# Process Synchronization

Why Process Synchronization

Critical Section problems and solutions

Semaphores and their problems

# Motivation

- Data sharing
  - Cooperating processes -- through files or messaging.
  - Threads -- directly share a logical address space
  - Concurrent access to shared data may result in data inconsistency
- Mechanisms required to ensure the orderly execution of cooperating processes or threads in order to maintain data consistency

# Critical Section

- **Critical section** is a segment of code, in which there are shared modifiable data that can be accessed and modified by multiple processes (or threads).
- Critical sections can be entered by processes only in non-overlapping intervals (**mutually exclusive** in time)
  - When one process is in its critical section, other process should not be in their critical sections.

# The Critical-Section Problem

- Problem: Regulate access to the critical section.
- Any solution must satisfy the following conditions:
  - **Mutual exclusion**
  - **Progress**
  - **Bounded waiting**

# Conditions for Critical-Section Solutions

- **Mutual exclusion**
  - If one process is executing in its critical section, then no other processes can be executing in their critical section
- **Progress**
  - If no process is executing in its critical section and some processes wish to enter their critical section, then one of the processes will be selected to enter its critical section.
  - This selection cannot be postponed indefinitely

# Conditions for Critical-Section Solutions

- **Bounded waiting:**
  - There is a **bound** on the number of times that other processes are allowed to enter their critical sections while a process is **waiting** (i.e. it has made a request to enter its critical section and that request has not been granted yet)
  - Purpose: Prevent starvation
    - **Starvation** – A process (thread) is perpetually denied access to some resources it requests.

# Solving Critical-Section Problems

## Generic solution

Request permission to enter → entry section

Notify about completion → exit section

```
while (true){

    entry section

        critical section

    exit section

        remainder section
}
```

Structure of a typical process that requires access to its critical section.

# Semaphores

- A **semaphore** is a mechanism provided by the system to implement **mutual exclusion**.
- A **semaphore** is
  1. a special integer variable **S** that,
  2. apart from initialization, is accessed only through two standard **atomic** operations: ***acquire()*** and ***release()***
  3. It is associated with a queue that stores the references to the processes that are waiting.

CM1205

# acquire() Operation

- **acquire()** ( originally termed **P()** ). It is called when a process wants to enter its critical section

```
acquire(S){
  S--;
  if (S<0)  {
     put this process into the waiting queue
     and block the process
   }
}
```

- It must be executed **indivisibly / atomically.**

# release() Operation

- **release()** ( originally termed **V()** ). It is called when a process exits its critical section

```
release(S){
  S++;
  if (S<=0) {
        remove a process from the waiting queue and
        wake up the process
  }
}
```

  - It must be executed **indivisibly / atomically.**

# Using a semaphore for mutual exclusion

- Mutual exclusion on a semaphore is enforced with **acquire()** and **release()**.

```
Semaphore S; // Shared among processes

while (true){

        acquire(S);          <- Request permission to enter

            critical section

        release(S);          <- Notify about completion

            remainder section
}
```

# Mutex Lock and Counting Semaphores

- **A mutex lock semaphore** is used to control access to the critical section for a process
  - Its value is …, -3, -2, -1, 0, 1. **At most 1.**
- A **counting semaphore** is used to control access to a given resource consisting a finite number of instances
  - Its value can range over an unrestricted domain

# Initializing a Semaphore

- A **mutex lock semaphore** is initialized to 1.
- A **counting semaphore** is initialized to the number of available instances of a given resource

# An Example

Semaphore S=1; // Shared variable initialization

S = 1

process P1

1 acquire(S);
2 Critical section;
4 release(S)

process P2

3 acquire(S);
5 Critical section;
6 release(S)

```
acquire(S){
  S--;
  if (S<0)  {
     put this process into queue and
     block the process
  }}
```

```
release(S){
  S++;
  if (S<=0) {
       remove a process from the waiting queue
       and wakeup the process
  }}
```

# Problems with Semaphores

- Use of semaphores may result in deadlock and starvation situations
  - **Deadlock**—One or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes
  - **Starvation**—Starvation occurs when a process is perpetually denied resource access. Without accessing the resources, the process is unable to complete its task.

# A Deadlock Example

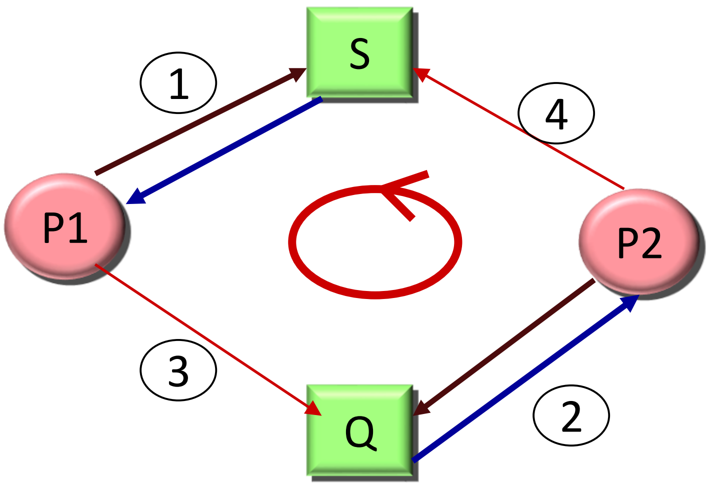
- Consider a system consisting of two processes P1 and P2, each accessing two semaphores, S and Q, set to the value 1:

```
process P1

acquire(S)
acquire(Q)
…
release(S)
release(Q)
```

```
process P2

acquire(Q)
acquire(S)
…
release(Q)
release(S)
```




Deadlocked!

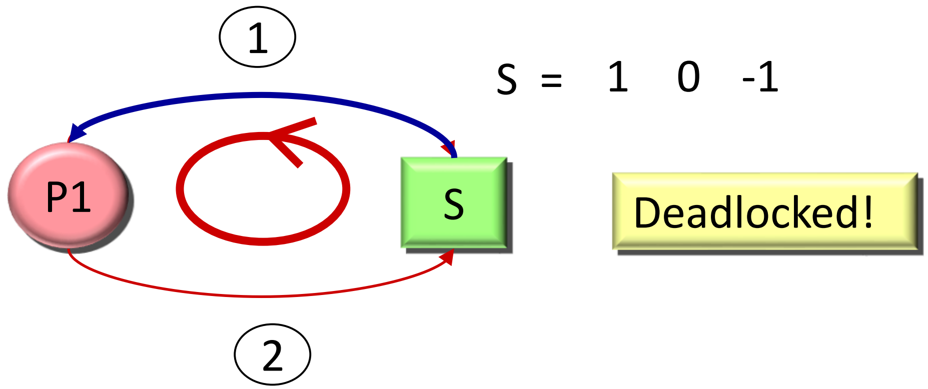
S = 1 0 -1

Q = 1 0 -1

# Another Deadlock Example

- Consider a system consisting of three processes P1, P2 and P3, each accessing a semaphore S which is set to the value 1

| process P1 | process P2 | process P3 |
|---|---|---|
| acquire(S); | acquire(S); | acquire(S); |
| … | … | … |
| acquire(S); | release(S); | release(S); |



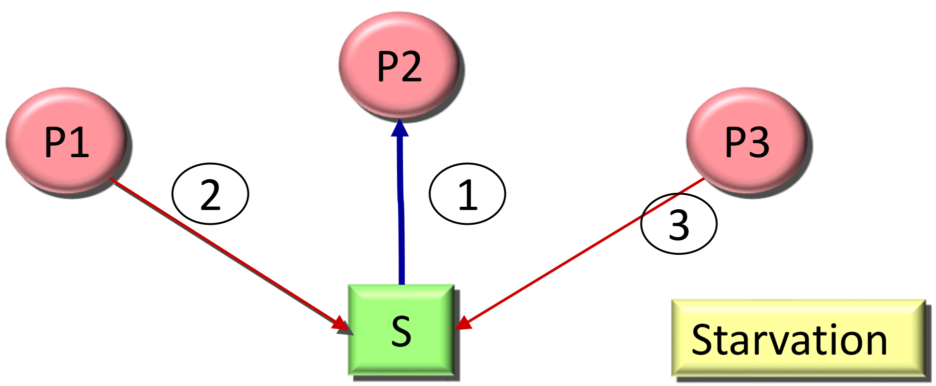
# A Starvation Example

- Consider a system consisting of three processes P1, P2 and P3, each accessing a semaphore S which is set to the value 1

```
process P1
acquire(S)
..
release(S);
```

```
process P2
acquire(S);
…
```

```
process P3
acquire(S)
…
release(S);
```

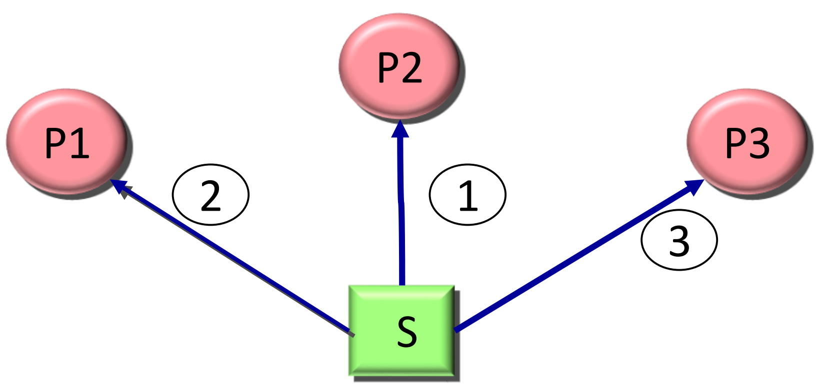



S = 1 0 -1 -2

# Examples of No Mutual Exclusion

- What happens if a process interchanges the order of acquire and release operations on a semaphore S that is set to 1?

| process P1 | process P2 | process P3 |
|---|---|---|
| release(S); | acquire(S); | acquire(S); |
| … | … | … |
| acquire(S); | release(S); | release(S); |




S = 1 0 1 0

# Summary

- Motivation
  - Bounded-Buffer Producer and Consumer Problem
- Critical-Section Problem
  - Solution must meet three conditions
    - Mutual exclusion
    - Progress
    - Bounded waiting
- Semaphores
  - Definition
  - Implementation: acquire() and release()
  - Problems with semaphores