Semaphores

Definition 6.5 Semaphores: A shared integer variable with nonnegative values that can be subjected only to the following operations:
1. Initialization (specified as part of its declaration)
2. The indivisible operations wait and signal

\begin{verbatim}
procedure wait (S)
begin
  if S > 0
    then S := S-1;
    else block the process on S;
  end:

procedure signal (S)
begin
  if some processes are blocked on S
    then activate one blocked process;
    else S := S+1;
  end:
\end{verbatim}

Figure 6.22 Semantics of the wait and signal operations on a semaphore.

• Also called counting semaphores
• Indivisibility of the wait and signal operations ensures that race conditions cannot arise over a semaphore

Uses of Semaphores in Concurrent Systems

Table 6.3 Uses of Semaphores in Implementing Concurrent Systems

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Mutual exclusion can be implemented by using a semaphore that is initialized to 1. A process performs a wait operation on the semaphore before entering a CS and a signal operation on exiting from it. A special kind of semaphore called a binary semaphore further simplifies CS implementation.</td>
</tr>
<tr>
<td>Bounded concurrency</td>
<td>Bounded concurrency implies that a function may be executed, to a resource, by a process concurrently, (1 \leq n \leq c), where (c) is a constant. A semaphore initialized to (c) can be used to implement bounded concurrency.</td>
</tr>
<tr>
<td>Signaling</td>
<td>Signaling is used when a process (P_i) wishes to perform an operation (a_j) only after process (P_j) has performed an operation (a_j). It is implemented by using a semaphore initialized to 0. (P_i) performs a wait on the semaphore before performing operation (a_j). (P_j) performs a signal on the semaphore after it performs operation (a_j).</td>
</tr>
</tbody>
</table>
Uses: Mutual Exclusion

\begin{verbatim}
var sem_CS : semaphore := 1;
Par begin
    repeat
        wait (sem_CS);
        { Critical Section }
        signal (sem_CS);
        { Remainder of the cycle }
    forever;
End.
\end{verbatim}

\textbf{Figure 6.23} CS implementation with semaphores.

- Above implementation possesses \textit{mutual exclusion} and \textit{progress} properties, but violates \textit{bounded wait}
- \textit{Binary semaphore}: a special kind of semaphore used for implementing mutual exclusion, often called a \textit{mutex}
  - Initialized to 1; can have values 0 and 1 only

Uses: Bounded Concurrency

- Up to $c$ (a constant $\geq 1$) processes can concurrently perform $op_i$
- Implemented by initializing a semaphore $sem_c$ to $c$
- Every process wishing to perform $op_i$ performs a $wait(sem_c)$ before performing $op_i$ and a $signal(sem_c)$ after performing it
Uses: Signaling between Processes

- Process \( P_i \) should perform an operation \( a_i \) only after process \( P_j \) performs an operation \( a_j \).

\[
\text{var } \text{sync : semaphore := 0}; \\
\text{Parbegin} \\
\quad \ldots \\
\quad \text{wait (sync)}; \\
\quad \{ \text{Perform action } a_j \} \\
\quad \text{signal (sync)}; \\
\text{end.}
\]

Figure 6.25 Signaling using semaphores.

Producers-Consumers Using Semaphores

- Total concurrency in system is 1

\[
\text{type } \text{item} = \ldots ; \\
\text{var} \\
\quad \text{full : Semaphore := 0; } \{ \text{Initializations} \} \\
\quad \text{empty : Semaphore := 1; } \\
\quad \text{buffer : array [0] of item;} \\
\text{begin} \\
\text{Parbegin} \\
\quad \text{repeat} \\
\quad \text{wait (empty);} \\
\quad \{ \text{if full \( \Rightarrow \) empty} \} \\
\quad \text{buffer [0] := \ldots ;} \\
\quad \{ \text{i.e., produce} \} \\
\quad \text{signal (full);} \\
\quad \{ \text{remainder of the cycle} \} \\
\text{end}, \\
\text{Repeat} \\
\text{forever}, \\
\text{Producer} \\
\text{Parbegin} \\
\text{Repeat} \\
\text{forever}, \\
\text{Consumer}
\]

Figure 6.26 Producers-consumers with a single buffer.
Producers-Consumers Using Semaphores (continued)

- Solution to $n$-buffer producers–consumers problem:

```c
const n = ...;
type item = ...;
var buffer : array [0..n-1] of item;
full : Semaphore := 0;  // Initializations
empty : Semaphore := n;
prod_ptr, cons_ptr : integer;

begin
  prod_ptr := 0;
  cons_ptr := 0;

Parbegin
  repeat
    wait (empty);
    prod_ptr := prod_ptr + 1 mod n;
    signal (full);
  forever;
end.

Producer

Consumer
```

Buffers are consumed in FIFO order

Readers-Writers Using Semaphores

- Significance of counters:
  - runread: Number of readers reading
  - totread: Number of readers wishing to read or reading
  - Similarly runwrite and totwrite

Figure 6.28 Bounded buffers using semaphores.

Figure 6.29 Refined solution outline for readers–writers.
Implementation of Semaphores

Type declaration for Semaphore

```plaintext
type
  semaphore = record
    value : integer;  { value of the semaphore }
    list : list;      { list of blocked processes }
    lock : boolean;  { lock variable for operations on this semaphore }
end;
```

Procedures for implementing `wait` and `signal` operations

```plaintext
procedure wait (sem) begin
  Close_lock (sem.lock);
  if sem.value > 0 then
    sem.value := sem.value - 1;
    Open_lock (sem.lock);
  else
    Add id of the process to list of processes blocked on sem;
    block_proc (sem.lock);
end;

procedure signal (sem) begin
  Close_lock (sem.lock);
  if some processes are blocked on sem then
    proc_id := id of a process blocked on sem;
    awake (proc_id);
  else
    sem.value := sem.value + 1;
    Open_lock (sem.lock);
end;
```

Implementations of `wait` and `signal`:
- Kernel-Level: Through system calls
- User-Level: Through library procedures
- Hybrid: Combination of the two