Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000
Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.
The Critical-Section Problem

- \( n \) processes all competing to use some shared data
- Each process has a code segment, called \textit{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.
Solution to Critical-Section Problem

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.
Initial Attempts to Solve Problem

- Only 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (other process $P_j$)

\[
\text{do } \{ \\
\text{entry section} \\
\text{critical section} \\
\text{exit section} \\
\text{reminder section} \\
\} \text{ while (1);} \\
\]

- Processes may share some common variables to synchronize their actions.
Algorithm 1

- **Shared variables:**
  - `int turn;` 
    initially `turn = 0`
  - `turn = i` ⇒ $P_i$ can enter its critical section

- **Process $P_i$**

  ```
  do {
    Loop while (turn != i) no-op
    critical section
    turn = j;
    reminder section
  }
  ```

- Satisfies mutual exclusion, but not progress
Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes $P_i$ and $P_j$ receive the same number, if $i < j$, then $P_i$ is served first; else $P_j$ is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 4, 5...
Synchronization Hardware

- Test and modify the content of a word atomically
  
  ```c
  boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
  }
  ```
Mutual Exclusion with Test-and-Set

- Shared data:
  ```java
  boolean lock = false;
  ```

- Process $P_i$
  ```java
  do {
    while (TestAndSet(lock)) ;
    critical section
    lock = false;
    remainder section
  }
  ```
Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations
  
  \[ \text{wait} \ (S): \]
  
  \[ \text{while } S \leq 0 \text{ do } \text{no-op}; \]
  \[ S--; \]

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

- Shared data:
  
  \[
  \text{semaphore } \text{mutex}; \\
  \text{initially } \text{mutex} = 1
  \]

- Process $P_i$:
  
  \[
  \text{wait} (\text{mutex}); \\
  \text{critical section} \\
  \text{signal} (\text{mutex}); \\
  \text{remainder section}
  \]
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

  \[
  \begin{align*}
  P_0 & \quad P_1 \\
  \text{wait}(S); & \quad \text{wait}(Q); \\
  \text{wait}(Q); & \quad \text{wait}(S); \\
  \vdots & \quad \vdots \\
  \text{signal}(S); & \quad \text{signal}(Q); \\
  \text{signal}(Q) & \quad \text{signal}(S);
  \end{align*}
  \]

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Two 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.
- Can implement a counting semaphore $S$ as a binary semaphore.
Critical Regions

- High-level synchronization construct
- A shared variable \( v \) of type \( T \), is declared as: \( v: \text{shared} T \)
- 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 \( v \).
Critical Regions

- 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

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

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

End of Section