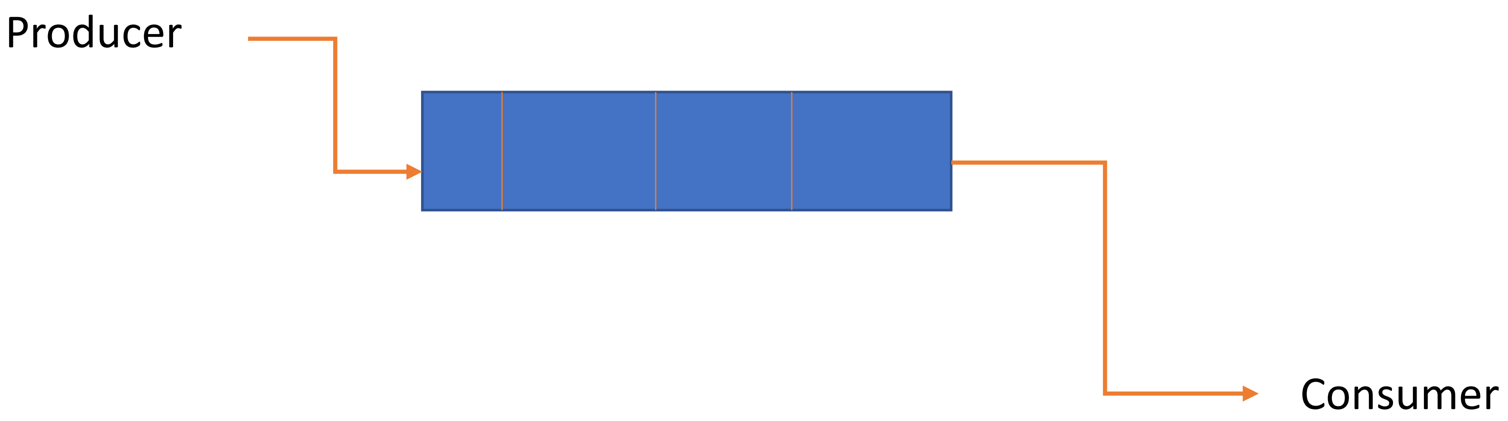
Process Synchronization

- Background
- The Critical-Section Problem
- Semaphores
- Classical Problems of Synchronization
- Monitors

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating/concurrent processes.
- That mechanism/logic is called process synchronization.



The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i

repeat

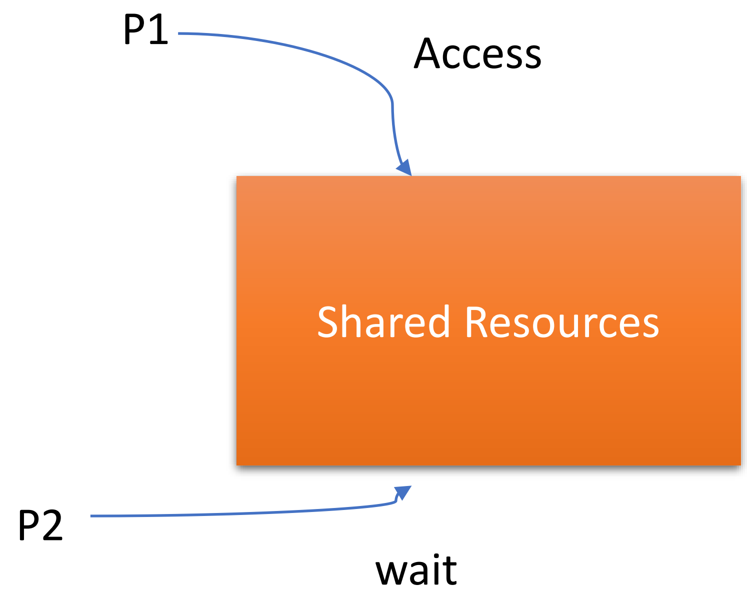
entry section

critical section

exit section

reminder section

until *false*;



Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

If one processing is accessing the shared data other should wait.

2. **Progress**. All the concurrent process involved in the synchronization (mutual exclusion) must progress and driven to completion.

3. **Bounded Waiting**. There should be definite waiting time for processes following mutual exclusion.

Semaphore

- It is the solution to critical section problem.
- Helps in achieving mutual exclusion.
- Semaphore S – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (S): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;

signal (S): $S := S + 1$;

Example: Critical Section of *n* Processes

- Shared variables
  - **var** *mutex* : *semaphore*
  - initially *mutex* = 1
- Process P_i

repeat

wait(*mutex*); 0: locked

$\boxed{\text{critical section}}$

signal(*mutex*); 1: opened

$\boxed{\text{remainder section}}$

until *false*;

Semaphore Implementation

- Define a semaphore as a record

  type *semaphore* = **record**

  value: *integer*

  L: **list of** *process*;

  end;

- Assume two simple operations:
  - Block: suspends the process that invokes it.
  - wakeup(*P*): resumes the execution of a blocked process P.

Implementation (Cont.)

- Semaphore operations now defined as

wait(*S*): *S.value* := *S.value* – 1;

if *S.value* < 0

then begin

add this process to *S*.L;

*blo*ck;

end;

signal(*S*): *S.value* := *S.value* + 1;

if *S.value* $\leq$ 0

then begin

remove a process *P* from *S.L*;

wakeup(*P*);

end;

Semaphore as General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
$\vdots$	$\vdots$
A	*wait*(*flag*)
signal(*flag*)	B

B can not be executed until unless process Pi execute the signal(flag) instruction

wait (*S*): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;
signal (*S*): $S := S + 1$;

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
wait(S);	*wait*(Q);
wait(Q);	*wait*(S);
⋮	⋮
signal(S);	*signal*(Q);
signal(Q)	*signal*(S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.