Background

- Concurrent access to shared data may result in data inconsistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
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 \( P_i \)

\[
\text{repeat} \\
\text{entry section} \\
\text{critical section} \\
\text{exit section} \\
\text{reminder section} \\
\text{until false;}
\]
Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

\[
\text{ wait } (S): \text{ while } S \leq 0 \text{ do no-op; } \\
S := S - 1; \\
\]

\[
\text{ signal } (S): S := S + 1; \\
\]
Example: Critical Section of $n$ Processes

- **Shared variables**
  - `var mutex : semaphore`
  - initially $mutex = 1$

- **Process $P_i$**

```
repeat
  wait(mutex);
  critical section
  signal(mutex);
  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.
  - \texttt{wakeup}(P) resumes the execution of a blocked process \texttt{P}. 

Implementation (Cont.)

- Semaphore operations now defined as

  \[ \text{wait}(S): \quad S.value := S.value - 1; \]
  \[ \text{if } S.value < 0 \]
  \[ \text{then begin} \]
  \[ \text{add this process to } S.L; \]
  \[ \text{block}; \]
  \[ \text{end}; \]
  \[ \text{signal}(S): \quad S.value := S.value = 1; \]
  \[ \text{if } S.value \leq 0 \]
  \[ \text{then begin} \]
  \[ \text{remove a process } P \text{ from } S.L; \]
  \[ \text{wakeup}(P); \]
  \[ \text{end}; \]
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
:  
A
wait(flag)
signal(flag)
B
```
• Locks provide mutual exclusion
• Locks have two operations:
  – Acquire()
  – Release()
• Only one process can hold a lock at a time
  – If a lock is free, the first process to acquire it is said to hold the lock
  – All subsequent processes are blocked until the first one does a release operation
  – Then one process acquires it and the rest continue waiting
• Locks are a lot like semaphores, but only the holder of the lock can release it
Monitors: Condition Variables

- To allow a process to wait within the lock (as opposed to waiting for the lock), a *condition* variable must be used.

- Condition variables support three operations:
  - **Wait**
    - Releases the lock
    - Blocks the process until a signal or broadcast is done
    - Reacquires the lock
  - **Signal**
    - Wakes up one waiting process
  - **Broadcast**
    - Wakes up all waiting processes
Bounded Buffer: Producer

- Given a Lock L and Conditions empty (initially 1) and full (initially 0)

`<produce>` // Produce the data

`Acquire(&L);` // Get the lock

`Wait(&empty);` // Wait for the shared buffer to be empty

`<put data in shared buffer>`

`Signal(&full);` // Wake up the consumer

`Release(&L);` // Release the lock
Bounded Buffer: Consumer

- Given a Lock L and Conditions empty and full

```c
Acquire(&L);  // Get the lock
Wait(&full);  // Wait for the buffer to be full
<get data from shared buffer>
Signal(&empty);  // Signal that the buffer is empty
Release(&L);  // Release the lock
```