G52CON: Concepts of Concurrency

Lecture 8 Semaphores II

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

Outline of this lecture

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

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.

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

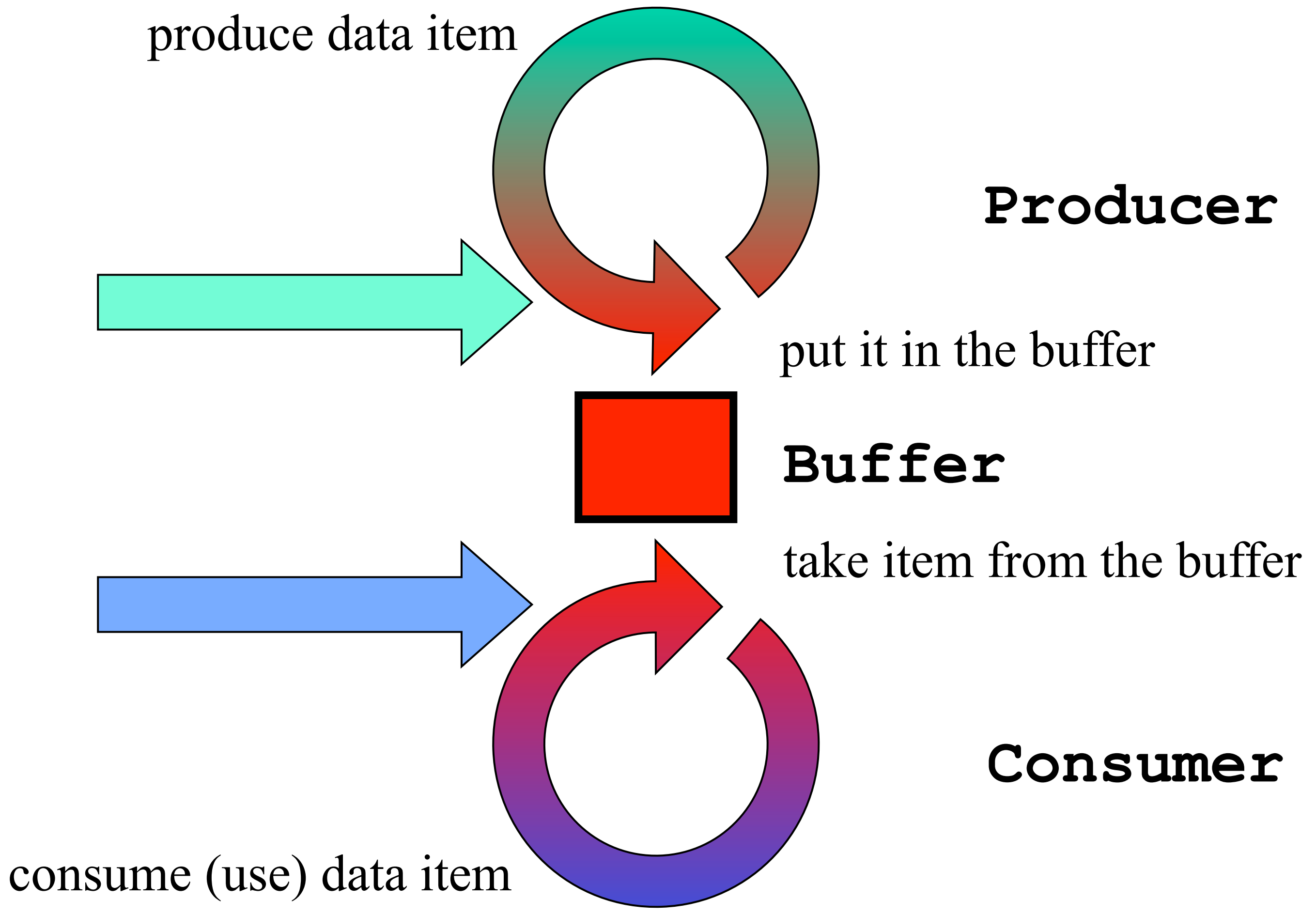
Buffer-based solutions

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.

Interprocess communication



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
Object v = null;
integer in = 0;
while(true) {
    // produce data v
    ...
    buf[in] = v;
    in = in + 1;
    V(n);
}
```

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

```
// 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

Object x = null;
while(true) {
    // produce data x
    ...
    P(s);
    buf = x;
    V(s);
}
```

```
// Consumer process

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

```
// 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

Object x = null;
while(true) {
    // produce data x
    ...
    buf = x;
    V(s);
}
```

```
// Consumer process

Object y = null;
while(true) {
    P(s);
    y = buf;
    // use the data y
    ...
}
```

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

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

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

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

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.

Single element buffer solution

```
// Producer process

Object x = null;
while(true) {
    // produce data x
    ...
    P(empty);
    buf = x;
    V(full);
}
```

```
// Consumer process

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

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

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

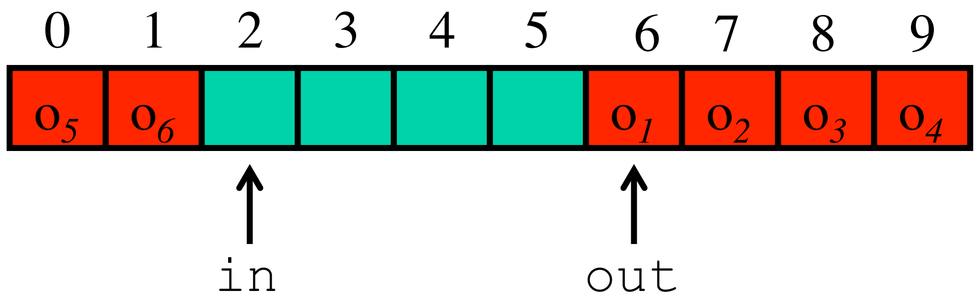
A single element buffer works well if the Producer and Consumer processes run *at the same rate*:

- 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
Object x = null;
integer in = 0;
while(true) {
    // produce data x
    ...
    P(empty);
    buf[in] = x;
    in = (in + 1) % n;
    V(full);
}
```

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

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

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

```
// 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.

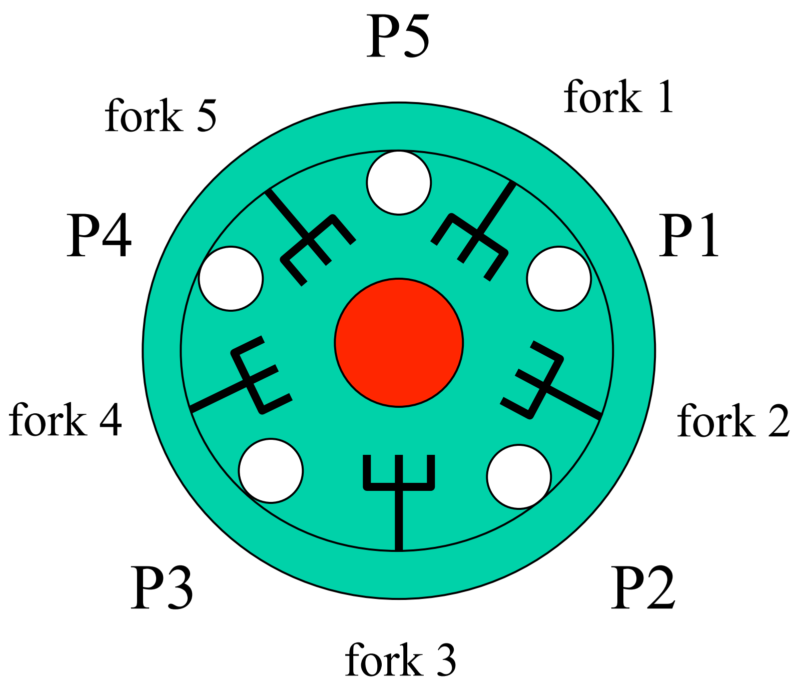
Dining Philosophers Problem

The *Dining Philosophers* problem illustrates mutual exclusion between processes which compete for overlapping sets of shared variables

- five philosophers sit around a circular table
- each philosopher alternately thinks and eats spaghetti from a dish in the middle of the table
- the philosophers can only afford five forks–one fork is placed between each pair of philosophers
- to eat, a philosopher needs to obtain mutually exclusive access to the fork on their left and right

The problem is to avoid *starvation*–e.g., each philosopher acquires one fork and refuses to give it up.

Dining Philosophers Problem



Deadlock in the Dining Philosophers

The key to the solution is to avoid *deadlock* caused by circular waiting:

- process 1 is waiting for a resource (fork) held by process 2
- process 2 is waiting for a resource held by process 3
- process 3 is waiting for a resource held by process 4
- process 4 is waiting for a resource held by process 5
- process 5 is waiting for a resource held by process 1.

No process can make progress and all processes remain deadlocked.

Semaphore Solution

```
// Philosopher i, i == 1-4

while(true) {
    //get right fork then left
    P(fork[i]);
    P(fork[i+1]);
    // eat ...
    V(fork[i]);
    V(fork[i+1]);
    // think ...
}
```

```
// Philosopher 5

while(true) {
    //get left fork then right
    P(fork[1]);
    P(fork[5]);
    // eat ...
    V(fork[1]);
    V(fork[5]);
    // think ...
}
```

```
// Shared variables
binary semaphore fork[5] = {1, 1, 1, 1, 1};
```

Exercise: 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.