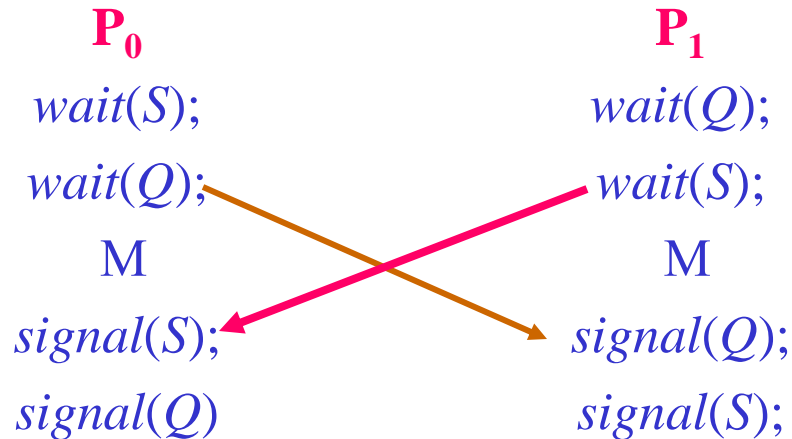
process 2 statements

## Dangers of Handling the Critical Section Problem

- Solutions to the critical section problem may lead to:
  - Starvation
  - Deadlock
- **Starvation:** A process never gets a resource because the resource is allocated to other processes
  - Higher priority
  - Consequence of scheduling algorithm
- Frequent solution: **aging**
  - The longer a process waits for a resource, the higher its priority until it eventually has the highest priority among the competing processes
- **Deadlock:** Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

## Deadlock and Starvation with Semaphore

- Let  $S$  and  $Q$  be two semaphores initialized to 1



- $P_0$  is waiting for  $P_1$  to execute  $signal(Q)$
- $P_1$  is waiting for  $P_0$  to execute  $signal(S)$
- Both processes are in a deadlock!**
- Starvation:** A process may never be removed from the semaphore queue in which it is suspended.
  - LIFO queue implementation.

## Two Types of Semaphores

- **Counting semaphore:** Integer value can range over an unrestricted domain.  
i.e the one described before.
- **Binary semaphore:** Integer value can range only between 0 and 1.
  - Can be simpler to implement depending on the underlying HW support.
  - Used by 2 processes to ensure only one can enter critical section.
- We can implement a counting semaphore **S** with binary semaphores.



## Implementing Counting Semaphore S

- Can we implement a counting semaphore with binary semaphores? **Yes**
- Data structures: **binary-semaphore S1, S2;**  
**int C;**
- Initialization:  
**S1 = 1**  
**S2 = 0**  
**C = initial value of the counting semaphore S**
- *wait* operation on the counting semaphore **S** can be implemented as:  
**wait(S1);**  
**C--;**  
**if (C < 0) {**  
    **signal(S1);**  
    **wait(S2);**  
**}**  
**signal(S1);**
- *signal* operation on the counting semaphore **S** can be implemented as:  
**wait(S1);**  
**C ++;**  
**if (C <= 0)**  
    **signal(S2);**  
**else**  
    **signal(S1);**

## Incorrect using of Semaphores

- Although semaphores provide an effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect.
- All processes share a semaphore variable **mutex**, which is initialized to 1.
- Each process must execute:  
    wait (mutex); before entering  
    critical-section  
    signal (mutex);
- Suppose that a process interchange the order of wait and signal, i.e:  
    signal (mutex);  
    critical-section  
    wait (mutex);  
    Several processes may be executing their CS simultaneously
- Suppose that a process replaces wait with signal, i.e:  
    signal (mutex);  
    critical-section  
    signal (mutex);  
    Deadlock may occur
- Suppose that a process omits the wait or the signal or both: in this case either mutual exclusion is violated or a deadlock will occur.

## Example Semaphores

- Three processes all share a resource on which
  - One draws an A
  - One draws a B
  - One draws a C
- Implement a form of synchronization so that A B C appears in this sequence.

**Process A**

```
think();  
draw_A();
```

**Process B**

```
think();  
draw_B();
```

**Process C**

```
think();  
draw_C();
```

## Example Semaphores

No semaphores

**A**  
think();  
draw\_A();

**B**  
think();  
draw\_B();

?

**C**  
think();  
draw\_C();

## Example Semaphores

No semaphores

**A**  
think();  
draw\_A();

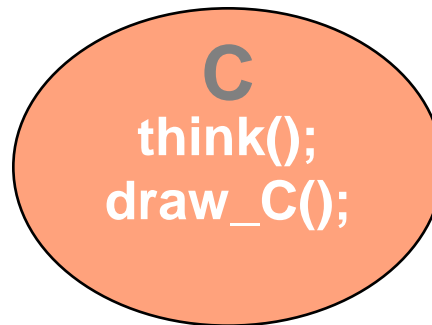
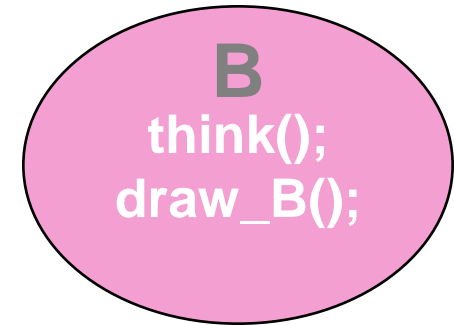
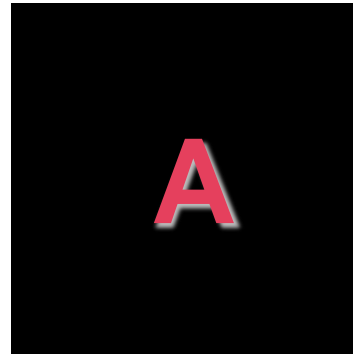
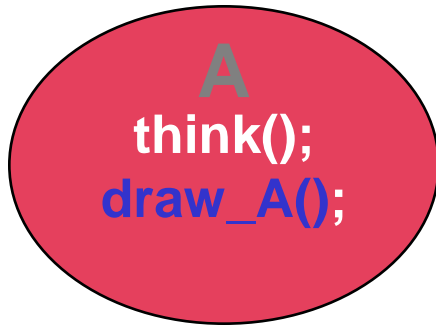
**B**  
think();  
draw\_B();

Race  
Condition !

**C**  
think();  
draw\_C();

## Example Semaphores

No semaphores



## Example Semaphores

No semaphores

**A**  
think();  
draw\_A();

**B**  
think();  
draw\_B();

**C**

**C**  
think();  
draw\_C();

## Example Semaphores

No semaphores

**A**  
think();  
draw\_A();

**B**

**B**  
think();  
draw\_B();

**C**  
think();  
draw\_C();



## Example Semaphores

**Semaphores  $b = 0, c = 0$ ;**

### Process A

```
think();  
draw_A();  
b.signal();
```

### Process B

```
b.wait();  
think();  
draw_B();  
c.signal();
```

### Process C

```
c.wait();  
think();  
draw_C();
```

## Example Semaphores

### Semaphores

$b = 0$

$c = 0$

**A**

```
think();  
draw_A();  
b.signal();
```

**B**

```
b.wait();  
think();  
draw_B();  
c.signal();
```

**C**

```
c.wait();  
think();  
draw_C();
```

## Example Semaphores

### Semaphore

$b = -1$

$c = -1$

A

```
think();  
draw_A();  
b.signal();
```

B

```
b.wait();  
think();  
draw_B();  
c.signal();
```

C

```
c.wait();  
think();  
draw_C();
```

## Example Semaphores

A

```
think();  
draw_A();  
b.signal();
```

Semaphore

$b = -1$

$c = -1$

A

B

```
b.wait();  
think();  
draw_B();  
c.signal();
```

C

```
c.wait();  
think();  
draw_C();
```

## Example Semaphores

### Semaphore

$b = 0,$   
 $c = -1;$

A

```
think();  
draw_A();  
b.signal();
```

B

```
b.wait();  
think();  
draw_B();  
c.signal();
```

C

```
c.wait();  
think();  
draw_C();
```

## Example Semaphores

**A**

```
think();  
draw_A();  
b.signal();
```

**Semaphore**

**b = 0**

**c = -1**

**B**

**B**

```
b.wait();  
think();  
draw_B();  
c.signal();
```

**C**

```
c.wait();  
think();  
draw_C();
```

## Example Semaphores

**Semaphore**

**b = 0,**

**c = 0;**

**A**

```
think();  
draw_A();  
b.signal();
```

**B**

```
b.wait();  
think();  
draw_B();  
c.signal();
```

**C**

```
c.wait();  
think();  
draw_C();
```

## Classical Problems of Synchronization

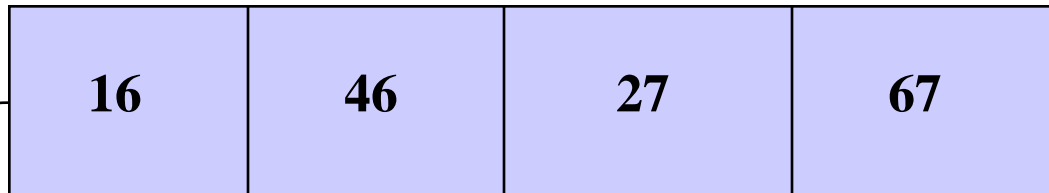
- The following problems are used for testing nearly every newly proposed synchronization scheme:
  - **Bounded-Buffer Problem**
    - Also known as the “consumer-producer” problem
  - **Readers and Writers Problem**
    - Exclusive access to shared object/DB when modification of the object/DB is required
  - **Dining-Philosophers Problem:** Need to allocate several resources among several processes without deadlock and starvation.



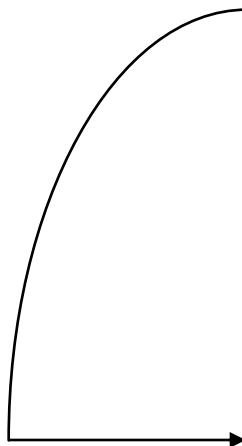
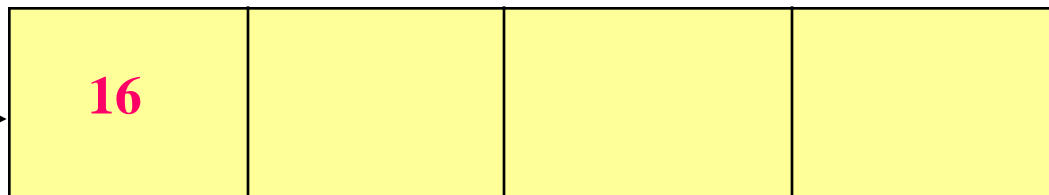
## Bounded Buffer or “Consumer-Producer”

- Two processes (one producer, one consumer) share a common, fixed-size buffer
- Producer places information into the buffer
- Consumer takes it out

**Producer:**



**Consumer:**



# Producer-Consumer Problem

- Why do we need synchronization?
  - The producer wants to place a new item into the buffer, but the buffer is already full
  - Consumer wants to consume an item, but the buffer is empty
- Solution:
  - If the buffer is full the producer goes to sleep,
    - Wakes up when the consumer has emptied one or more items.
  - If buffer is empty, consumer goes to sleep,
    - Wakes up when the producer has produced items
- Race conditions may occur
  - Wakeup call might be lost
  - Producer will eventually fill buffer and then goes to sleep
  - Consumer will also sleep
  - Both will sleep forever

## Semaphore Solution

- The structure of the producer process:

```
while (true) {
```

```
    // produce an item
```

```
    wait (empty); // initially empty = N
```

```
    wait (mutex); // initially mutex = 1
```

```
    // add the item to the buffer
```

```
    signal (mutex); // currently mutex = 0
```

```
    signal (full);  // initially full = 0
```

```
}
```

# Semaphore Solution

- The structure of the consumer process

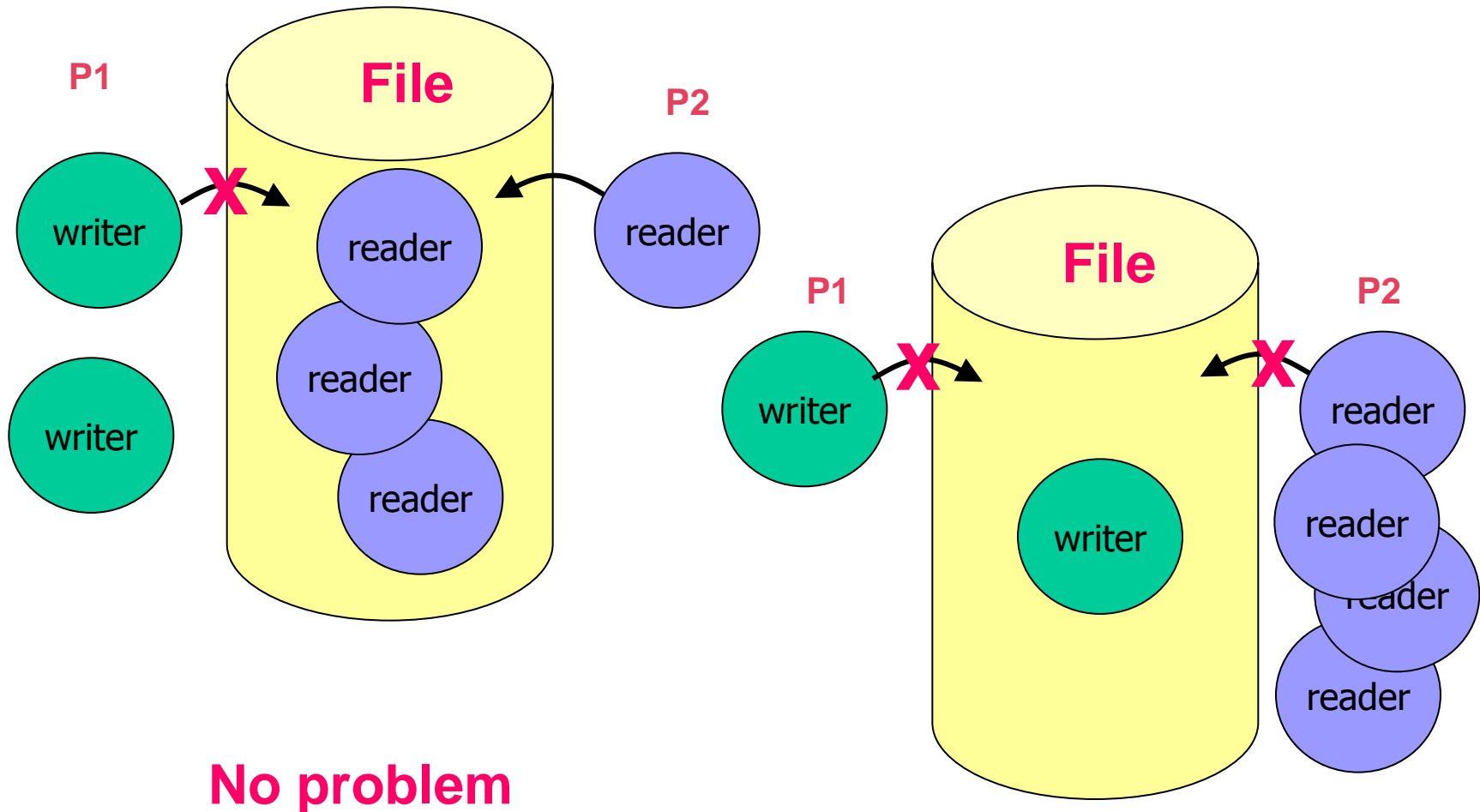
```
while (true) {  
    wait (full);      // initially full = 0  
    wait (mutex);     // initially mutex = 1  
  
    // remove an item from buffer  
  
    signal (mutex);   // currently mutex = 0  
    signal (empty);   // initially empty = N  
  
    // consume the removed item  
  
}
```

# Readers-Writers

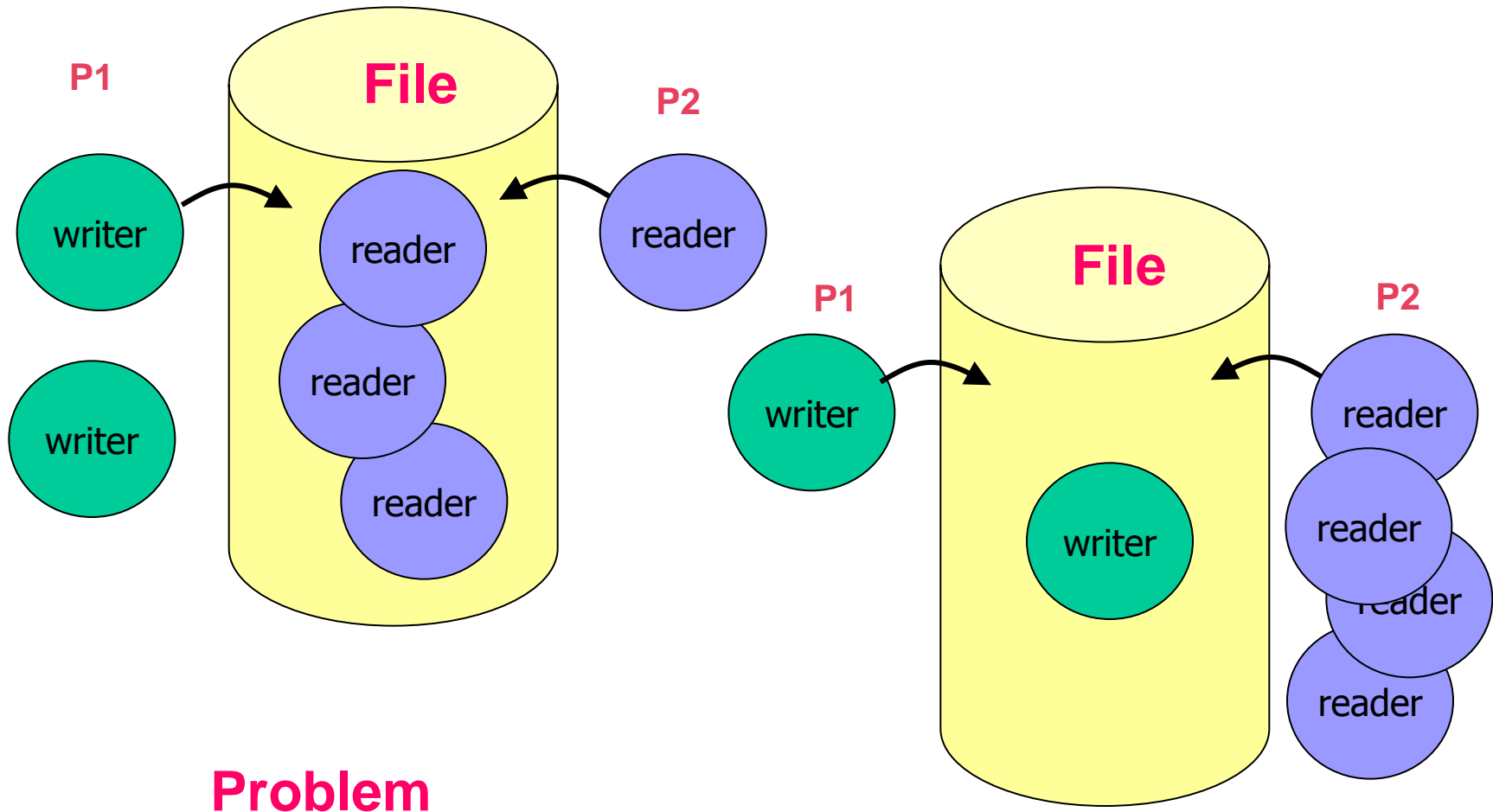
- Concurrent processes share a file, record, or other resources
- Some may read only (readers), some may write (writers)
- Two concurrent reads have no adverse effects
- Problems if
  - Concurrent reads and writes
  - Multiple writes
  - May result in starvation, deadlock
- Race conditions may occur if the resource is modified by two processes simultaneously
- **Solution:** use semaphores:
  - Semaphore **mutex** initialized to 1.
  - Semaphore **wrt** initialized to 1.
  - Integer **readcount** initialized to 0.

## Classical Problem 2: The Readers-Writers Problem

- Multiple readers or a single writer can use DB.



## Classical Problem 2: The Readers-Writers Problem



# Readers-Writers Problem

- The structure of a writer process

```
while (true) {  
    wait (wrt) ;  
  
    //  writing is performed  
  
    signal (wrt) ;  
}
```



# Readers-Writers Problem

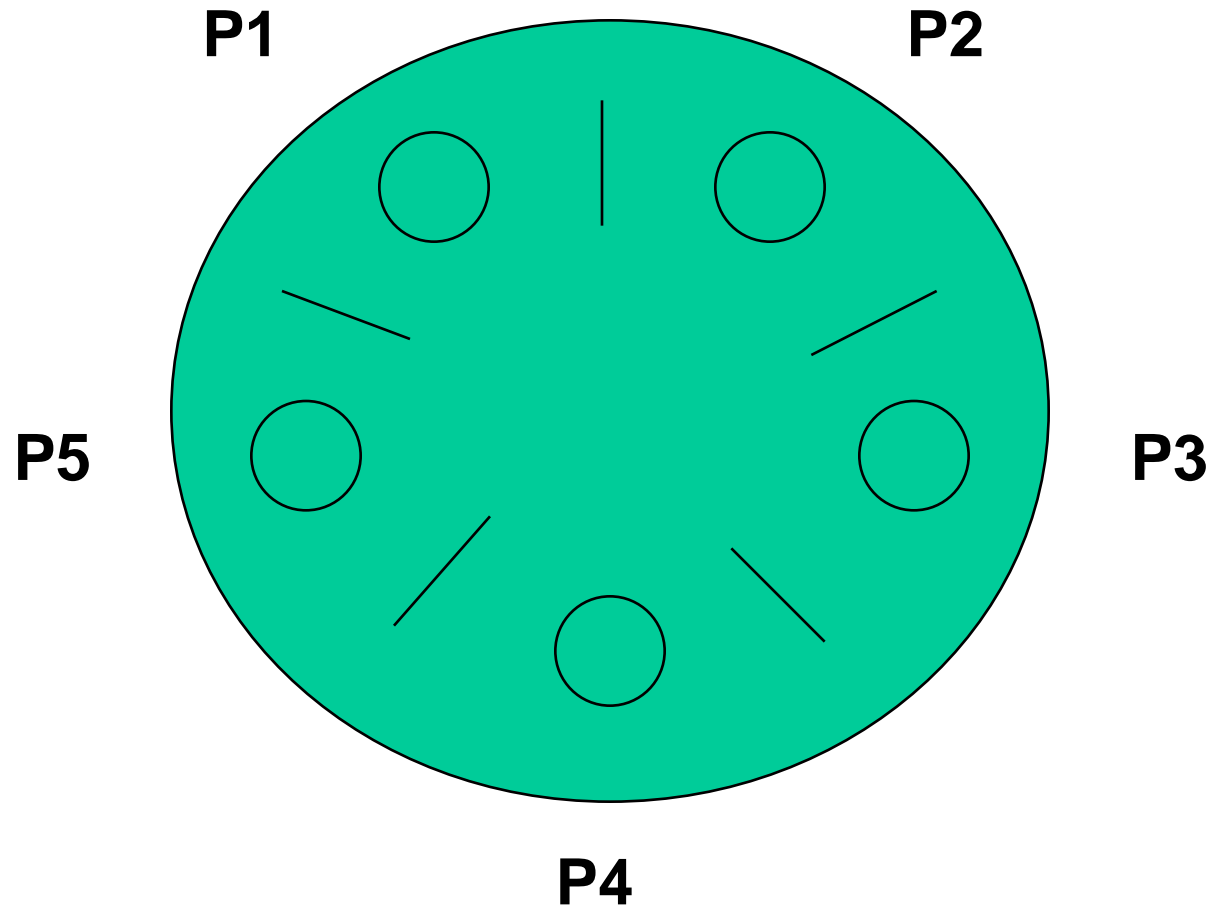
- The structure of a reader process

```
while (true) {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1) wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount - - ;  
    if (readcount == 0) signal (wrt) ;  
    signal (mutex) ;  
}
```

# Dining Philosophers

- Five philosophers sit at a round table - thinking and eating
- Each philosopher has one chopstick
  - Five chopsticks total
- A philosopher needs two chopsticks to eat
  - Philosophers must share chopsticks to eat
- No interaction occurs while thinking
- Problem:
  - **Starvation**
    - A philosopher may never get the two chopsticks necessary to eat
  - **Deadlocks**
    - Two neighboring philosophers may try to eat at same time
- Solution:
  - Utilize semaphores to prevent deadlocks and/or starvation
  - **Each chopstick is represented by a semaphore**
- Advantages
  - Guarantees that no two neighbors will attempt to eat at the same time

# Dining Philosophers Diagram



## Possible Solution

- One semaphore per philosopher

**var** chopstick: **array** [5] **of** semaphore; / **all initialized to 1**

**repeat**

wait(chopstick[i]);

wait(chopstick[i + 1 **mod** 5]); // **no two neighbors will eat at the same time**

...

**eat**

...

signal(chopstick[i]);

signal(chopstick[i + 1 **mod** 5]);

...

**think**

...

**until** false;

## Outline of a Correct Solution

- Deadlock might happen if all philosophers decided to eat at the same time.
- Solution:
  - A philosopher is allowed to pick up chopsticks only if both are available.
  - Allow at most four philosopher to be sitting simultaneously at the table.
  - This requires careful coordination (e.g. critical sections)
  - Does not automatically resolve starvation

# Critical Regions

- To avoid the previous errors of semaphores, a high-level language synchronization construct called **critical-region**.
- We assume that the a process consist of some local data, and a sequential program that can operate on the data.
- The local data can be only accessed by only the sequential program encapsulated within the same process. (**one process can not access the data of another process**)
- A shared variable  $v$  of type  $T$ , is declared as:  
 **$v$ : shared  $T$**
- Variable  $v$  accessed only inside statement  
**region  $v$  when  $B$  do  $S$**

where  $B$  is a Boolean expression.

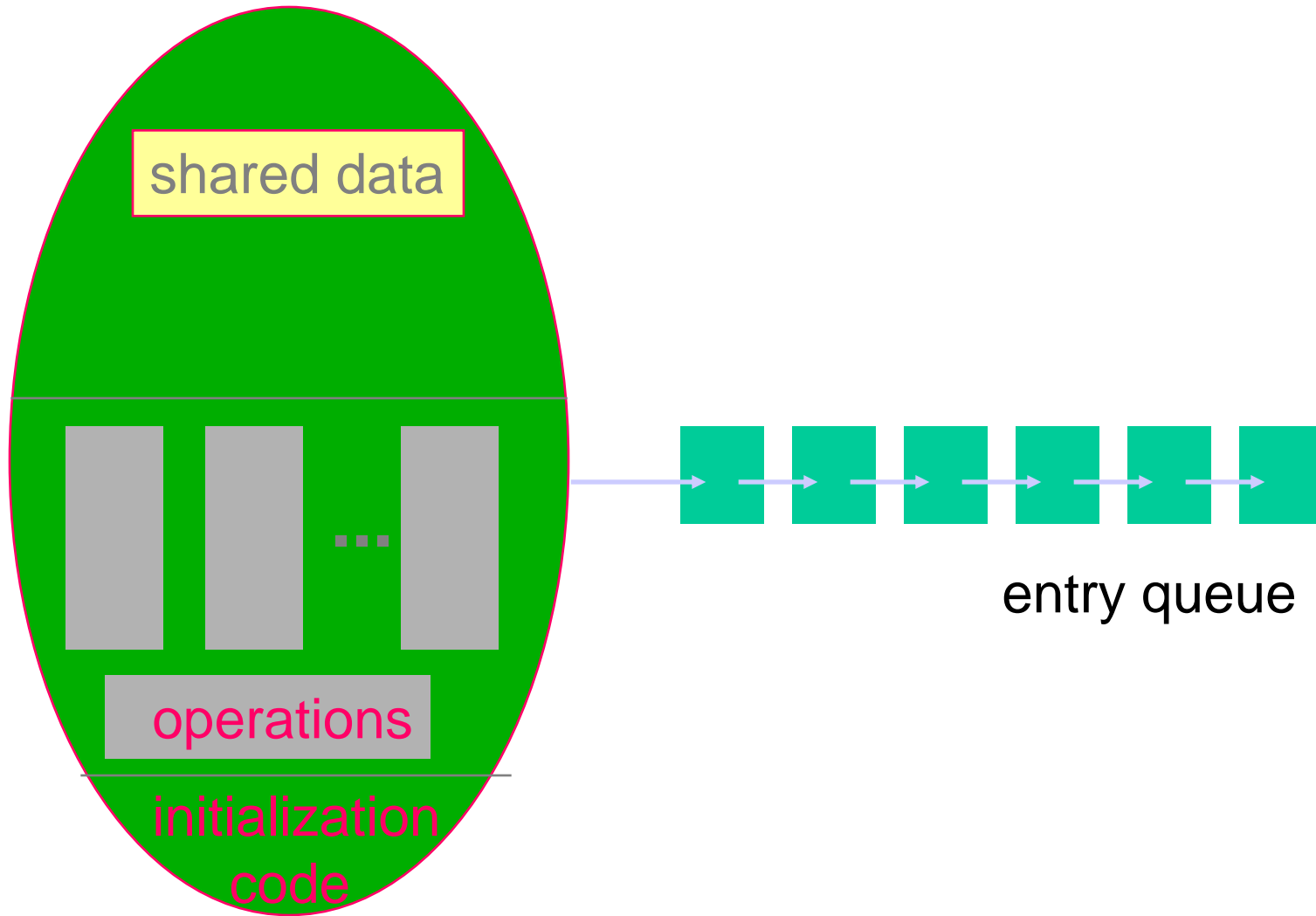
- While statement  $S$  is being executed, no other process can access variable  $v$ .
- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression  $B$  is evaluated. If  $B$  is true, statement  $S$  is executed. If it is false, the process is delayed until  $B$  becomes true and no other process is in the region associated with  $v$ .

# Monitors

- Another programmer-defined operators construct that allows the safe sharing of an abstract data type among concurrent processes.
- A monitor type consists of declarations of variables whose values define the state of an instance of the type and the procedures or functions that implement operations on the type.
- Below is the monitor syntax.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

# Monitor Diagram

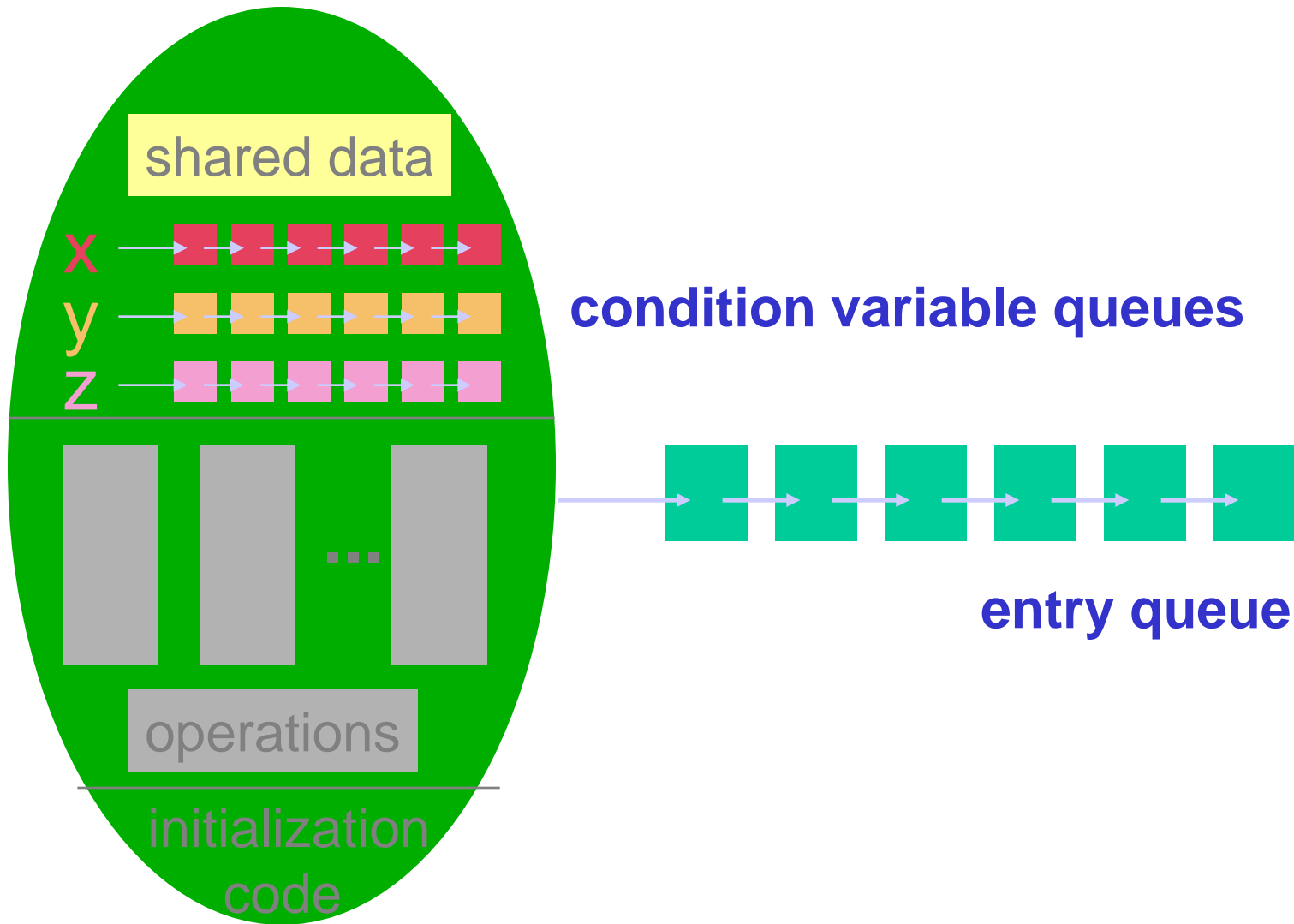




## A Problem with Monitors

- To allow a process to block themselves when they cannot proceed within the monitor, a **condition** variable must be declared, as  
**condition x, y;**
- The only operations that can be invoked on a condition variable is: **wait** and **signal**.
  - **x.wait** suspends the process until it is invoked by another process, and
  - **x.signal** releases exactly one process from the affiliated waiting queue
- Only usable in a few programming languages
- Solves mutual exclusion problem only for CPUs that all have access to common memory; not designed for distributed systems

# Monitor Diagram



## Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses adaptive **mutexes** for efficiency when protecting data from short code segments.
- Uses condition variables and readers-writers locks when longer sections of code need access to data.
- Uses **turnstile** to order the list of threads waiting to acquire either an adaptive **mutex** or **reader-writer** lock.

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uni-processor systems.
- Uses spinlocks on multiprocessor systems.
- Also provides dispatcher objects which may act as wither mutexes and semaphores.
- Dispatcher objects may also provide events. An event acts much like a condition variable.



**The End!!**

**Thank you**

**Any Questions?**