G52CON:
Concepts of Concurrency

Lecture 11: Semaphores I

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

Outline of this lecture

- problems with Peterson's algorithm
- semaphores
- implementing semaphores
- using semaphores
  - for Mutual Exclusion
  - for Condition Synchronisation
- semaphores and Java

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;
}
```

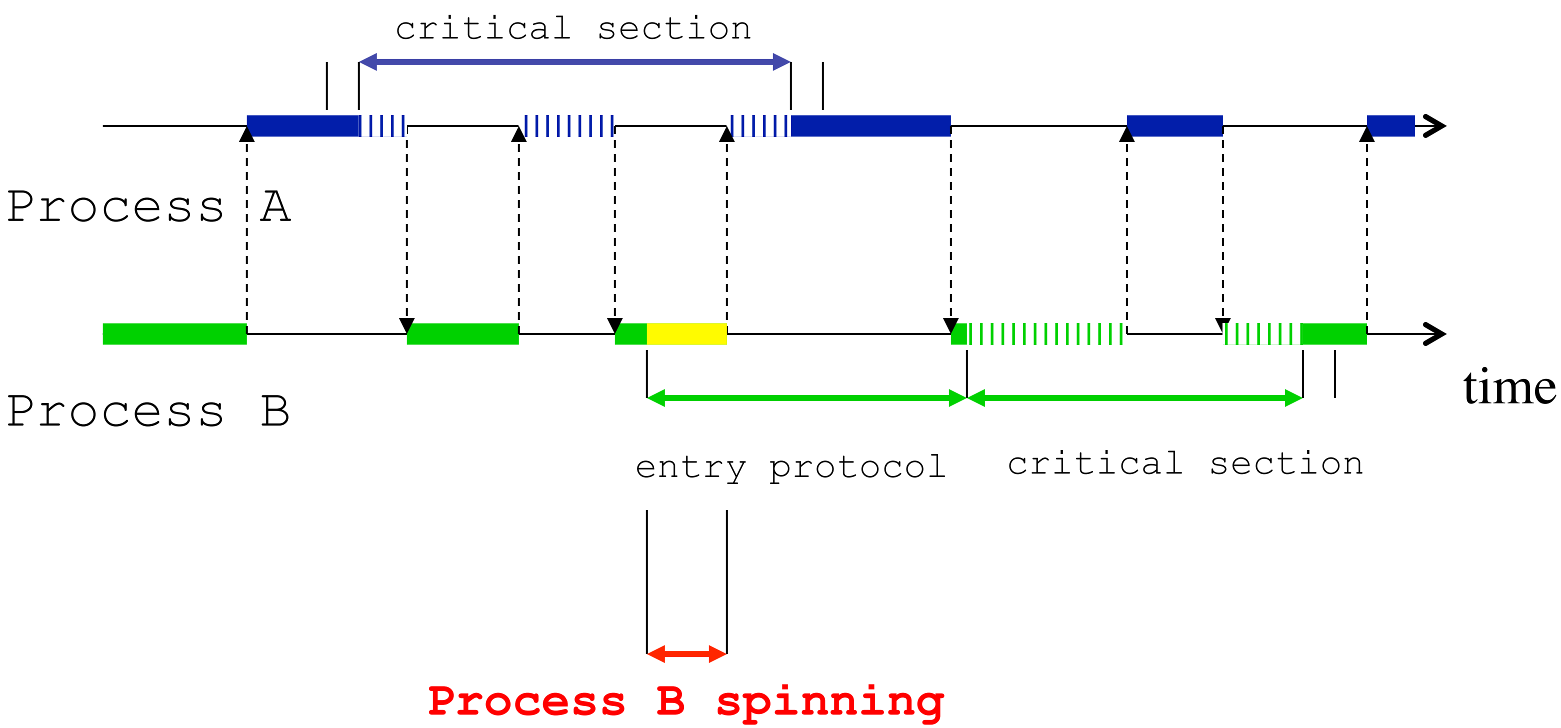
```
// shared variables
bool c1 = c2 = false;
integer turn == 1;
```

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

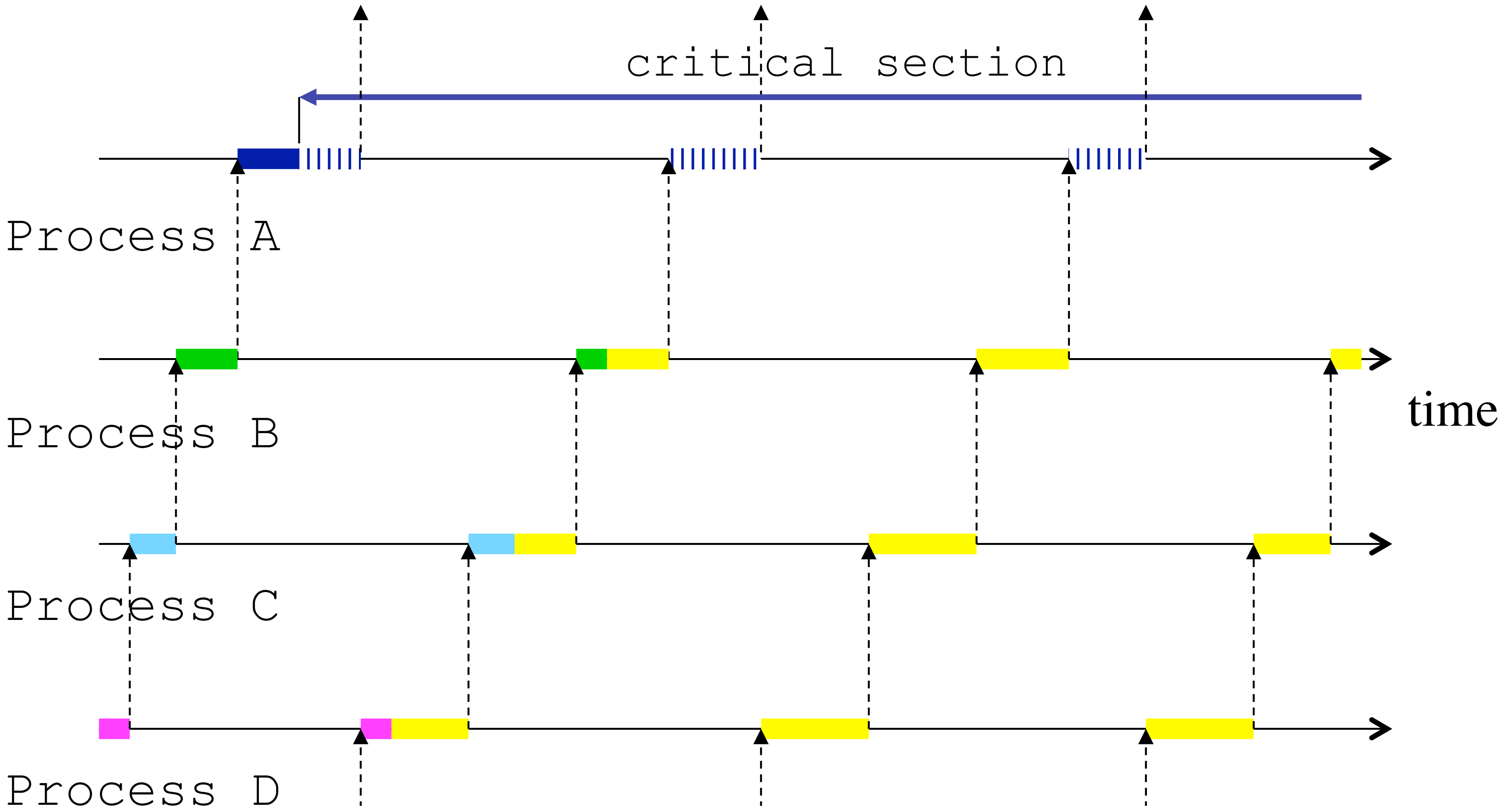
Overhead of spin locks



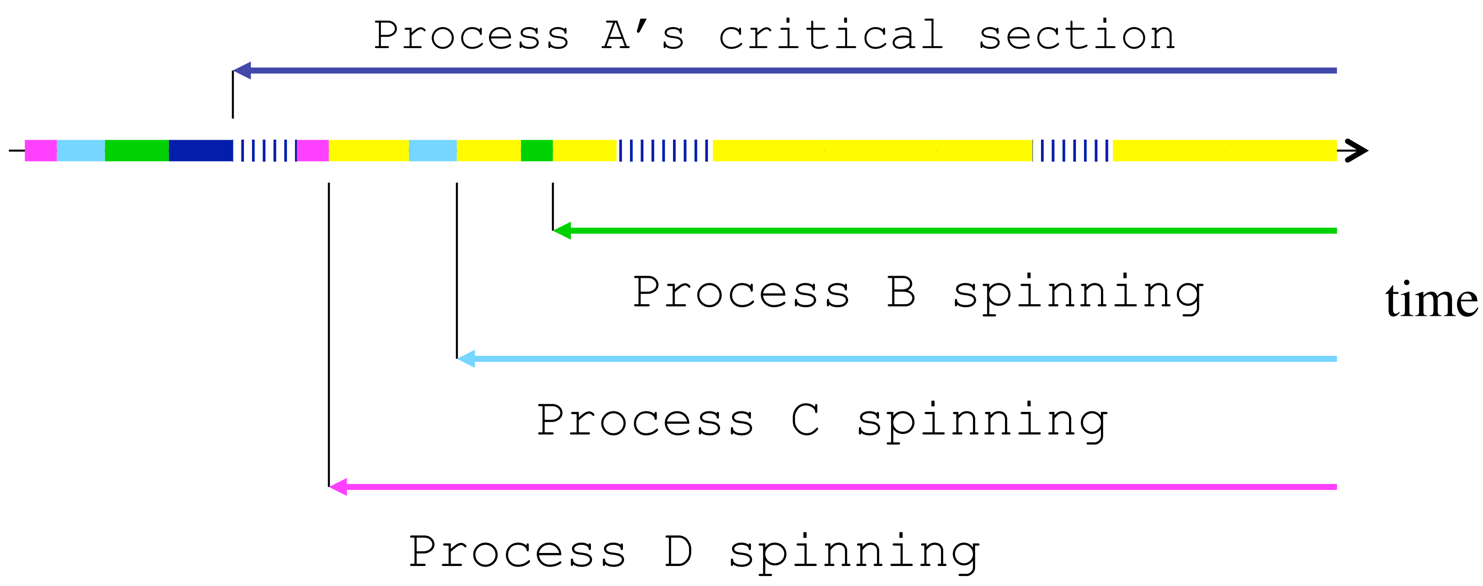
Overhead of spin locks

- 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 doesn't 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:

P(s) : if $s > 0$ then $s = s - 1$, else *suspend* execution of the process that called ***P***(s)

V(s) : if some process p is suspended by a previous ***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 and ***V*** is called *signal*
- I'll 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 doesn't 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;
}
```

An example trace 1

```
// Process 1                    // Process 2

init1;                          init2;
```

`s == 1`

An example trace 2

```
// Process 1                    // Process 2

init1;                          init2;
while(true)
```

s == 1

An example trace 3

```
// Process 1                    // Process 2

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



}
```

s == 0

An example trace 4

```
// Process 1                  // Process 2

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



}
```

s == 0

An example trace 5

```
// Process 1

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

}
```

```
// Process 2

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

}
```

`s == 0`

An example trace 6

```
// Process 1                    // Process 2

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



}                               }
```

s == 0

An example trace 7

```
// Process 1

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

}
```

```
// Process 2

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



}
```

`s == 0`

An example trace 8

```
// Process 1

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

```
// Process 2

init2;
while(true) {
    P(s);
    crit2;


}
```

`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*.

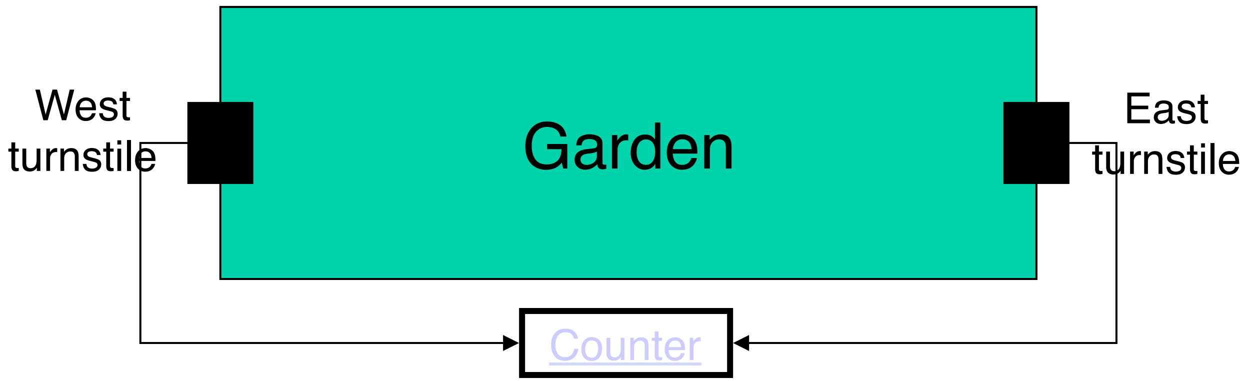
Other advantages

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

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

```
// West turnstile

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

```
binary semaphore s == 1
integer count == 0
```

Comparison with Peterson's algorithm

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

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

An example trace 1

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

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

`n == 0 buf == []`

An example trace 2

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

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

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

An example trace 3

```
// Producer process

Object v = null;
integer in = 0;
```

```
// Consumer process

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

`n == 0 buf == []`

An example trace 4

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

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



}
```

`n == 0 buf == []`

An example trace 5

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

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



}
```

`n == 0 buf == []`

An example trace 6

```
// 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 == []

An example trace 7

```
// 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]

An example trace 8

```
// 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]

An example trace 9

```
// 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`.

An example trace 10

```
// 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 == [~~o_1~~, o_2]

An example trace 11

```
// 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 == [X1, o2]`

An example trace 12

```
// 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 == [~~o~~$_1$, o$_2$]

An example trace 13

```
// 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 == $[X_1, o_2, o_3]$

An example trace 14

```
// 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 == [~~o_1~~, o_2 , o_3]

An example trace 15

```
// 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 == [X_1, o_2 , o_3]

An example trace 16

```
// 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 == [~~o_1~~, o_2 , o_3]

An example trace 17

```
// 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 == [~~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

The next lecture

Semaphores II

Suggested reading:

- Andrews (2000), chapter 4, sections 4.1–4.2;
- Ben-Ari (1982), chapter 4;
- Burns & Davies (1993), chapter 6.