Process Synchronization

Prepared By:
Saed Swedan
Omar Hirzallah

Supervised By: Dr. Lo’ai Tawalbeh
Jordan University of Science and Technology
2006
Agenda

- Task Synchronization
- Critical Section
- Semaphores
- Real-Time System Issues and Solutions.
A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes often access shared data, which may lead to inconsistent data.

Maintaining data consistency requires synchronization mechanisms.
Producer process:

item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

Producer – Consumer Module
Producer – Consumer Module

- Consumer process:
  item nextConsumed;

  while (1) {
    while (counter == 0)
      ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
  }
Example

- The Statements:
  
  ```
  counter++;  
  counter--;  
  ```

  must be executed atomically.

- Atomic operation means an operation that completes in its entirety without interruption.
Example Cont.

- Suppose *Counter++* is implemented as follows:
  
  \[
  \text{register}_1 = \text{counter} \\
  \text{register}_1 = \text{register}_1 + 1 \\
  \text{counter} = \text{register}_1
  \]

- And suppose *Counter--* is implemented as follows:
  
  \[
  \text{register}_2 = \text{counter} \\
  \text{register}_2 = \text{register}_2 - 1 \\
  \text{counter} = \text{register}_2
  \]
Example Cont.

- Concurrent execution of `counter++` and `counter--` may be interleaved as follows (initially `counter = 5`):

  \[
  \begin{align*}
  T_0 &: \text{register}_1 = \text{counter} & \{\text{register}_1 = 5\} \\
  T_1 &: \text{register}_1 = \text{register}_1 + 1 & \{\text{register}_1 = 6\} \\
  T_2 &: \text{register}_2 = \text{counter} & \{\text{register}_2 = 5\} \\
  T_3 &: \text{register}_2 = \text{register}_2 - 1 & \{\text{register}_2 = 4\} \\
  T_4 &: \text{counter} = \text{register}_1 & \{\text{counter} = 6\} \\
  T_5 &: \text{counter} = \text{register}_2 & \{\text{counter} = 4\}
  \end{align*}
  \]

- Which leaves `counter` at an incorrect value
Critical Section

- A Critical Section is a piece of code that accesses a shared resource (data structure or device).

- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
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:** No process has to wait indefinitely to access the critical section... 😊
Initial Attempts to Solve Problem

- Only 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (other process $P_j$)
  
  \begin{verbatim}
  do {
    entry section
    critical section
    exit section
    reminder section
  } while (1);
  \end{verbatim}

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

- Shared variables:
  - int turn;
    - initially turn = 0
  - turn = i \Rightarrow P_i \text{ can enter its critical section}
- Process $P_i$
  
  \begin{verbatim}
  do {
      while (turn != i) ;
      critical section
      turn = j;
      reminder section
  } while (1);
  \end{verbatim}
- Satisfies mutual exclusion, but not progress
Algorithm 2

- Shared variables
  - boolean flag[2];
    initially flag[0] = flag[1] = false.
  - flag[i] = true ⇒ $P_i$ ready to enter its critical section
- Process $P_i$
  do {
    flag[i] := true;
    while (flag[j]) ;
    critical section
    flag[i] = false;
    remainder section
  } while (1);
- Satisfies mutual exclusion, but not progress requirement.
Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process $P_i$
  
  \[
  \text{do } \{
  \text{flag } [i] := \text{true};
  \text{turn } = j;
  \text{while (flag } [j] \text{ and turn } = j) ;
  \text{critical section}
  \text{flag } [i] = \text{false};
  \text{remainder section}
  \} \text{ while (1);} 
  \]

- Meets all three requirements; solves the critical-section problem for two processes.
A Semaphore is an integer variable that can be only accessed by two atomic operations: \textit{wait}, and \textit{signal}.

\begin{itemize}
  \item \textbf{Wait(S)} \\
        \{ \\
          \text{While (S \leq 0); //no-op} \\
          S--; \\
        \}
  \item \textbf{Signal(S)} \{S++;\}
\end{itemize}
Example 1: Synchronize access to a critical section

- We have two processes $P_1$ and $P_2$ that share a Semaphore `mutex` initialized to 1.
- $P_1$ and $P_2$ execute the following code:
  ```c
  while(1)
  {
    wait (mutex);
    critical section
    signal (mutex);
    remainder section
  }
  ```
Example 2: Synchronize process execution

- We have two processes $P_1$ and $P_2$ that share a Semaphore $\text{synch}$ initialized to 0.
- We want $P_1$ to execute $S_1$ only after $P_2$ executes $S_2$.
- $P_1$ code:
  
  ```
  wait (synch)
  S_1;
  S_1;
  ```
- $P_2$ code:
  
  ```
  S_2;
  signal (synch);
  ```
Semaphore Implementation

- Previous Implementation wastes CPU cycles on waiting processes.
- A better implementation can be achieved as follows:
  - Each semaphore has an Integer `val` and a waiting list `L`.
  - We have 2 extra operations `Block(P)` and `Wakeup(P)`.
  - `Block(P)`: block process P.
  - `Wakeup(P)`: let process P continue executing.
Now Wait(S) looks like this:

```c
Wait(S)
{
    S.val--;  
    if (S.val < 0)  
    {
        add process P to S.L; //add to waiting list
        Block(P);
    }
}
```
Semaphore Implementation
Cont.

- Signal (S) looks like this:
  ```
  Signal(S)
  {
    S.val++;    //are there are blocked processes
    if (S.val ≤ 0) {
      remove process P from S.L;
      Wakeup(P);
    }
  }
  ```
Problems with Semaphores

- If semaphores are used incorrectly in the program it can lead to timing errors.

- These errors can be difficult to detect and correct, because they occur only occasionally and only under certain circumstances.
Examples

- Interchange signal and wait.
  - signal(mutex);
  - critical section
  - wait(mutex);

- Replace signal with wait:
  - wait(mutex);
  - critical section
  - wait(mutex);
Examples Cont.

- Omit the wait.
  - ...
  - critical section
  - signal(mutex);

- Omit the signal:
  - wait(mutex);
  - critical section
  - ...

Metrics for real-time systems differ from that for time-sharing systems.

<table>
<thead>
<tr>
<th></th>
<th>Time-Sharing Systems</th>
<th>Real-Time Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>High throughput</td>
<td>Schedulability</td>
</tr>
<tr>
<td>Overload</td>
<td>Fairness</td>
<td>Stability</td>
</tr>
</tbody>
</table>

- **schedulability** is the ability of tasks to meet all hard deadlines
- **stability** in overload means the system meets critical deadlines even if all deadlines cannot be met
In real-time systems using a FIFO queue for process waiting-lists and message queues is not practical.

Real-time systems use either Earliest Dead-Line First or Highest Priority First ordering policy.

If a higher priority thread wants to enter the critical section while a lower priority thread is in the Critical Section, it must wait for the lower priority thread to complete, this is called **priority inversion**.
A higher priority process waiting on a lower priority process is usually acceptable because critical sections normally have a few instructions.

A problem occurs when a process with a medium priority wants to run (not in the critical section), and reserves the CPU until it’s finished, which leads to unacceptable delays.
Real-Time System Issues Cont.

- To illustrate an extreme example of priority inversion, consider the executions of four periodic threads: A, B, C, and D; two resources (synchronized objects): Q and V

<table>
<thead>
<tr>
<th>thread</th>
<th>Priority</th>
<th>Execution Sequence</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>EQQQQQE</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>EE</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>EVVE</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>EEQVE</td>
<td>4</td>
</tr>
</tbody>
</table>

- Where E is executing for one time unit, Q is accessing resource Q for one time unit, V is accessing resource V for one time unit
This problem can be solved mainly in two ways:

1- Priority Inheritance:

- Let the lower priority task \( A \) use the highest priority of the higher priority tasks it blocks. In this way, the medium priority tasks can no longer preempt low priority task \( A \), which has blocked the higher priority tasks.
Priority Inheritance

- If thread $p$ is blocked by thread $q$, then $q$ runs with $p$'s priority
2- Priority Ceiling:

- A priority ceiling is assigned to each mutex, which is equal to the highest priority task that may use this mutex.
- A task can lock a mutex if and only if its priority is higher than the priority ceilings of all mutexes locked by other tasks.
- If a task is blocked by a lower priority task, the lower priority task inherits its priority.
Priority Ceiling

thread

D

C

B

A

0  2  4  6  8  10  12  14  16  18
References

- “Real-Time Mach: Towards a Predictable Real-Time System”, Tokuda, Nakajima, and Rao
- “RT-IPC: An IPC Extension for Real-Time Mach”, Kitayama, Nakajima, and Tokuda
- [http://www.intel.com/education/highered/Embedded/lectures/19_RealTime_Synchronization.ppt](http://www.intel.com/education/highered/Embedded/lectures/19_RealTime_Synchronization.ppt)
- [http://www.ics.uci.edu/~bic/courses/143/LEC/ch5.ppt](http://www.ics.uci.edu/~bic/courses/143/LEC/ch5.ppt)
- [http://tulip.bu.ac.th/~thanakorn.w/Course/cs420/cs420.files/mod6_2.pdf](http://tulip.bu.ac.th/~thanakorn.w/Course/cs420/cs420.files/mod6_2.pdf)
- [http://mathcs.holycross.edu/~croyden/os/notes/Lec24_critical_regions.ppt](http://mathcs.holycross.edu/~croyden/os/notes/Lec24_critical_regions.ppt)