Lecture 8

Producer
Consumer
Problem
Semaphores as Counters

In this Lecture we show how to solve more general versions of the Producer Consumer problem, by considering integer semaphores.

- **Wait** – Decrement count
- **Signal** – Increment count
- **Init** - initialise count

Semaphore Initialisation

So far our semaphores have always been initialised to 1.

On occasions we may need them to be zeroed ie initially "unavailable".

A further method Init is often used - or a Constructor.
Initialisable Semaphore

```java
public class Semaphore {
    private int count; // semaphore value

    public Semaphore(int init) {
        count = init;
    }

    public void Wait() {
        ... details omitted ...
    }

    public void Signal() {
        ... details omitted ...
    }

    public void Start() {
    }
}
```
Producer Consumer Revisited

In this problem two tasks; Producer and Consumer inter-communicate messages via a common buffer.

The buffer allows the producer to get ahead of the Consumer. **NB** There are four messages in the above buffer. The primitives; put and get respectively write to and read from the buffer.
Produce Consumer …

The primitives produce and consume, are the birth and destruction events for the messages. The problem is to enable the two tasks to run as quickly as possible without unnecessary delay.

Producer Body

while true do
    produce(m);
    put(m, buffer);
end while

Consumer Body

while true do
    get(m, buffer);
    consume(m);
end while

There is however, a safety problem with the above system.
Safety Issues

It is considered erroneous for the Consumer to access an empty buffer and messages should only be consumed exactly once.

We use a semaphore to permit the Producer to signal the availability of messages.

Semaphores provide an “integrated count”.

If the Producer gets ahead of the Consumer then \( S \) increases beyond one.

**NB** this could not happen in the mutual exclusion problem since at most one signal can increment the semaphore.
P/C Solution

Semaphore $S = \textbf{new} \text{ Semaphore}(0)$;

Producer : \textbf{task} message m ;
  \hspace{1em} \textbf{while} true \textbf{do}
  \hspace{1em} produce(m) ;
  \hspace{1em} put(m,buffer) ;
  \hspace{1em} signal(S);
  \hspace{1em} \textbf{end while}
\textbf{end task}

Consumer : \textbf{task} message m ;
  \hspace{1em} \textbf{while} true \textbf{do}
  \hspace{1em} wait(S);
  \hspace{1em} get(m,buffer) ;
  \hspace{1em} consume(m) ;
  \hspace{1em} \textbf{end while}
\textbf{end task}
Key Points

• It is important to realise that in the Producer Consumer Solution the process that does the signalling is different from the one that does the waiting.

• In the (simple) Mutual Exclusion Solution the process that first waits successfully is the same one that releases the semaphore by signalling it.

• Both are valid, yet distinct approaches.

NB

As a concurrent programmer it is important to know when to use which idiom!
Buffer Types

Our solution assumes an unbounded buffer, that is, the Producer can get ahead of the Consumer by an infinite number of signals.

In the more realistic bounded buffer problem we assume that there are a maximum of say ten slots only in the buffer.

Semaphore $S = \text{new Semaphore}(0)$;
Semaphore $\text{SPACES} = \text{new Semaphore}(10)$;

Producer : task message $m$ ;
   while true do
      produce($m$) ;
      wait(\text{SPACES}) ;
      put($m$,buffer) ;
      signal($S$) ;
   end while
end task
Bounded Buffer

Consumer: task message m;
    while true do
        wait(S);
        get(m,buffer);
        signal(SPACES);
        consume(m);
    end while
end task

The two co-operating tasks guarantee that the buffer has less than ten items if a put is to take place, and has at least one, if a get is to take place.
Bounded Buffer Solution

Semaphore S = new Semaphore(0);
Semaphore SPACES = new Semaphore(10);

Producer : task message m ;
    while true do
        produce(m) ;
        wait(SPACES);
        put(m,buffer) ;
        signal(S);
    end while
end task

Consumer : task message m ;
    while true do
        wait(S);
        get(m,buffer) ;
        signal(SPACES);
        consume(m) ;
    end while
end task
Some Simple Scenarios

In order to understand the solution we look at some simple scenarios. Uneven “bursts” are a good way of understanding the code.

Scenario 1
- Maximum Production Burst
- Maximum Consumption Burst

The Producer rapidly creates ten items but then SPACES hits zero and it then has to wait.

At this point S has reached ten

The Consumer now rapidly disposes of ten items, It can’t get disposes of any more items because S is now zero [and of course the buffer is empty]

Scenario 2
As above, but thirdly, a Production burst.
Some Simple Scenarios ...

Scenario 2
- Maximum Production Burst
- Maximum Consumption Burst
- Maximum Production Burst

Question
How do we know for the second burst that no more than ten new items can be produced?

Answer
After the Consumption burst SPACES has been counted back up to ten.

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>SPACES</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>P Burst</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>C Burst</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>P Burst</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
More Simple Scenarios

Let us consider a partial filling of the buffer.

Scenario 3
- 3 Production Burst
- 3 Consumption Burst

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>SPACES</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>P Burst</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>C Burst</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

In general, the system maintains the invariant

\[ \#\text{SPACES} + \#S = 10 \]

If we decrease the spaces this must manifest itself as “items” that are available for the consumer. When items are consumed “spaces” are returned.
Binary Semaphore

Sometimes coders use the “test and set” approach so we only have a 0/1 semaphore but this is easily extended to “protect” local counts [or queues] so that we still achieve the general semaphore solution as presented.

Put and Get

We haven’t yet looked at these when say the physical length of the buffer is much larger than the logical length.

Often modulo arithmetic is used for the put and the get so that the "item block” wraps around the physical end of the buffer.

Exercise

Use a third semaphore CRIT to protect the puts and gets which now must take place under mutual exclusion.
Partial Order

A directed graph in which there are no directed cycles is called a partial order.

Example

A Partial Order of \{a,b,c,d,e,f\}

They are extremely useful in start-up initialisations of industrial processes.

Above we express that c and d can only start-up providing a has . f can only start-up providing both d and b have. Finally e can only start-up providing both c and d have.
PO and Semaphores

Suppose a, b, c, … f are tasks and we want to ensure that initialisations of "higher" tasks in the partial order have occurred before "lower" tasks start their initialisations, then we simply associate a semaphore with each edge(arc) of the po.

Each "higher" task needs to signal its outgoing semaphores and each "lower" task needs to wait on its incoming semaphores.
Complex Start-Up

Example

Zero each of $S_1, S_2, \ldots, S_6$

{  
  task init_a ; signal(S_1) ; signal(S_2);
      rest_of_a; end task
  task init_b ; signal(S_3);
      rest_of_b; end task
  task wait(S_1); init_c; signal(S_4);
      rest_of_c; end task
  task wait(S_2); init_d; signal(S_5); signal(S_6);
      rest_of_d; end task
  task wait(S_4); wait(S_5); init_e ;
      rest_of_e; end task
  task wait(S_3); wait(S_6); init_f;
      rest_of_f; end task
  }

CPICS Lecture 8
# C# Semaphore Family

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Type</th>
<th>Owner</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor.Enter()</td>
<td>Binary</td>
<td>Single</td>
<td>No</td>
</tr>
<tr>
<td>Monitor.Exit()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock()</td>
<td>Binary</td>
<td>Single</td>
<td>Yes</td>
</tr>
<tr>
<td>{   ...     }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutex</td>
<td>Binary</td>
<td>Single</td>
<td>Yes</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Integer</td>
<td>Multi</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Index

0  Front Page
1  Semaphores as Counters
2  Initialisable Semaphore
3  Producer Consumer Revisited
4  Producer Consumer ...
5  Safety Issue
6  P/C Solution
7  Key Points
8  Buffer Types
9  Bounded Buffer
Index

10  Bounded Buffer Solution
11  Some Simple Scenarios
12  Some Simple Scenarios ...
13  More Simple Scenarios
14  Binary Semaphore
15  Partial Order
16  PO and semaphores
17  Complex Start-Up
18  C# Semaphore Family