Semaphores

Johannes Åman Pohjola
CSIRO's Data61 (and UNSW)
Term 2 2021
In the last lecture, we saw more about hardware-assisted critical sections and how they are used to implement a basic unit of synchronisation, called a *lock* or *mutex*.

In this lecture, we will generalise this concept to a *semaphore* with a particular analysis of the *producer consumer problem*. 

**Where we are at**
Definition

A *semaphore* can be abstractly viewed as a pair \((v, L)\) of a natural number \(v\) and a set of processes \(L\). The semaphore must always be initialised to some \((v, \emptyset)\).
Semaphores

Definition

A *semaphore* can be abstractly viewed as a pair $(v, L)$ of a natural number $v$ and a set of processes $L$. The semaphore must always be initialised to some $(v, \emptyset)$.

- $v$ : how many more processes we can let in without waiting.
- $L$ : the processes currently waiting to get in.
Semaphores

Definition

A *semaphore* can be abstractly viewed as a pair \((v, L)\) of a natural number \(v\) and a set of processes \(L\). The semaphore must always be initialised to some \((v, \emptyset)\).

- \(v\) : how many more processes we can let in without waiting.
- \(L\) : the processes currently waiting to get in.
Semaphores

Definition

There are two basic operations a process \( p \) can do on semaphore \( S \):

- **wait(\( S \))** or **P(\( S \))**, decrements \( v \) if positive, otherwise adds \( p \) to \( L \) and *blocks* \( p \).
Semaphores

Definition
There are two basic operations a process $p$ can do on semaphore $S$:

- **wait**$(S)$ or $P(S)$, decrements $v$ if positive, otherwise adds $p$ to $L$ and *blocks* $p$.
- **signal**$(S)$ or $V(S)$, if $L \neq \emptyset$, unblocks a member of $L$. Otherwise increment $v$. 

Example (Promela Encoding)

1. `inline wait(S) { d_step { S > 0; S-- }}`
2. `inline signal(S) { d_step { S ++ } }`

This is known as a busy-wait semaphore as the set $L$ is implicitly the set of (busy-)waiting processes on the integer.
**Semaphores**

**Definition**
There are two basic operations a process $p$ can do on semaphore $S$:

- **wait**($S$) or $P(S)$, decrements $v$ if positive, otherwise adds $p$ to $L$ and blocks $p$.
- **signal**($S$) or $V(S)$, if $L \neq \emptyset$, unblocks a member of $L$. Otherwise increment $v$.

**Example (Promela Encoding)**

```plaintext
1  inline wait(S) { d_step { S > 0; S-- } }
2  inline signal(S) { d_step { S ++ } }
```

This is known as a **busy-wait** semaphore as the set $L$ is implicitly the set of (busy-‐)waiting processes on the integer.
Critical Sections

Locks are just semaphores where the integer starts at 1:
Critical Sections

Locks are just semaphores where the integer starts at 1:

<table>
<thead>
<tr>
<th>semaphore $S \leftarrow (1, \emptyset)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>forever do</td>
</tr>
<tr>
<td>$p_1$ non-critical s.</td>
</tr>
<tr>
<td>$p_2$ wait ($S$)</td>
</tr>
<tr>
<td>$p_3$ critical s.</td>
</tr>
<tr>
<td>$p_4$ signal ($S$)</td>
</tr>
<tr>
<td>forever do</td>
</tr>
<tr>
<td>$q_1$ non-critical s.</td>
</tr>
<tr>
<td>$q_2$ wait ($S$);</td>
</tr>
<tr>
<td>$q_3$ critical s.</td>
</tr>
<tr>
<td>$q_4$ signal ($S$);</td>
</tr>
</tbody>
</table>

A weak semaphore is like our set model earlier. A busy-wait semaphore has no set, and implements blocking by spinning in a loop.

Question

What impact does weak vs. busy-wait have on eventual entry?
Semaphores

Producer-Consumer

Critical Sections

Locks are just semaphores where the integer starts at 1:

<table>
<thead>
<tr>
<th>forever do</th>
<th>forever do</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁</td>
<td>non-critical s.</td>
</tr>
<tr>
<td>p₂</td>
<td>wait (S)</td>
</tr>
<tr>
<td>p₃</td>
<td>critical s.</td>
</tr>
<tr>
<td>p₄</td>
<td>signal (S)</td>
</tr>
<tr>
<td>q₁</td>
<td>non-critical s.</td>
</tr>
<tr>
<td>q₂</td>
<td>wait (S);</td>
</tr>
<tr>
<td>q₃</td>
<td>critical s.</td>
</tr>
<tr>
<td>q₄</td>
<td>signal (S);</td>
</tr>
</tbody>
</table>

A weak semaphore is like our set model earlier. A busy-wait semaphore has no set, and implements blocking by spinning in a loop.

Question

What impact does weak vs. busy-wait have on eventual entry?
For $N$ processes

<table>
<thead>
<tr>
<th>semaphore $S \leftarrow (1, \emptyset)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>each process $i$:</td>
</tr>
<tr>
<td>forever do</td>
</tr>
<tr>
<td>$i_1$  non-critical section</td>
</tr>
<tr>
<td>$i_2$  wait ($S$)</td>
</tr>
<tr>
<td>$i_3$  critical section</td>
</tr>
<tr>
<td>$i_4$  signal ($S$)</td>
</tr>
</tbody>
</table>

Problem 1: With a weak or busy-wait semaphore we don’t get eventual entry.

Problem 2: Even with strong fairness, we don’t have linear waiting.

Strong Semaphores
Replace the set $L$ with a queue, wake processes up in FIFO order. This guarantees linear waiting, but is harder to implement and potentially more expensive.
For $N$ processes

<table>
<thead>
<tr>
<th>semaphore $S \leftarrow (1, \emptyset)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>each process</em> $i$:</td>
</tr>
<tr>
<td>forever do</td>
</tr>
<tr>
<td>$i_1$ <em>non-critical section</em></td>
</tr>
<tr>
<td>$i_2$ <em>wait</em> ($S$)</td>
</tr>
<tr>
<td>$i_3$ <em>critical section</em></td>
</tr>
<tr>
<td>$i_4$ <em>signal</em> ($S$)</td>
</tr>
</tbody>
</table>

**Problem 1:** With a weak or busy-wait semaphore we don’t get eventual entry.

**Problem 2:** Even with strong fairness, we don’t have *linear waiting*. 
For $N$ processes

<table>
<thead>
<tr>
<th>semaphore $S \leftarrow (1, \emptyset)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>each process $i$:</td>
</tr>
<tr>
<td>forever do</td>
</tr>
<tr>
<td>$i_1$ non-critical section</td>
</tr>
<tr>
<td>$i_2$ wait ($S$)</td>
</tr>
<tr>
<td>$i_3$ critical section</td>
</tr>
<tr>
<td>$i_4$ signal ($S$)</td>
</tr>
</tbody>
</table>

**Problem 1:** With a weak or busy-wait semaphore we don’t get eventual entry.

**Problem 2:** Even with strong fairness, we don’t have *linear waiting*.

**Strong Semaphores**

Replace the set $L$ with a *queue*, wake processes up in FIFO order.

This guarantees *linear waiting*, but is harder to implement and potentially more expensive.
Reasoning about Semaphores

For a semaphore $S = (v, L)$ initialised to $(k, \emptyset)$, the following invariants always hold:

1. $v = k + \#\text{signal}(S) - \#\text{wait}(S)$
2. $v \geq 0$

Definitions

1. $\#\text{signal}(S)$: how many times $\text{signal}(S)$ has successfully executed.
2. $\#\text{wait}(S)$: how many times $\text{wait}(S)$ has successfully executed.

A successful execution happens when the process has proceeded to the next statement. So if a process is blocked on a $\text{wait}(S)$, then $\#\text{wait}(S)$ will not increase until the process is unblocked.

Example (Mutual Exclusion)

The no. of processes in their CS = $\#\text{wait}(S) - \#\text{signal}(S)$. Let’s use this to show our usual properties.
Reasoning about Semaphores

For a semaphore $S = (v, L)$ initialised to $(k, \emptyset)$, the following invariants always hold:

1. $v = k + \#\text{signal}(S) - \#\text{wait}(S)$
2. $v \geq 0$
Reasoning about Semaphores

For a semaphore $S = (v, L)$ initialised to $(k, \emptyset)$, the following invariants always hold:

1. $v = k + \#\text{signal}(S) - \#\text{wait}(S)$
2. $v \geq 0$

Definitions

1. $\#\text{signal}(S)$: how many times $\text{signal}(S)$ has successfully executed.
2. $\#\text{wait}(S)$: how many times $\text{wait}(S)$ has successfully executed.
Reasoning about Semaphores

For a semaphore $S = (v, L)$ initialised to $(k, \emptyset)$, the following invariants always hold:

1. $v = k + \#\text{signal}(S) - \#\text{wait}(S)$
2. $v \geq 0$

Definitions

1. $\#\text{signal}(S)$: how many times $\text{signal}(S)$ has successfully executed.
2. $\#\text{wait}(S)$: how many times $\text{wait}(S)$ has successfully executed.

A *successful* execution happens when the process has proceeded to the next statement. So if a process is blocked on a $\text{wait}(S)$, then $\#\text{wait}(S)$ will not increase until the process is unblocked.
Reasoning about Semaphores

For a semaphore $S = (v, L)$ initialised to $(k, \emptyset)$, the following invariants always hold:

1. $v = k + \#\text{signal}(S) - \#\text{wait}(S)$
2. $v \geq 0$

Definitions

1. $\#\text{signal}(S)$: how many times $\text{signal}(S)$ has successfully executed.
2. $\#\text{wait}(S)$: how many times $\text{wait}(S)$ has successfully executed.

A successful execution happens when the process has proceeded to the next statement. So if a process is blocked on a $\text{wait}(S)$, then $\#\text{wait}(S)$ will not increase until the process is unblocked.

Example (Mutual Exclusion)

The no. of processes in their CS = $\#\text{wait}(S) - \#\text{signal}(S)$. Let’s use this to show our usual properties.
Safety Properties

Mutual Exclusion

We know:

1. \( v = 1 + \#\text{signal}(S) - \#\text{wait}(S) \) (our first semaphore invariant)

2. \( v \geq 0 \) (our second semaphore invariant)

3. \( \#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S) \) (our observed invariant)

From these invariants it is possible to show that \( \#\text{CS} \leq 1 \), i.e. mutual exclusion.

Absence of Deadlock

Assume that deadlock occurs by all processes being blocked on \textit{wait}, so no process can enter its critical section \( (\#\text{CS} = 0) \). Then \( v = 0 \), contradicting our semaphore invariants above. So there cannot be deadlock.
Safety Properties

Mutual Exclusion

We know:

1. \( v = 1 + \#\text{signal}(S) - \#\text{wait}(S) \) (our first semaphore invariant)
2. \( v \geq 0 \) (our second semaphore invariant)
Semaphores

Producer-Consumer

Safety Properties

Mutual Exclusion

We know:

1. \( v = 1 + \#\text{signal}(S) - \#\text{wait}(S) \) (our first semaphore invariant)

2. \( v \geq 0 \) (our second semaphore invariant)

3. \( \#CS = \#\text{wait}(S) - \#\text{signal}(S) \) (our observed invariant)

From these invariants it is possible to show that \( \#CS \leq 1 \), i.e. mutual exclusion.

Absence of Deadlock

Assume that deadlock occurs by all processes being blocked on \text{wait}, so no process can enter its critical section \( \#CS = 0 \). Then \( v = 0 \), contradicting our semaphore invariants above. So there cannot be deadlock.
Safety Properties

Mutual Exclusion

We know:

1. \( v = 1 + \#\text{signal}(S) - \#\text{wait}(S) \) (our first semaphore invariant)
2. \( v \geq 0 \) (our second semaphore invariant)
3. \( \#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S) \) (our observed invariant)

From these invariants it is possible to show that \( \#\text{CS} \leq 1 \), i.e. mutual exclusion.

Absence of Deadlock

Assume that deadlock occurs by all processes being blocked on `wait`, so no process can enter its critical section (\( \#\text{CS} = 0 \)). Then \( v = 0 \), contradicting our semaphore invariants above. So there cannot be deadlock.
Safety Properties

Mutual Exclusion

We know:

1. \( v = 1 + \#\text{signal}(S) - \#\text{wait}(S) \) (our first semaphore invariant)
2. \( v \geq 0 \) (our second semaphore invariant)
3. \( \#CS = \#\text{wait}(S) - \#\text{signal}(S) \) (our observed invariant)

From these invariants it is possible to show that \( \#CS \leq 1 \), i.e. mutual exclusion.

Absence of Deadlock

Assume that deadlock occurs by all processes being blocked on \text{wait}, so no process can enter its critical section (\( \#CS = 0 \)).
Safety Properties

Mutual Exclusion
We know:
1. \( v = 1 + \#\text{signal}(S) - \#\text{wait}(S) \) (our first semaphore invariant)
2. \( v \geq 0 \) (our second semaphore invariant)
3. \( \#CS = \#\text{wait}(S) - \#\text{signal}(S) \) (our observed invariant)
From these invariants it is possible to show that \( \#CS \leq 1 \), i.e. mutual exclusion.

Absence of Deadlock
Assume that deadlock occurs by all processes being blocked on \textit{wait}, so no process can enter its critical section (\( \#CS = 0 \)). Then \( v = 0 \), contradicting our semaphore invariants above. So there cannot be deadlock.
Liveness Properties

To simplify things, we will prove for only two processes, $p$ and $q$.

**Eventual Entry for $p$ (with weak semaphores)**

Assume that $p$ is starved, indefinitely blocked on the wait.
Liveness Properties

To simplify things, we will prove for only two processes, $p$ and $q$.

**Eventual Entry for $p$ (with weak semaphores)**

Assume that $p$ is starved, indefinitely blocked on the wait. Therefore $S = (0, L)$ and $p \in L$. 

We know therefore, substituting into our invariants:

1. $0 = 1 + \#\text{signal}(S) - \#\text{wait}(S)$
2. $\#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S)$

From which we can conclude that $\#\text{CS} = 1$.

Therefore $q$ must be in its critical section and $L = \{p\}$.

We know (or rather, assume) that eventually $q$ will eventually finish its CS and signal$(S)$.

Thus, $p$ will be unblocked, causing it to gain entry — **Contradiction**.
Liveness Properties

To simplify things, we will prove for only two processes, $p$ and $q$.

**Eventual Entry for $p$ (with weak semaphores)**

Assume that $p$ is starved, indefinitely blocked on the `wait`. Therefore $S = (0, L)$ and $p \in L$.

We know therefore, substituting into our invariants:

1. $0 = 1 + \#\text{signal}(S) - \#\text{wait}(S)$
Liveness Properties

To simplify things, we will prove for only two processes, $p$ and $q$.

**Eventual Entry for $p$ (with weak semaphores)**

Assume that $p$ is starved, indefinitely blocked on the `wait`. Therefore $S = (0, L)$ and $p \in L$.

We know therefore, substituting into our invariants:

1. $0 = 1 + \#\text{signal}(S) - \#\text{wait}(S)$
2. $\#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S)$

Therefore $q$ must be in its critical section and $L = \{p\}$.

We know (or rather, assume) that eventually $q$ will eventually finish its CS and `signal(S)`. Thus, $p$ will be unblocked, causing it to gain entry — Contradiction.
Liveness Properties

To simplify things, we will prove for only two processes, \( p \) and \( q \).

**Eventual Entry for \( p \) (with weak semaphores)**

Assume that \( p \) is starved, indefinitely blocked on the `wait`. Therefore \( S = (0, L) \) and \( p \in L \).

We know therefore, substituting into our invariants:

1. \( 0 = 1 + \#\text{signal}(S) - \#\text{wait}(S) \)
2. \( \#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S) \)

From which we can conclude that \( \#\text{CS} = 1 \).
Liveness Properties

To simplify things, we will prove for only two processes, $p$ and $q$.

**Eventual Entry for $p$ (with weak semaphores)**

Assume that $p$ is starved, indefinitely blocked on the `wait`. Therefore $S = (0, L)$ and $p \in L$.

We know therefore, substituting into our invariants:

1. $0 = 1 + \#\text{signal}(S) - \#\text{wait}(S)$
2. $\#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S)$

From which we can conclude that $\#\text{CS} = 1$. Therefore $q$ must be in its critical section and $L = \{p\}$. 
Liveness Properties

To simplify things, we will prove for only two processes, $p$ and $q$.

**Eventual Entry for $p$ (with weak semaphores)**

Assume that $p$ is starved, indefinitely blocked on the `wait`. Therefore $S = (0, L)$ and $p \in L$.

We know therefore, substituting into our invariants:

1. $0 = 1 + \#\text{signal}(S) - \#\text{wait}(S)$
2. $\#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S)$

From which we can conclude that $\#\text{CS} = 1$. Therefore $q$ must be in its critical section and $L = \{p\}$.

We know (or rather, assume) that eventually $q$ will eventually finish its CS and $\text{signal}(S)$.
Liveness Properties

To simplify things, we will prove for only two processes, \( p \) and \( q \).

**Eventual Entry for \( p \) (with weak semaphores)**

Assume that \( p \) is starved, indefinitely blocked on the `wait`. Therefore \( S = (0, L) \) and \( p \in L \).

We know therefore, substituting into our invariants:

1. \( 0 = 1 + \#\text{signal}(S) - \#\text{wait}(S) \)
2. \( \#\text{CS} = \#\text{wait}(S) - \#\text{signal}(S) \)

From which we can conclude that \( \#\text{CS} = 1 \). Therefore \( q \) must be in its critical section and \( L = \{ p \} \).

We know (or rather, assume) that eventually \( q \) will eventually finish its CS and `signal(S)`.

Thus, \( p \) will be unblocked, causing it to gain entry — Contradiction.
Rendezvous

In addition (and perhaps simpler) than the mutual exclusion/critical section problem, the *rendezvous* problem is also a basic unit of synchronisation for solving concurrency problems. Assume we have two processes with two statements each:

<table>
<thead>
<tr>
<th>Rendezvous</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>$first_P$</td>
</tr>
<tr>
<td>$second_P$</td>
</tr>
</tbody>
</table>
Rendezvous

In addition (and perhaps simpler) than the mutual exclusion/critical section problem, the *rendezvous* problem is also a basic unit of synchronisation for solving concurrency problems. Assume we have two processes with two statements each:

<table>
<thead>
<tr>
<th>Rendezvous</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>first_P</td>
</tr>
<tr>
<td>second_P</td>
</tr>
<tr>
<td>Q</td>
</tr>
<tr>
<td>first_Q</td>
</tr>
<tr>
<td>second_Q</td>
</tr>
</tbody>
</table>

**Problem**

How do we ensure that all *first* statements happen before all *second* statements?

In Java
Producer-Consumer

*Binary semaphores* (aka locks) are not the only use of semaphores!
Producer-Consumer

Binary semaphores (aka locks) are not the only use of semaphores!

Producer-Consumer Problem

A producer process and a consumer process share access to a shared buffer of data. This buffer acts as a queue. The producer adds messages to the queue, and the consumer reads messages from the queue. If there are no messages in the queue, the consumer blocks until there are messages.
**Semaphores**

*Binary semaphores* (aka locks) are not the only use of semaphores!

### Producer-Consumer Problem

A **producer** process and a **consumer** process share access to a shared buffer of data. This buffer acts as a **queue**. The producer adds messages to the queue, and the consumer reads messages from the queue. If there are no messages in the queue, the consumer blocks until there are messages.

#### Algorithm 1.3: Producer-consumer (infinite buffer)

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>T d</strong></td>
<td><strong>T d</strong></td>
</tr>
<tr>
<td><strong>forever do</strong></td>
<td><strong>forever do</strong></td>
</tr>
<tr>
<td>p1: d ← produce</td>
<td>q1: <strong>wait</strong>(full)</td>
</tr>
<tr>
<td>p2: append(d, buffer)</td>
<td>q2: d ← take(buffer)</td>
</tr>
<tr>
<td>p3: <strong>signal</strong>(full)</td>
<td>q3: consume(d)</td>
</tr>
</tbody>
</table>
Finite buffer

What about if the buffer can become full, and the producer must block until space is freed?
Finite buffer

What about if the buffer can become full, and the producer must block until space is freed?

Use another semaphore!: 

```
bounded[N] queue[T] buffer ← empty queue
semaphore full ← (0, ∅)
semaphore empty ← (N, ∅)

producer consumer
T d T d
loop forever loop forever
p1: d ← produce q1: wait (full)
p2: wait (empty) q2: d ← take(buffer)
p3: append(d, buffer) q3: signal (empty)
p4: signal (full) q4: consume(d)
```

This pattern is called split semaphores.
Finite buffer

What about if the buffer can become full, and the producer must block until space is freed?

Use another semaphore:

Algorithm 1.6: Producer-consumer (finite buffer, semaphores)

| bounded[N] queue[T] buffer ← empty queue |
| semantic full ← (0, ∅) |
| semantic empty ← (N, ∅) |

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>T d</td>
<td>T d</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: d ← produce</td>
<td>q1: wait(full)</td>
</tr>
<tr>
<td>p2: wait(empty)</td>
<td>q2: d ← take(buffer)</td>
</tr>
<tr>
<td>p3: append(d, buffer)</td>
<td>q3: signal(empty)</td>
</tr>
<tr>
<td>p4: signal(full)</td>
<td>q4: consume(d)</td>
</tr>
</tbody>
</table>

This pattern is called split semaphores.
## A specific Example

### Algorithm 1.7: Producer/Consumer ($b$-place buffer, sem’s)

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer $i \leftarrow 0$</td>
<td>integer $k \leftarrow 0$, $t \leftarrow 0$</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: <code>wait(empty)</code></td>
<td>q1: <code>wait(full)</code></td>
</tr>
<tr>
<td>p2: $\text{data}[i \mod b] \leftarrow g(i)$</td>
<td>q2: $t \leftarrow t + \text{data}[k \mod b]$</td>
</tr>
<tr>
<td>p3: $i++$</td>
<td>q3: $k++$</td>
</tr>
<tr>
<td>p4: <code>signal(full)</code></td>
<td>q4: <code>signal(empty)</code></td>
</tr>
</tbody>
</table>
What do we prove?

The crucial properties of this pair of processes include:

**safety**  \[ S = \left( t = \sum_{j=0}^{k-1} g(j) \right) \] is an invariant

**liveness**  \( k \) keeps increasing
How do we prove?

To show the safety property, we

- translate the pseudo code into transition diagrams,
- define a pre-condition $\phi$,
- define an assertion network $Q$,
- prove that $Q$ is (a) inductive and (b) interference-free,
- prove that the initial assertions $Q_{p1}$ and $Q_{q1}$ follow from $\phi$, and
- prove that each of the consumer’s assertions implies the invariant $S$. 
1 Transition Diagrams

Semaphores

Producer-Consumer

- p1
  - Transition: e > 0; e--
  - Transition: f++
  - Transition: \( \text{data}[i \% b], i \leftarrow g(i), i + 1 \)

- p2

- q1
  - Transition: f > 0; f--
  - Transition: e++
  - Transition: t, k \leftarrow t + \text{data}[k \% b], k + 1

- q2

- q4
  - Transition: f++
2 Precondition

As precondition we collect the initial values of those global and local variables which are read before they are written.

\[ \phi = (e = b \land f = 0 \land i = k = t = 0) \]
3 Assertion Network 1

We start by collecting further likely invariants.

The consumer can’t overtake the producer:

\[ 0 \leq k \leq i \]  

(1)

The producer can’t lap the consumer:

\[ i - k \leq b \]  

(2)

The buffer shows a subsequence of \( g \)’s values:

\[ \forall j \in a..i - 1 \ (data[j \% b] = g(j)) \ , \text{ where } a = \max(0, i - b) \]  

(3)
3 Assertion Network II

semaphore invariants:

\[ e, f \in 0..b \]  
\[ e = b + \text{#signal}(e) - \text{#wait}(e) \]  
\[ f = \text{#signal}(f) - \text{#wait}(f) \]

numbers of waits and signals are correlated:

\[ \text{#wait}(e) = \text{#signal}(f) + 1 - p_1 = i + p_2 \]  
\[ \text{#signal}(f) = \text{#wait}(e) - p_{2,4} = i - p_4 \]  
\[ \text{#wait}(f) = \text{#signal}(e) + 1 - q_1 = k + q_2 \]  
\[ \text{#signal}(e) = \text{#wait}(f) - q_{2,4} = k - q_4 \]
3 Assertion Network III

Semaphore values are correlated:

\[ e + f = b - p_{2,4} - q_{2,4} \]  \hspace{1cm} (11)

Our goal:

\[ S = (1) \land \ldots \land (12) \]  \hspace{1cm} (12)

Assuming that the invariants (1)–(12) gather all that’s going on, we may now try to prove that the assertion network consisting of the same assertion,

\[ I = (1) \land \ldots \land (12) \]

at every location is inductive and interference-free.
4(a) Q is inductive

We need to prove local correctness of each of the 6 transitions. We assume that the auxiliary variables $p_1$, $p_2$, $p_4$, $q_1$, $q_2$, and $q_4$ are implicitly set to 0 resp. 1, depending on the locations.

\[
\begin{align*}
    p_1 \rightarrow p_2: & \quad \models I \land e > 0 \implies I \circ (e \leftarrow e - 1) \quad (13) \\
    p_2 \rightarrow p_4: & \quad \models I \implies I \circ (data[i\%b], i \leftarrow g(i), i + 1) \quad (14) \\
    p_4 \rightarrow p_1: & \quad \models I \implies I \circ (f \leftarrow f + 1) \quad (15) \\
    q_1 \rightarrow q_2: & \quad \models I \land f > 0 \implies I \circ (f \leftarrow f - 1) \quad (16) \\
    q_2 \rightarrow q_4: & \quad \models I \implies I \circ (t, k \leftarrow t + data[k\%b], i + 1) \quad (17) \\
    q_4 \rightarrow q_1: & \quad \models I \implies I \circ (e \leftarrow e + 1) \quad (18)
\end{align*}
\]
4(b) $Q$ is interference-free

Finally it pays off to give such a degenerate assertion network: *interference-freedom comes for free* since we’ve proved inductivity (local correctness) already.
5 \( \phi \) is strong enough

Since all assertions are the same, we only need to show that (at \( p1 \) and \( q1 \)):

\[ \phi \implies I \]

which is straightforward.

6 \( S \) follows from \( Q \)

Trivially true since \( S \) is the last conjunct of \( I \).
Liveness

**Deadlock Freedom**

The only global location with a potential for deadlock would be $p_1/q_1$.
Constant $b > 0$ and invariant (11) ensure that at $p_1/q_1$, not both semaphores can be 0.
Liveness

**Deadlock Freedom**

The only global location with a potential for deadlock would be $p_1/q_1$.

Constant $b > 0$ and invariant (11) ensure that at $p_1/q_1$, not both semaphores can be 0.

**Liveness Property**

Suppose one of the processes (say the consumer) is stuck at location 1 forever, and thus $k$ does not increase.
Liveness

**Deadlock Freedom**
The only global location with a potential for deadlock would be $p_1/q_1$.
Constant $b > 0$ and invariant (11) ensure that at $p_1/q_1$, not both semaphores can be 0.

**Liveness Property**
Suppose one of the processes (say the consumer) is stuck at location 1 forever, and thus $k$ does not increase.
Then, by deadlock-freedom, the producer would have to keep going indefinitely without ever incrementing $f$—but it does so every round.
What Now?

Next lecture, we’ll be looking at Monitors and the Readers and Writers problem. This week’s homework involves Java programming. There’s a number of resources (prepared by Vladimir Tosic) on the website to assist you. Assignment 1 is also coming out this week.