

# G52CON: Concepts of Concurrency

## Lecture 7: Semaphores I

Brian Logan  
School of Computer Science & IT  
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                                // 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) {};
    crit1;                                  crit2;
    // exit protocol                       // exit protocol
    c1 = false;                            c2 = false;
    rem1;                                  rem2;
}                                           }

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

© Brian Logan 2007

G52CON Lecture 7: Semaphores

3

## Problems with Peterson's algorithm

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

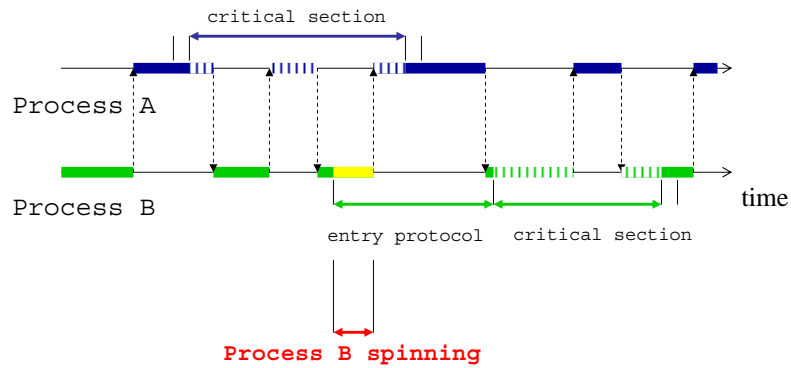
- solutions to the Mutual Exclusion problem for  $n$  processes are quite complex
- it uses busy-waiting (spin locks) to achieve synchronisation, which is often unacceptable in a multiprogramming environment

© Brian Logan 2007

G52CON Lecture 7: Semaphores

4

## Overhead of spin locks



© Brian Logan 2007

G52CON Lecture 7: Semaphores

5

## Overhead of spin locks

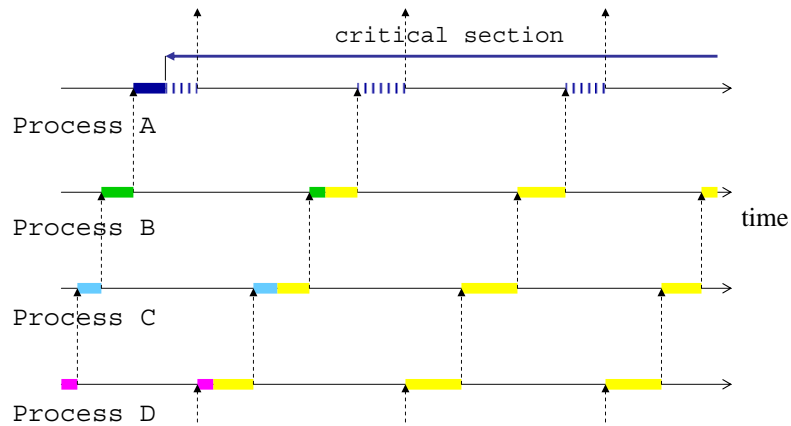
- time spent spinning is necessary to ensure mutual exclusion
- it is also wasted CPU—Process B can do no useful work while Process A is in its critical section
- however, the scheduler doesn't know this, and will (repeatedly) try to run Process B even while process A is in its critical section
- if 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

© Brian Logan 2007

G52CON Lecture 7: Semaphores

6

## Overhead of spin locks

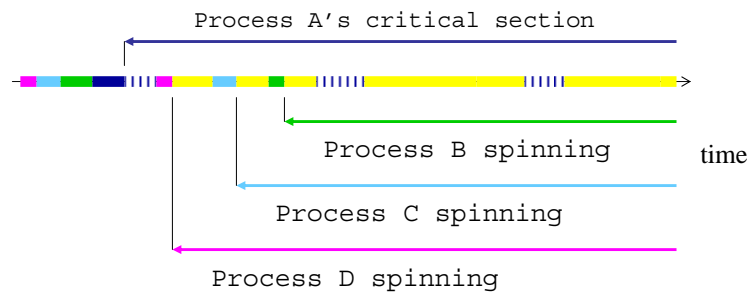


© Brian Logan 2007

G52CON Lecture 7: Semaphores

7

## Overhead of spin locks



© Brian Logan 2007

G52CON Lecture 7: Semaphores

8

# 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.

# 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 distinct semaphores need not be mutually exclusive.

## 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:

- 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 of  $P$  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,

$i = 1, \dots, n$

```
// Process i
initi;
while(true) {
  // entry protocol
  criti;
  // exit protocol
  remi;
}
```

## Mutual exclusion using a binary semaphore

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

// Process i
initi;
while(true) {
    P(s);                // entry protocol
    criti;
    V(s);                // exit protocol
    remi;
}
```

## 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.

## Solving the Ornamental Gardens

```
// East turnstile          // West turnstile

// initialisation        // initialisation
while(true) {            while(true) {
    // wait for turnstile    // wait for turnstile

    count = count + 1;      count = count + 1;

    // other stuff ...      // other stuff ...
}

integer count == 0
```

© Brian Logan 2007

G52CON Lecture 7: Semaphores

28

## 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
initi;
while(true) {
    P(s);                  // entry protocol
    criti;
    V(s);                  // exit protocol
    remi;
}
```

© Brian Logan 2007

G52CON Lecture 7: Semaphores

32

## 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           // 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[∞];
semaphore n = 0;
```

© Brian Logan 2007

G52CON Lecture 7: Semaphores

35

## 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

© Brian Logan 2007

G52CON Lecture 7: Semaphores

54

## 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.