Control Synchronization and Indivisible Operations

- Interacting processes need to coordinate their execution with respect to one another, to perform their actions in a desired order
  - Requirement met through control synchronization

\[
\begin{align*}
\text{Process } P_i & \quad \text{Performs operation } a_j \\
\text{Process } P_j & \quad \text{Performs operation } a_i
\end{align*}
\]

\textbf{Figure 6.5} Processes requiring control synchronization.

- \textit{Signaling} is a general technique of control synchronization

\begin{verbatim}
var
  operation_{aj}_performed : boolean;
  pi_blocked : boolean;

begin
  operation_{aj}_performed := false;
  pi_blocked := false;

Parbegin
  . . .
  if \text{operation}_{aj}_performed = false
    then
      pi_blocked := true;
      block \langle P_j \rangle;
      \text{perform operation}_{aj} \}
  . . .
  . . .
Parend;

end.

\end{verbatim}

\textbf{Figure 6.6} A naive attempt at signaling through boolean variables.
Control Synchronization and Indivisible Operations (continued)

- Naive signaling in previous example does not work
  - $P_i$ may face indefinite blocking in some situations

Table 6.2 Race Condition in Process Synchronization

<table>
<thead>
<tr>
<th>Time</th>
<th>Actions of process $P_i$</th>
<th>Actions of process $P_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>if action$_i$.performed = false</td>
<td></td>
</tr>
<tr>
<td>$t_2$</td>
<td>perform action$_i$</td>
<td></td>
</tr>
<tr>
<td>$t_3$</td>
<td>if piBlocked = true</td>
<td>action$_i$.performed :=true</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_{20}$</td>
<td>piBlocked := true;</td>
<td></td>
</tr>
<tr>
<td>$t_{21}$</td>
<td>block ($P_j$)</td>
<td></td>
</tr>
</tbody>
</table>

- Use *indivisible* or *atomic operations* instead

---

Control Synchronization and Indivisible Operations (continued)

**Definition 6.4 Indivisible Operation**

An operation on a set of data items that cannot be executed concurrently either with itself or with any other operation on a data item included in the set.

```plaintext
procedure check$_{aj}$
begin
  if operation$_{ai}$.performed=false
  then
    piBlocked:=true;
    block($P_j$)
  end
end

procedure post$_{aj}$
begin
  if piBlocked=true
  then
    piBlocked:=false;
    activate($P_j$)
  else
    operation$_{ai}$.performed:=true;
  end
end
```

Figure 6.7 Indivisible operations check$_{aj}$ and post$_{aj}$ for signaling.
Synchronization Approaches

- Looping versus Blocking
- Hardware Support for Process Synchronization
- Algorithmic Approaches, Synchronization Primitives, and Concurrent Programming Constructs

6.5

Looping versus Blocking

- **Busy wait:**
  
  ```
  while (some process is in a critical section on \{d_i\} or is executing an indivisible operation using \{d_i\})
  { do nothing }
  ```

  ![Critical section or indivisible operation using \{d_i\}]

- A busy wait has many consequences
  - Cannot provide the bounded wait property
  - System performance degradation due to looping
  - Processes may wait for each other indefinitely

6.6
Looping versus Blocking (continued)

• To avoid busy waits, a process waiting for entry to a CS is put in blocked state
  – Changed to ready only when it can enter the CS
    if (some process is in a critical section on \(d_c\) or
    is executing an indivisible operation using \(d_i\))
    then make a system call to block itself;

• Process decides to loop or block
  – Decision is subject to race conditions. Avoided through
    • Algorithmic approach
    • Use of computer hardware features

Hardware Support for Process Synchronization

• Indivisible instructions
  – Avoid race conditions on memory locations
• Used with a lock variable to implement CS and indivisible operations

Figure 6.3 Implementing a critical section or indivisible operation by using a lock variable.
– entry_test performed with indivisible instruction
  • Test-and-set (TS) instruction
  • Swap instruction
Hardware Support for Process Synchronization (continued)

- **Algorithmic Approaches**
  - For implementing mutual exclusion
  - Independent of hardware or software platform
    - Busy waiting for synchronization

- **Synchronization Primitives**
  - Implemented using indivisible instructions
  - E.g., *wait* and *signal* of semaphores
    - Problem: can be used haphazardly

- **Concurrent Programming Constructs**
  - *Monitors*