

Process Synchronization



Race Condition

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

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

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

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

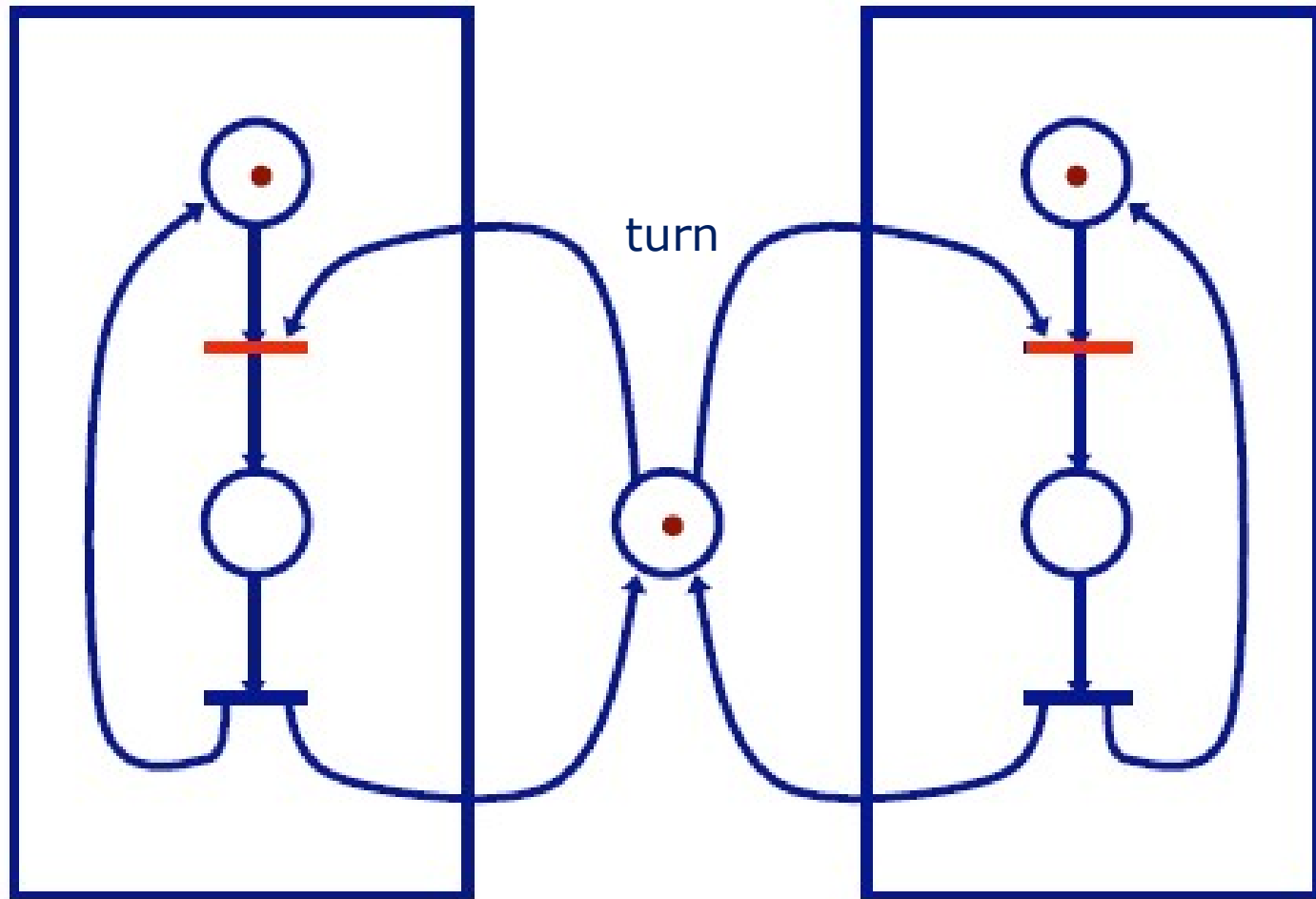
- Consider this execution interleaving with “count = 5” initially:

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}

Algorithm for Process P_i

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

Petri net diagram for mutual exclusion



Critical Section

■ General structure of process P_i

do {

entry section

critical section

exit section

remainder section

} while (true);

Conditions for the Solution

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 processes 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
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

acquire() and release()

```
■ acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}  
■ release() {  
    available = true;  
}  
■ do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

- **wait()** and **signal()**

- ▶ Originally called **P()** and **V()**

- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

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


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
- ```
typedef struct{
 int value;
 struct process *list;
} semaphore;
```

## Implementation with no Busy waiting (Cont.)

---

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

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

# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$   
Create a semaphore “**synch**” initialized to 0  
P1:  
     $S_1$ ;  
    **signal(synch)** ;  
P2:  
    **wait(synch)** ;  
     $S_2$ ;
- Can implement a counting semaphore  $S$  as a binary semaphore

# Deadlock and Starvation

- **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

| $P_0$                    | $P_1$                    |
|--------------------------|--------------------------|
| <code>wait(S) ;</code>   | <code>wait(Q) ;</code>   |
| <code>wait(Q) ;</code>   | <code>wait(S) ;</code>   |
| <code>...</code>         | <code>...</code>         |
| <code>signal(S) ;</code> | <code>signal(Q) ;</code> |
| <code>signal(Q) ;</code> | <code>signal(S) ;</code> |

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

# Monitors

---

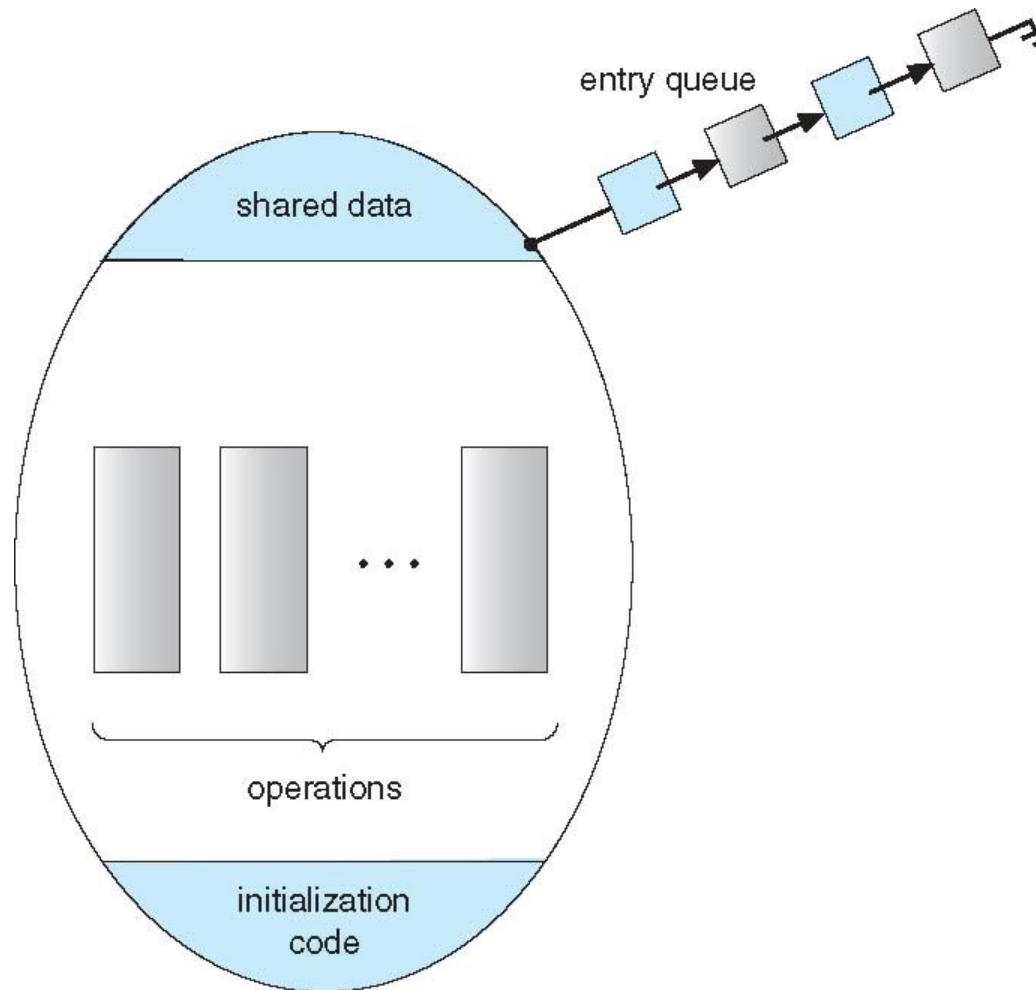
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
 // shared variable declarations
 procedure P1 (...) { ... }

 procedure Pn (...) {.....}

 Initialization code (...) { ... }
}
}
```

# Schematic view of a Monitor

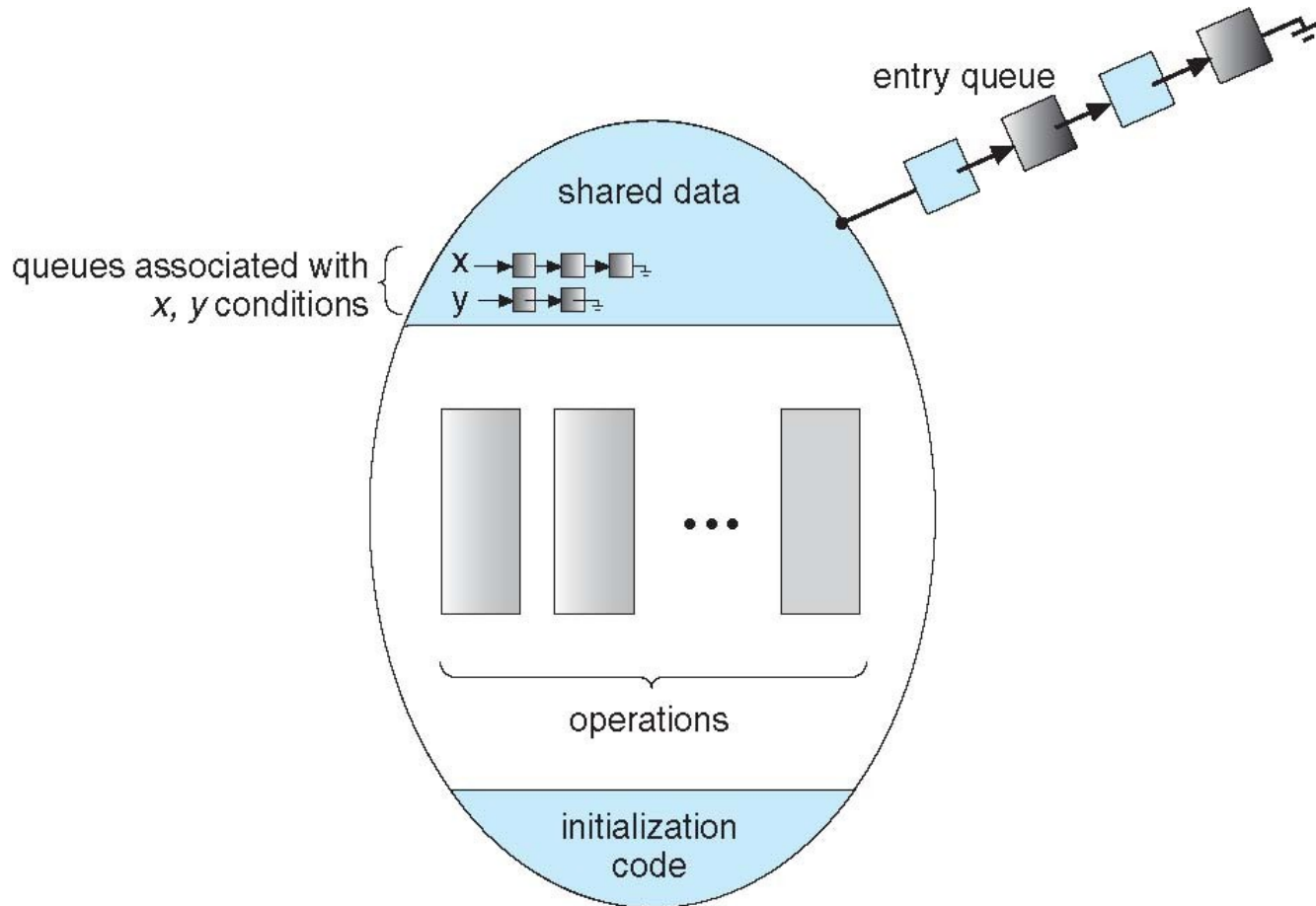


# Condition Variables

---

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - ▶ If no `x.wait()` on the variable, then it has no effect on the variable

# Monitor with Condition Variables





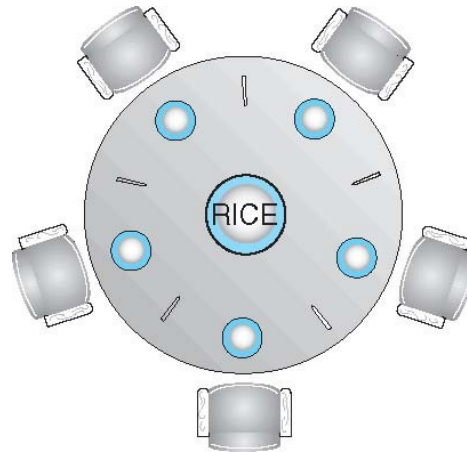
# Classical Problems of Synchronization

---

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

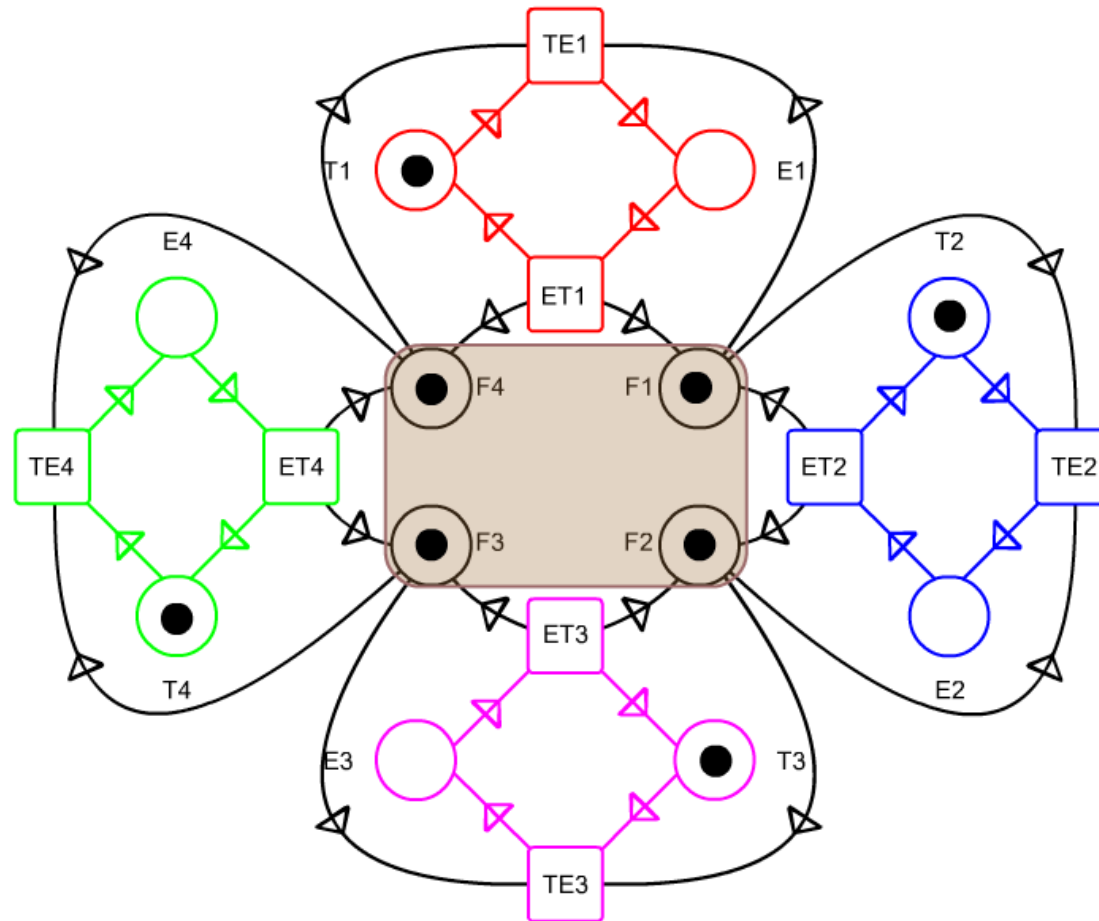
# Dining-Philosophers Problem

---



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - ▶ Bowl of rice (data set)
    - ▶ Semaphore **chopstick** [5] initialized to 1

# Dining-Philosophers Petri net



# DP Petri-net another Version

