Concurrent Algorithms
Lecture 6 - Semaphores

Eugene Kenny

eugkenny.lit@gmail.com

Limerick Institute of Technology

November 6th 2014

Outline

- problems with Peterson's algorithm
- semaphores
- implementing semaphores
- using semaphores
    - for Mutual Exclusion
    - for Condition Synchronisation
- semaphores and Java
- problem solving with semaphores
- solving Producer-Consumer problems using buffers:
    - single element buffer
    - bounded buffer
- Dining Philosophers problem
- exercise: semaphores

Peterson's Algorithm

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

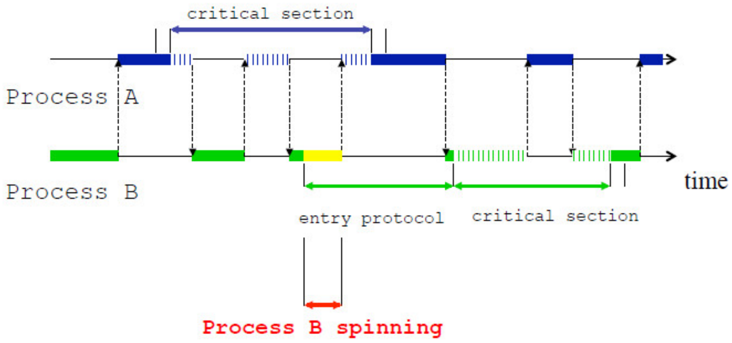
```
// shared variables
bool c1 = c2 = false;
integer turn == 1;
```

Problems with Peterson's Algorithm

Peterson's algorithm is *correct*, however it is complex and inefficient:

- solutions to the Mutual Exclusion problem for *n* processes are quite complex
- it uses busy-waiting (spin locks) to achieve synchronisation, which is often unacceptable in a multiprogramming environment

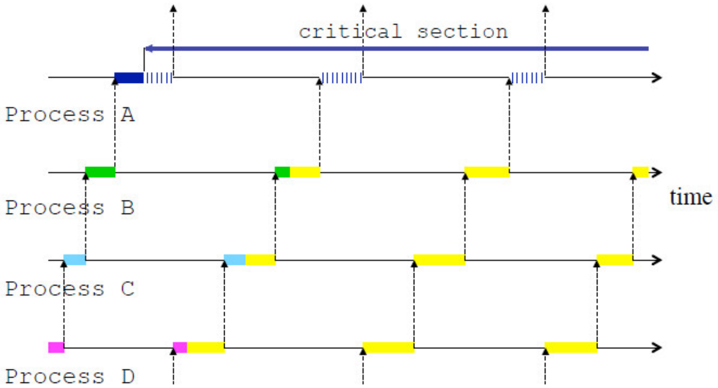
Overhead of Spin Locks



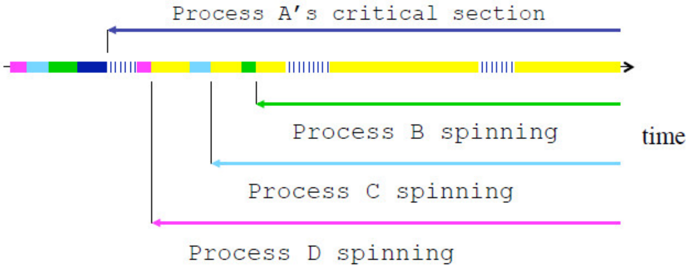
Overhead of Spin Locks

- time spent spinning is necessary to ensure mutual exclusion
- it is also wasted CPU – Process B can do no useful work while Process A is in its critical section
- however, the scheduler doesnt know this, and will (repeatedly) try to run Process B even while process A is in its critical section
- if the critical sections are large relative to the rest of the program, or there are a large number of processes contending for access to the critical section, this will slow down your concurrent program
- e.g., with 10 processes competing to access their critical sections, in the worst case we could end up wasting 90% (or more) of the CPU

Overhead of Spin Locks



Overhead of Spin Locks



Semaphores

A *semaphore* s is an integer variable which can take only non-negative values. Once it has been given its initial value, the only permissible operations on s are the atomic actions:

> $\mathbf{P}(s)$: if $s > 0$ then $s = s - 1$, else *suspend* execution of the process that called $\mathbf{P}(s)$
>
> $\mathbf{V}(s)$: if some process p is suspended by a previous $\mathbf{P}(s)$ on this semaphore then resume p, else $s = s + 1$

A *general semaphore* can have any non-negative value; a *binary semaphore* is one whose value is always 0 or 1.

Note on Terminology

- In some textbooks **P** is called *wait* and **V** is called *signal*
- We will call them **P** and **V** to avoid confusion with two different operations called *wait* and *signal* which are defined on monitors (later lecture)

Semaphores as Abstract Data Types

A semaphore can be seen as an **abstract data type**:

- a set of permissible values; and
- a set of permissible operations on instances of the type.

However, unlike normal abstract data types, we require that the **P** and **V** operations on semaphores be implemented as *atomic actions*.

P and **V** as Atomic Actions

Reading and writing the semaphore value is itself a *critical section*:

- **P** and **V** operations must be mutually exclusive
- e.g., suppose we have a semaphore, *s*, which has the value 1, and two processes simultaneously attempt to execute **P** on *s*:
    - only one of these operations will be able to complete before the next **V** operation on *s*;
    - the other process attempting to perform a **P** operation is suspended.
- Semaphore operations on different semaphores need not be mutually exclusive.

V on Binary Semaphores

- effects of performing a **V** operation on a binary semaphore which has a current value of 1 are implementation dependent:
    - operation may be ignored
    - may increment the semaphore
    - may throw an exception
- we will assume that a **V** operation on a binary semaphore which has value 1 does not increment the value of the semaphore.

Resuming Suspended Processes

Note that the definition of **V** doesn't specify which process is woken up if more than one process has been suspended on the same semaphore

- this has implications for the fairness of algorithms implemented using semaphores and properties like Eventual Entry.
- we will come back to this later ...

Implementing Semaphores

To implement **P** and **V** as atomic actions, we can use any of the mutual exclusion algorithms we have seen so far, e.g.:

- Peterson's algorithm
- special hardware instructions (e.g. Test-and-Set)
- disabling interrupts

There are several ways a processes can be suspended:

- busy waiting – this is inefficient
- blocking: a process is *blocked* if it is waiting for an event to occur without using any processor cycles (e.g., a not-runnable thread in Java).

Using Semaphores

We can think if **P** and **V** as controlling access to a resource:

- when a process wants to use the resource, it performs a **P** operation:
    - if this succeeds, it decrements the amount of resource available and the process continues;
    - if all the resource is currently in use, the process has to wait.
- when a process is finished with the resource, it performs a **V** operation:
    - if there were processes waiting on the resource, one of these is woken up;
    - if there were no waiting processes, the semaphore is incremented indicating that there is now more of the resource free.
    - note that the definition of **V** doesnt specify *which* process is woken up if more than one process has been suspended on the same semaphore.

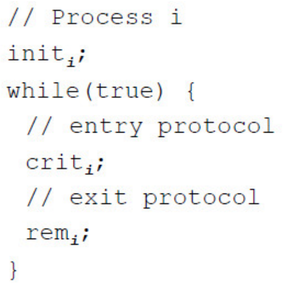
Semaphores for Mutual Exclusion and Condition Synchronisation

Semaphores can be used to solve mutual exclusion and condition synchronisation problems:

- semaphores can be used to implement the entry and exit protocols of mutual exclusion protocols in a straightforward way
- semaphores can also be used to implement more efficient solutions to the condition synchronisation problem

General Form of a Solution

We assume that each of the n processes have the following form, $i = 1, ..., n$



```
// Process i
init$_i$;
while(true) {
  // entry protocol
  crit$_i$;
  // exit protocol
  rem$_i$;
}
```

Mutual Exclusion Using a Binary Semaphore

```
binary semaphore s = 1;    // shared binary
                           // semaphore

// Process i
init_i;
while(true) {
 P(s);                // entry protocol
 crit_i;
 V(s);                // exit protocol
 rem_i;
}
```

Example Trace

```
// Process 1                // Process 2

init1;                      init2;
```

```
s == 1
```

Example Trace

```
// Process 1                 // Process 2

init1;                       init2;
while(true)
```

s == 1

Example Trace

```
// Process 1                        // Process 2

init1;                              init2;
while(true) {
    P(s);



}
```

```
s == 0
```

Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true)
     P(s);
     crit1;



}
```

s == 0

Example Trace

```
// Process 1

init1;
while(true) {
     P(s);
     crit1;



}
```

```
// Process 2

init2;
while(true) {
     P(s);




}
```

s == 0

Example Trace

```
// Process 1                  // Process 2

init1;                        init2;
while(true) {                 while(true) {
     P(s);                         P(s);
     crit1;



}                             }
```

s == 0

Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    P(s);                           P(s);
    crit1;
    V(s);

}                               }
```

```
s == 0
```

Example Trace

```
// Process 1                   // Process 2

init1;                         init2;
while(true) {                  while(true) {
     P(s);                          P(s);
     crit1;                         crit2;
     V(s);
     rem1;
}                              }
```

s == 0

Properties of the Semaphore Solution

The semaphore solution has the following properties:

- **Mutual Exclusion:** yes
- **Absence of Deadlock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** guaranteed for 2 processes; if there are ¿ 2 processes, eventual entry is guaranteed only if the semaphores are *fair*.

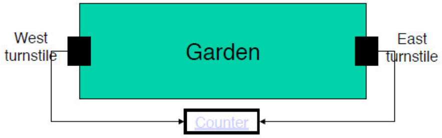
Other Advantages

In addition:

- the semaphore solution works for n processes;
- it is much simpler than an n process solution based on Peterson's algorithm; and
- it avoids busy waiting.

Example: Ornamental Gardens

A large ornamental garden is open to members of the public who can enter through either of two turnstiles.



- the owner of the garden writes a computer program to count how many people are in the garden at any one time
- the program has two processes, each of which monitors a turnstile and increments a shared counter whenever someone enters via that processes' turnstile.

Solving the Ornamental Gardens

```
// East turnstile

init1;
while(true) {
  // wait for turnstile

  count = count + 1;

  // other stuff ...
}
```

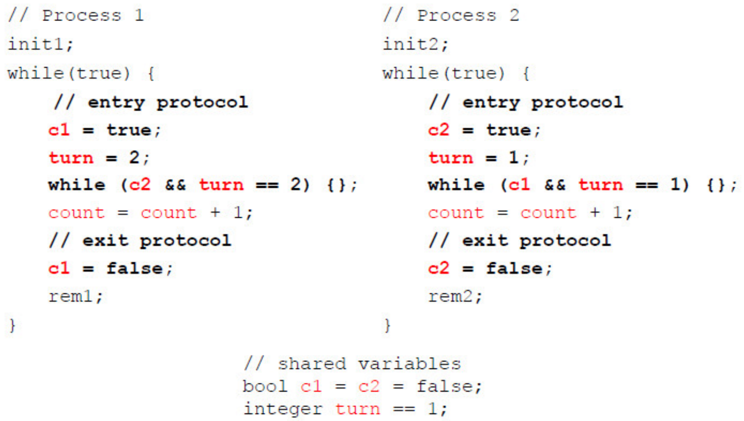
```
// West turnstile

init2;
while(true) {
  // wait for turnstile

  count = count + 1;

  // other stuff ...
}
```

```
integer count == 0
```

Solving the Ornamental Gardens

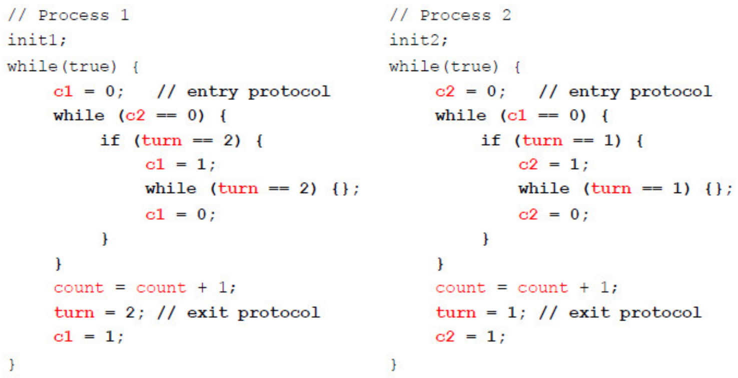
```
// East turnstile              // West turnstile

init1;                         init2;
while(true) {                  while(true) {
  // wait for turnstile          // wait for turnstile
  P(s);                          P(s);
  count = count + 1;             count = count + 1;
  V(s);                          V(s);
  // other stuff ...             // other stuff ...
}                              }
             binary semaphore s == 1
             integer count == 0
```

Comparison with Peterson's Algorithm



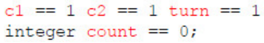
```
// Process 1                       // Process 2
init1;                             init2;
while(true) {                      while(true) {
    // entry protocol                  // entry protocol
    c1 = true;                         c2 = true;
    turn = 2;                          turn = 1;
    while (c2 && turn == 2) {};        while (c1 && turn == 1) {};
    count = count + 1;                 count = count + 1;
    // exit protocol                   // exit protocol
    c1 = false;                        c2 = false;
    rem1;                              rem2;
}                                  }

                  // shared variables
                  bool c1 = c2 = false;
                  integer turn == 1;
```

Comparison with Dekker's Algorithm



```
// Process 1
init1;
while(true) {
    c1 = 0;    // entry protocol
    while (c2 == 0) {
        if (turn == 2) {
            c1 = 1;
            while (turn == 2) {};
            c1 = 0;
        }
    }
    count = count + 1;
    turn = 2; // exit protocol
    c1 = 1;
}
```

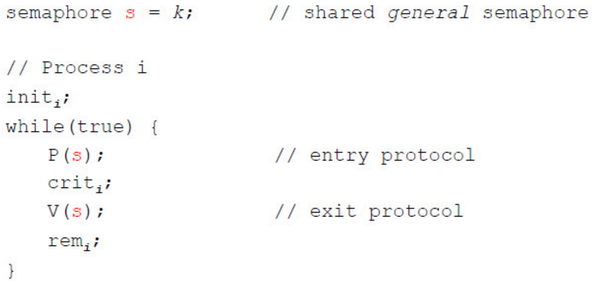
```
// Process 2
init2;
while(true) {
    c2 = 0;    // entry protocol
    while (c1 == 0) {
        if (turn == 1) {
            c2 = 1;
            while (turn == 1) {};
            c2 = 0;
        }
    }
    count = count + 1;
    turn = 1; // exit protocol
    c2 = 1;
}
```



```
c1 == 1 c2 == 1 turn == 1
integer count == 0;
```

Selective Mutual Exclusion with General Semaphores

If we have *n* processes, of which *k* can be in their critical section at the same time:



```
semaphore s = k;        // shared general semaphore

// Process i
init_i;
while(true) {
    P(s);               // entry protocol
    crit_i;
    V(s);               // exit protocol
    rem_i;
}
```

Semaphores and Condition Synchronisation

Condition synchronisation involves delaying a process until some boolean condition is true.

- condition synchronisation problems can be solved using *busy waiting*:
  - the process simply sits in a loop until the condition is true
  - busy waiting is inefficient
- semaphores are not only useful for implementing mutual exclusion, but can be used for general condition synchronisation.

Producer-Consumer with an Infinite Buffer

Given two processes, a **producer** which generates data items, and a **consumer** which consumes them:

- we assume that the processes communicate via an *infinite* shared buffer;
- the producer may produce a new item at any time;
- the consumer may only consume an item when the buffer is not empty; and
- all items produced are eventually consumed.

This is an example of a *Condition Synchronisation* problem: delaying a process until some Boolean condition is true.

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[∞];
semaphore n = 0;
```

Example Trace

```
// Producer process
Object v = null;
```

```
// Consumer process
Object w = null;
```

```
n == 0 buf == []
```

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
```

```
// Consumer process
Object w = null;
integer out = 0;
```

```
n == 0 buf == []
```

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true)
```

```
n == 0 buf == []
```

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);

}
```

n == 0 buf == []

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);





}
```

```
n == 0 buf == []
```

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
while(true)
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);




}
```

```
n == 0 buf == []
```

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
while(true) {
    // produce data v
    ...
    buf[in] = v;


}
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);



}
```

n == 0 buf == [o_1]

Example Trace

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

}
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);




}
```

n == 0 buf == [o_1]

Example Trace

```
// 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);


}
```

n == 0 buf == [o_1]

V with Blocked Processes

Once the Producer has placed an item in the buffer, it performs a **V** operation on the semaphore.

- this wakes up the suspended Consumer, which resumes at the point at which it blocked.
- note that the value of n remains *unchanged* – *n* would only have been incremented by the **V** operation if there were no processes suspended on *n*.

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
while(true) {
    // produce data v
    ...
    buf[in] = v;


}
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);
    w = buf[out];


}
```

n == 0 buf == $[\cancel{o_1}, o_2]$

Example Trace

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

}
```

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


}
```

n == 0 buf == [$\cancel{o_1}$, o_2]

Example Trace

```
// 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;

}
```

n == 1 buf == [$\cancel{o}_1$, o_2]

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
while(true) {
    // produce data v
    ...
    buf[in] = v;


}
```

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

$n == 1 \; buf == [\cancel{o}_1, o_2, o_3]$

Example Trace

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

}
```

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

n == 1 buf == $[\cancel{o_1}, o_2, o_3]$

Example Trace

```
// 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
    ...
}
```

n == 2 buf == $[\cancel{o_1}, o_2, o_3]$

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
while(true) {
    // produce data v
    ...


}
```

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

n == 2 buf == [$\cancel{o_1}$, o_2 , o_3]

Example Trace

```
// Producer process
Object v = null;
integer in = 0;
while(true) {
    // produce data v
    ...



}
```

```
// Consumer process
Object w = null;
integer out = 0;
while(true) {
    P(n);



}
```

n == 1 buf == $[\cancel{o}_1, o_2, o_3]$

Semaphores in Java

- as of Java 5, Java provides a `Semaphore` class in the package `java.util.concurrent`
- supports **P** and **V** operations (called `acquire()` and `release()` in the Java implementation)
- the constructor optionally accepts a *fairness* parameter
  - if this is false, the implementation makes no guarantees about the order in which threads are awoken following a `release()`
  - if *fairness* is true, the semaphore guarantees that threads invoking any of the acquire methods are processed first-in-first-out (FIFO)
- Java implementation of semaphores is based on higher-level concurrency constructs called monitors

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:

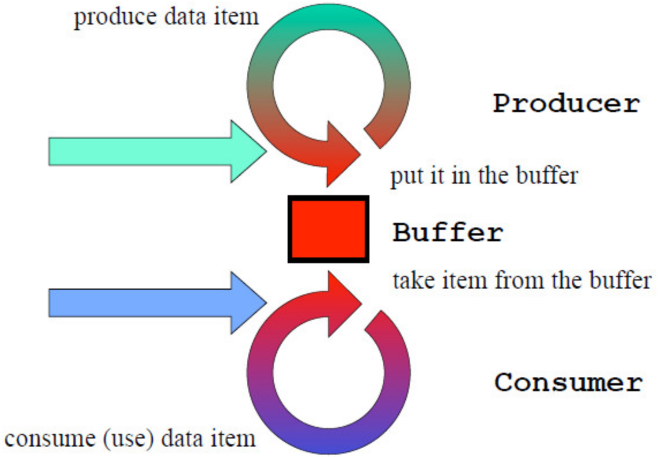
- multiple producers – single consumer
- single producer – multiple consumers
- multiple producers – multiple consumers

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;
- alertthe 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 ProducerConsumer 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:** alertno
- **data items are read only once:** alertno
- **data items are not overwritten before they are read**: alertno
- **items are consumed in the order they are produced:** yes
- **all items produced are eventually consumed:** alertno

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
- **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

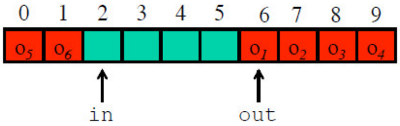
A single element buffer works well if the Producer and Consumer processes run *at the same rate*:

- processes dont 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 ProducerConsumer 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.

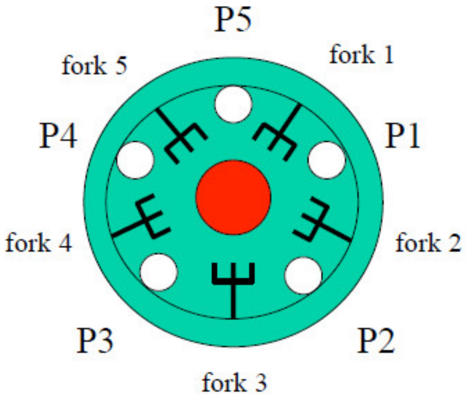
Dining Philosophers Problem

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          // Philosopher 5

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