

# G52CON: Concepts of Concurrency

## Lecture 8 Semaphores II

Brian Logan  
School of Computer Science & IT  
bsl@cs.nott.ac.uk

## Outline of this lecture

- problem solving with semaphores
- solving Producer-Consumer problems using buffers:
  - single element buffer
  - bounded buffer
- coursework
- exercise 2

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

2

## Producer-Consumer problem

Given two processes, a *producer* which generates data items, and a *consumer* which consumes them, find a mechanism for passing data from the producer to the consumer such that:

- no items are lost or duplicated in transit;
- items are consumed in the order they are produced; and
- all items produced are eventually consumed.

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

3

## Variants of the problem

The single Producer–single Consumer problem can be generalised:

- multiple producers–single consumer
- single producer–multiple consumers
- multiple producers–multiple consumers

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

4

## Buffer-based solutions

In multiprogramming or multiprocessing implementations of concurrency, communication between a producer and a consumer is often implemented using a *shared buffer*:

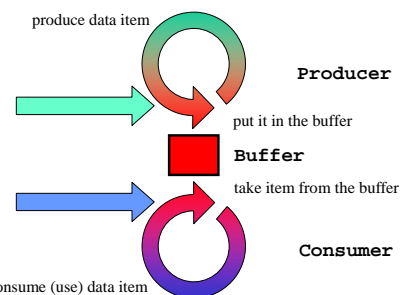
- a *buffer* is an area of memory used for the temporary storage of data while in transit from one process to another.
- the producer writes into the buffer and the consumer reads from the buffer, e.g., a Unix pipe.

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

5

## Interprocess communication



© Brian Logan 2007

G52CON Lecture 8: Semaphores II

6

## Synchronisation

The general multiple Producer-multiple Consumer problem requires both mutual exclusion and condition synchronisation:

- *mutual exclusion* is used to ensure that more than one producer or consumer does not access the same buffer slot at the same time;
- *condition synchronisation* is used to ensure that data is not read before it has been written, and that data is not overwritten before it has been read.

Synchronisation can be achieved using any of the techniques we have seen so far: e.g., Peterson's algorithm, semaphores.

## General synchronisation conditions

Buffer-based solutions to the Producer-Consumer problem should satisfy the following conditions:

- no "items" are read from an empty buffer;
- data items are read only once;
- data items are not overwritten before they are read;
- items are consumed in the order they are produced; and
- all items produced are eventually consumed.

in addition to the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry.

## Solutions

- may be judged on different criteria, e.g., correctness, fairness, efficiency
- may use different sizes of buffer, and different protocols for synchronising access to the buffer
- a particular solution can be implemented using different synchronisation primitives, e.g., spin locks or semaphores
- a particular synchronisation primitive or protocol can be implemented in different ways, e.g., busy waiting, blocking

## Infinite buffer

The producer and consumer communicate via an *infinite shared buffer*:

- no "items" are read from an empty buffer;
- data items are read only once;
- **the producer may produce a new item at any time;**
- items are consumed in the order they are produced; and
- all items produced are eventually consumed.

## Infinite buffer solution

```
// Producer process           // Consumer process
Object v = null;             Object w = null;
integer in = 0;              integer out = 0;
while(true) {                while(true) {
    // produce data v         P(n);
    ...                       w = buf[out];
    buf[in] = v;              out = out + 1;
    in = in + 1;              // use the data w
    V(n);                     ...
}                               }

// Shared variables
Object[] buf = new Object[∞];
general semaphore n = 0;
```

## Problem 1: single element buffer

Devise a solution to Producer-Consumer problem using a *single element buffer* which ensures that:

- the producer may only produce an item when the buffer is empty; and
- the consumer may only consume an item when the buffer is full.

Your solution should satisfy the general synchronisation requirements for the Producer-Consumer problem and the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry.

## Infinite vs single element buffer

- with an infinite buffer we had only one problem—to prevent the consumer getting ahead of the producer
- with a single element buffer we have two problems
  - preventing the consumer getting ahead of the producer; and
  - preventing the producer getting ahead of the consumer.

## Problem 1: first attempt

```
// Producer process           // Consumer process
Object x = null;             Object y = null;
while(true) {                while(true) {
    // produce data x         P(s);
    ...                       y = buf;
    P(s);                     V(s);
    buf = x;                  // use the data y
    V(s);                     ...
}                               }

// Shared variables
Object buf;
binary semaphore s = 1;
```

## Properties of the first attempt

Does the first attempt satisfy the following properties:

- **Mutual Exclusion:** yes/no
- **Absence of Deadlock:** yes/no
- **Absence of Unnecessary Delay:** yes/no
- **Eventual Entry:** yes/no

## Properties of the first attempt

Does the first attempt satisfy the following properties:

- **Mutual Exclusion:** yes
- **Absence of Deadlock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** yes

## First attempt synchronisation conditions

Does the solution satisfy the following properties:

- **no items are read from an empty buffer:** yes/no
- **data items are read only once:** yes/no
- **data items are not overwritten before they are read:** yes/no
- **items are consumed in the order they are produced:** yes/no
- **all items produced are eventually consumed:** yes/no

## First attempt synchronisation conditions

Does the solution satisfy the following properties:

- **no items are read from an empty buffer:** no
- **data items are read only once:** no
- **data items are not overwritten before they are read:** no
- **items are consumed in the order they are produced:** yes
- **all items produced are eventually consumed:** no

## Problem 1: second attempt

```
// Producer process           // Consumer process
Object x = null;             Object y = null;
while(true) {                 while(true) {
    // produce data x         P(s);
    ...                       y = buf;
    buf = x;                  // use the data y
    V(s);                     ...
}                               }

// Shared variables
Object buf;
binary semaphore s = 0;
```

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

19

## Properties of the second attempt

Does the second attempt satisfy the following properties:

- **Mutual Exclusion:** yes/no
- **Absence of Deadlock:** yes/no
- **Absence of Unnecessary Delay:** yes/no
- **Eventual Entry:** yes/no

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

20

## Properties of the second attempt

Does the second attempt satisfy the following properties:

- **Mutual Exclusion:** no
- **Absence of Deadlock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** yes

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

21

## Second attempt synchronisation conditions

Does the solution satisfy the following properties:

- **no items are read from an empty buffer:** yes/no
- **data items are read only once:** yes/no
- **data items are not overwritten before they are read:** yes/no
- **items are consumed in the order they are produced:** yes/no
- **all items produced are eventually consumed:** yes/no

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

22

## Second attempt synchronisation conditions

Does the solution satisfy the following properties:

- **no items are read from an empty buffer:** yes
- **data items are read only once: at most three times**
- **data items are not overwritten before they are read:** no
- **items are consumed in the order they are produced:** yes
- **all items produced are eventually consumed:** no

"Data items are read at most three times" if a V operation on a binary semaphore which has value 1 does not increment the value of the semaphore.

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

23

## Single element buffer solution

```
// Producer process           // Consumer process
Object x = null;             Object y = null;
while(true) {                 while(true) {
    // produce data x         P(full);
    ...                       y = buf;
    P(empty);                 V(empty);
    buf = x;                  // use the data y
    V(full);                   ...
}                               }

// Shared variables
Object buf;
binary semaphore empty = 1, full = 0;
```

© Brian Logan 2007

G52CON Lecture 8: Semaphores II

24

## Properties of the single buffer solution

The single element buffer solution satisfies the following properties:

- **Mutual Exclusion:** yes
- **Absence of Deadlock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** yes

## Single buffer solution synchronisation conditions

The single element buffer solution satisfies the following properties:

- **no items are read from an empty buffer:** yes
- **data items are read only once:** yes
- **data items are not overwritten before they are read:** yes
- **items are consumed in the order they are produced:** yes
- **all items produced are eventually consumed:** yes

## Applications of Single element buffers

- I/O to all types of peripheral devices
- dedicated programs running on bare machines.

## Larger buffers

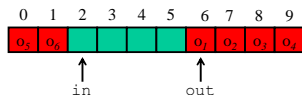
A single element buffer works well if the Producer and Consumer processes run *at the same rate*:

- processes don't have to wait very long to access the single buffer
- many low-level synchronisation problems are solved in this way, e.g., interrupt driven I/O.

If the speed of the Producer and Consumer is only *the same on average*, and fluctuates over short periods, a larger buffer can significantly increase performance by reducing the number of times processes block.

## Bounded buffers

A *bounded buffer* of length  $n$  is a circular communication buffer containing  $n$  slots. The buffer contains a queue of items which have produced but not yet consumed. For example



`out` is the index of the item at the head of the queue, and `in` is the index of the first empty slot at the end of the queue.

## Problem 2: bounded buffer

Devise a solution to Producer-Consumer problem using a *bounded buffer* which ensures that:

- the producer may only produce an item when there is an empty slot in the buffer; and
- the consumer may only consume an item when there is a full slot in the buffer.

Your solution should satisfy the general synchronisation requirements for the Producer-Consumer problem and the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry.

## Single element vs bounded buffer

Note that the synchronisation conditions are really the same as for the single element (& infinite) buffer:

- the producer may only produce an item when the buffer is not full; and
- the consumer may only consume an item when the buffer is not empty.

and the problems are the same:

- preventing the consumer getting ahead of the producer; and
- preventing the producer getting ahead of the consumer.

## Bounded buffer solution

```
// Producer process           // Consumer process
Object x = null;             Object y = null;
integer in = 0;              integer out = 0;
while(true) {                while(true) {
    // produce data x         P(full);
    ...                       y = buf[out];
    P(empty);                 out = (out + 1) % n;
    buf[in] = x;              V(empty);
    in = (in + 1) % n;        // use the data y
    V(full);                  ...
}                               }

// Shared variables
integer n = BUFFER_SIZE;
Object[] buf = new Object[n];
general semaphore empty = n, full = 0;
```

## Properties of the bounded buffer solution

The bounded buffer solution satisfies the following properties:

- **Mutual Exclusion:** yes
- **Absence of Deadlock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** yes

## Bounded buffer solution synchronisation conditions

The bounded buffer solution satisfies the following properties:

- **data items are not overwritten before they are read:** yes
- **data items are read only once:** yes
- **no items are read from an empty buffer:** yes
- **items are consumed in the order they are produced:** yes
- **all items produced are eventually consumed:** yes

## Bounded buffer solution 2

```
// Producer process           // Consumer process
Object x = null;             Object y = null;
integer in = 0;              integer out = 0;
while(true) {                while(true) {
    // produce data x         P(full);
    ...                       y = buf[out];
    P(empty);                 V(empty);
    buf[in] = x;              out = (out + 1) % n;
    V(full);                  // use the data y
    in = (in + 1) % n;        ...
}                               }

// Shared variables
final integer n = BUFFER_SIZE;
Object[] buf = new Object[n];
general semaphore empty = n, full = 0;
```

## Applications of bounded buffers

Bounded buffers are used for serial input and output streams in many operating systems:

- Unix maintains queues of characters for I/O on all serial character devices such as keyboards, screens and printers.
- Unix pipes are implemented using bounded buffers.

## Coursework

The coursework involves writing a Java `BinarySearchTree` class which provides the following public methods:

- `BinarySearchTree()`: constructs an empty binary search tree
- `boolean isEmpty()`: returns true if the tree contains no data items
- `boolean insert(Object x)`: if `x` is not already present in the tree, adds a node containing `x` and returns true, otherwise returns false
- `boolean search(Object x)`: returns true if the object `x` is present in the tree
- `boolean delete(Object x)`: if `x` is present in the tree, removes it and returns true, otherwise returns false

Your implementation should allow safe concurrent access by multiple threads. Full details on the module web page.

## Submission

- the submission consists of two parts:
  - the code implementing the `BinarySearchTree` class
  - an essay explaining your solution and why it is correct
- submissions are due on **Friday the 14th of March at 3:30pm** and should be made electronically using the coursework submission system
- the ID for the coursework is 90
- submissions open at 2pm on Thursday the 14<sup>th</sup> of February

## Assessment

- assessment is based on:
  - the correctness and clarity of your implementation
  - the degree of concurrency it allows
  - the correctness and clarity of your explanation of the code
- a solution which is correct will pass, but for a higher mark your solution should also maximise concurrency
- there will be a (small) prize for the best solution

## Plagiarism

From the School's Policy on Plagiarism:

*The following actions are considered to be plagiarism:*

- *copying paragraphs or programs from a textbook;*
- *copying another person's work either with or without their knowledge;*
- *working together in groups of two or more to produce a single program or essay and then each member of the group submitting a copy of this as their own work.*

Guidance on what constitutes plagiarism in the context of the coursework can be found on the module web page

## Exercise 2: Semaphores

- a) devise a solution to *multiple Producer–multiple Consumer* problem using a bounded buffer which ensures that:
- no items are read from an empty buffer;
  - data items are read only once;
  - data items are not overwritten before they are read;
  - items are consumed in the order they are produced; and
  - all items produced are eventually consumed.
- b) does your solution satisfy the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry?
- c) how many classes of critical sections does your solution have?

## The next lecture

### *Monitors*

Suggested reading:

- Andrews (2000), chapter 5;
- Ben-Ari (1982), chapter 5;
- Burns & Davies (1993), chapter 7, sections 7.4–7.9.