Classical concurrency control: topic overview 1
In these lectures we consider shared writeable data in main memory

Controlling access by concurrent processes to shared writeable data has been studied *as part of OS design* since the earliest OSs (1960s onwards).

Concurrent programming languages brought the same problems to *application programming*. For example, web servers have to handle large numbers of concurrent requests.

Our starting point:
critical regions: regions of code in parallel processes that read or write shared writeable data
implementing critical regions

- without blocking processes (they “spin-lock” or “busy-wait”)
- blocking processes that must wait, on semaphores

Classical, shared memory, concurrency control: topic overview 2

We then look at how semaphores can be used:

1. a single semaphore used to achieve mutual exclusion
2. a single semaphore used to achieve condition synchronisation
3. a single semaphore used for N-resource allocation

Then, how semaphores are implemented. This may be within the OS, or at application level, in the runtime system of a concurrent programming language

Programming with semaphores, using several semaphores to achieve both mutual exclusion and condition synchronisation

1. single producer, single consumer processes communicating via a shared buffer
2. many producers and consumers
3. readers and writers: multiple readers, single writer concurrency control

Classical, shared memory, concurrency control: topic overview 3

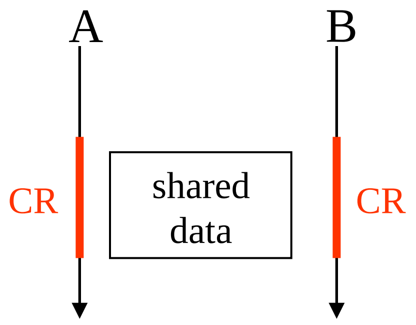
- Discussion of semaphore programming – problems and difficulties

- Concurrency control constructs in programming languages
  Can concurrent programming languages make concurrent programming easier than semaphore programming?
  Can the problems be solved, ameliorated (improved upon), or alleviated (made easier to bear)?
  We look at a number of different approaches in programming languages.

- Concurrent *composite operations* in main memory, introducing the notion that a single, meaningful, high-level, operation may involve several separate low-level operations.

- Lock-free programming – is it possible/easier/more-efficient to program without locks?

Critical regions

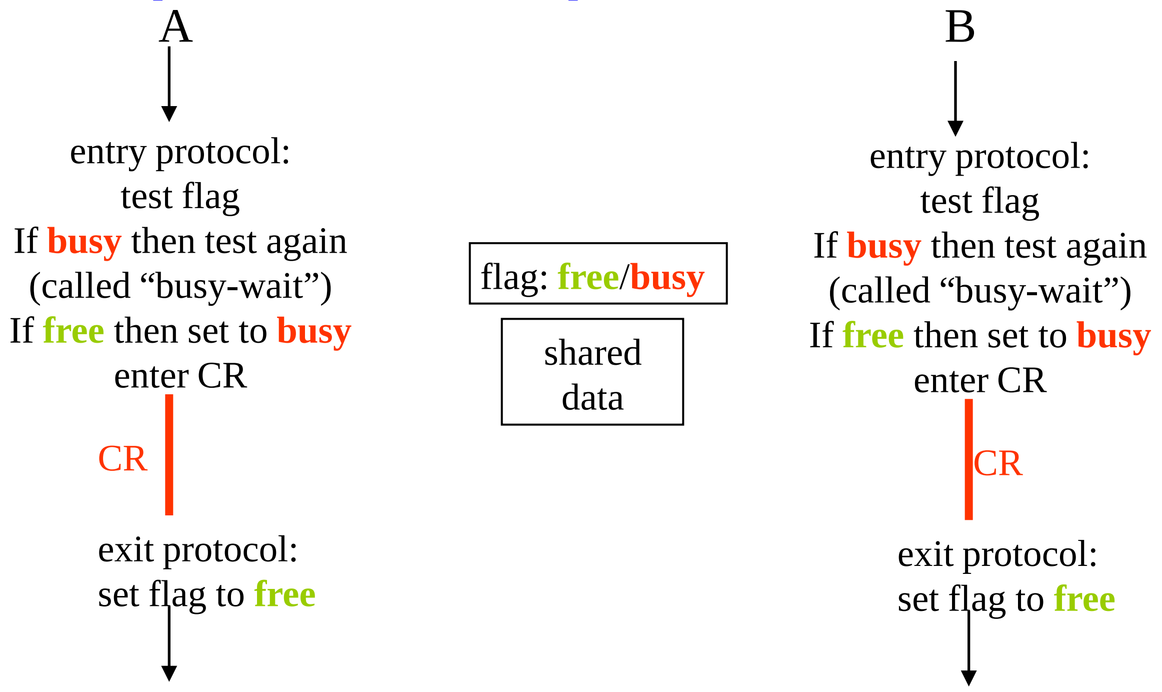


Processes A and B contain critical regions (CRs)
(code that reads or writes this shared data)

CRs are needed only if the data is writeable

A CR is associated with some specific shared data

How can CRs be implemented? – first attempt:



Indivisible test-and-set

The entry protocol is correct only if test and set of flag are atomic/indivisible – HOW?

- forbid interrupts? – NO – this would only work on a uniprocessor, and even then would be inappropriate for general use.
- machine instruction? - YES
- program only – no hardware exclusion - ?

CISC machines had many read-memory, test result, store-to-memory types of instruction

RISC (load/store) architectures may only use a single memory access per instruction

read-and-clear will work:

flag=0 //shared data is busy

flag=1 //shared data is free (initial value)

entry protocol:

read-and-clear, register flag

// if value in register is 0, shared data was busy so retry

// if value in register is 1, shared data was free and you claimed it

// can also be used for condition synchronisation – see later

Multicore machines have atomic instructions e.g. x86 LOCK instruction prefix

Mutual exclusion without hardware support

This was a hot topic in the 1970s and 80s.
Examples for N-process mutual exclusion are:

Eisenberg M. A. and McGuire M. R.,
Further comments on Dijkstra's concurrent programming control problem
CACM 15(11), 1972

Lamport L
A new solution to Dijkstra's concurrent programming problem
CACM, 17(8), 1974
(his N-process bakery algorithm)

For uniprocessors and multiprocessors these algorithms impose large overhead.
In practice, OSs built mutual exclusion on atomic instructions.

With multi-core instruction reordering it is not proven that such programs are correct.

Dijkstra THE 1968

The entry protocols above involve busy-waiting (retry if flag is busy), wasting CPU time

It is better to block a waiting process

Define a new type of variable – semaphore

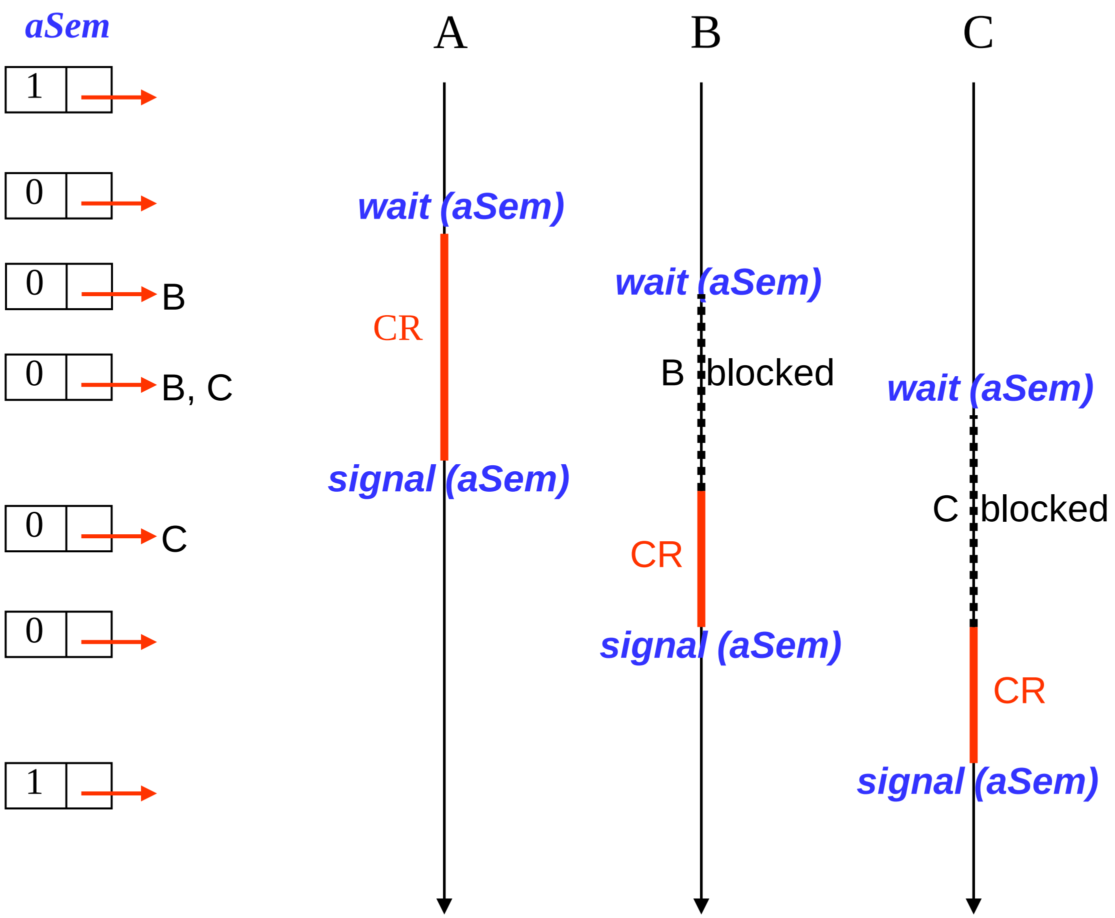
Operations for the type are:

```
wait (aSem)
    if aSem > 0 then aSem = aSem – 1
                else suspend the executing process waiting on aSem

signal (aSem)
    if there are no processes waiting on aSem
        then aSem = aSem + 1
        else free one waiting process – continues after its wait instruction
```

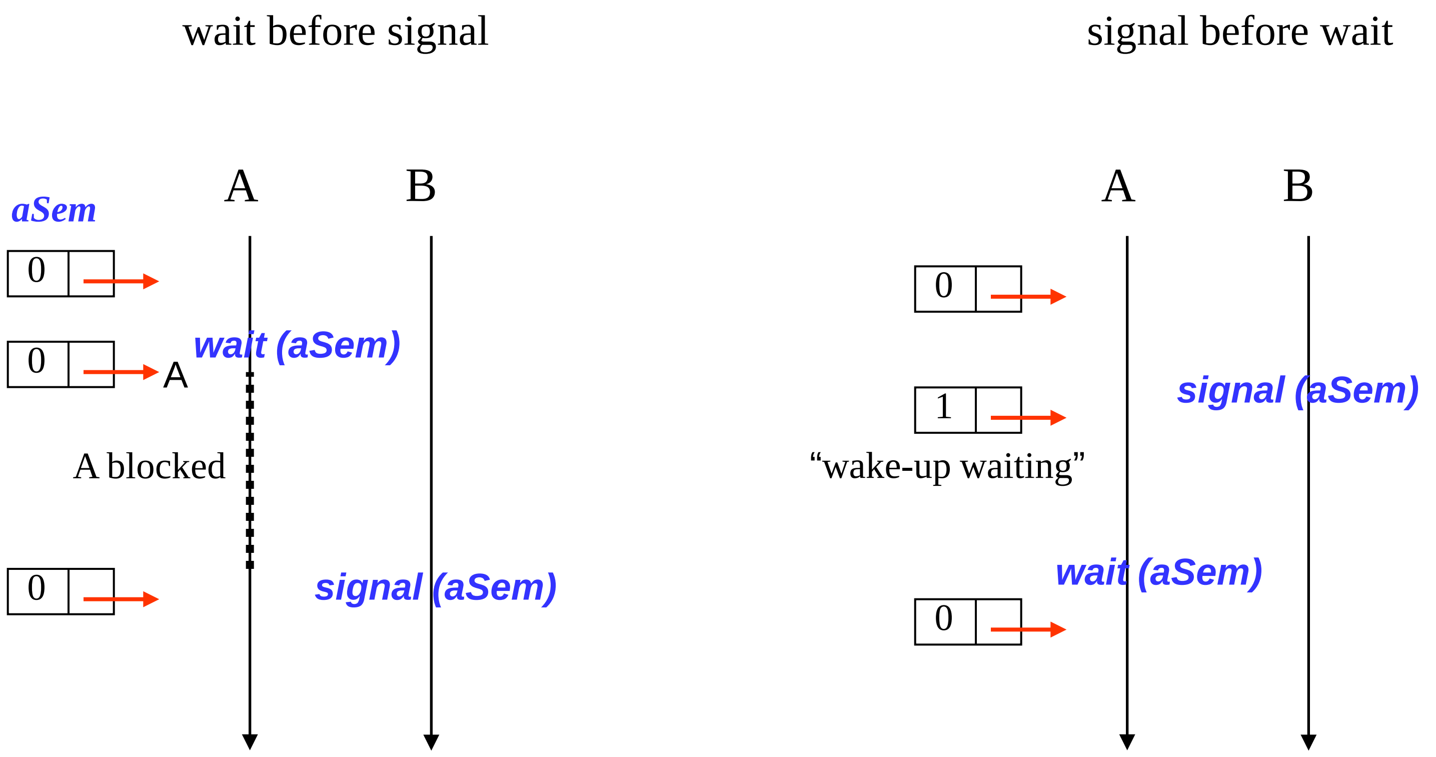
Implementation: an integer and a queue

Mutual exclusion using a semaphore

concurrent processes: **serialisation** of critical regions



Two-process synchronisation



N-resource allocation using a semaphore

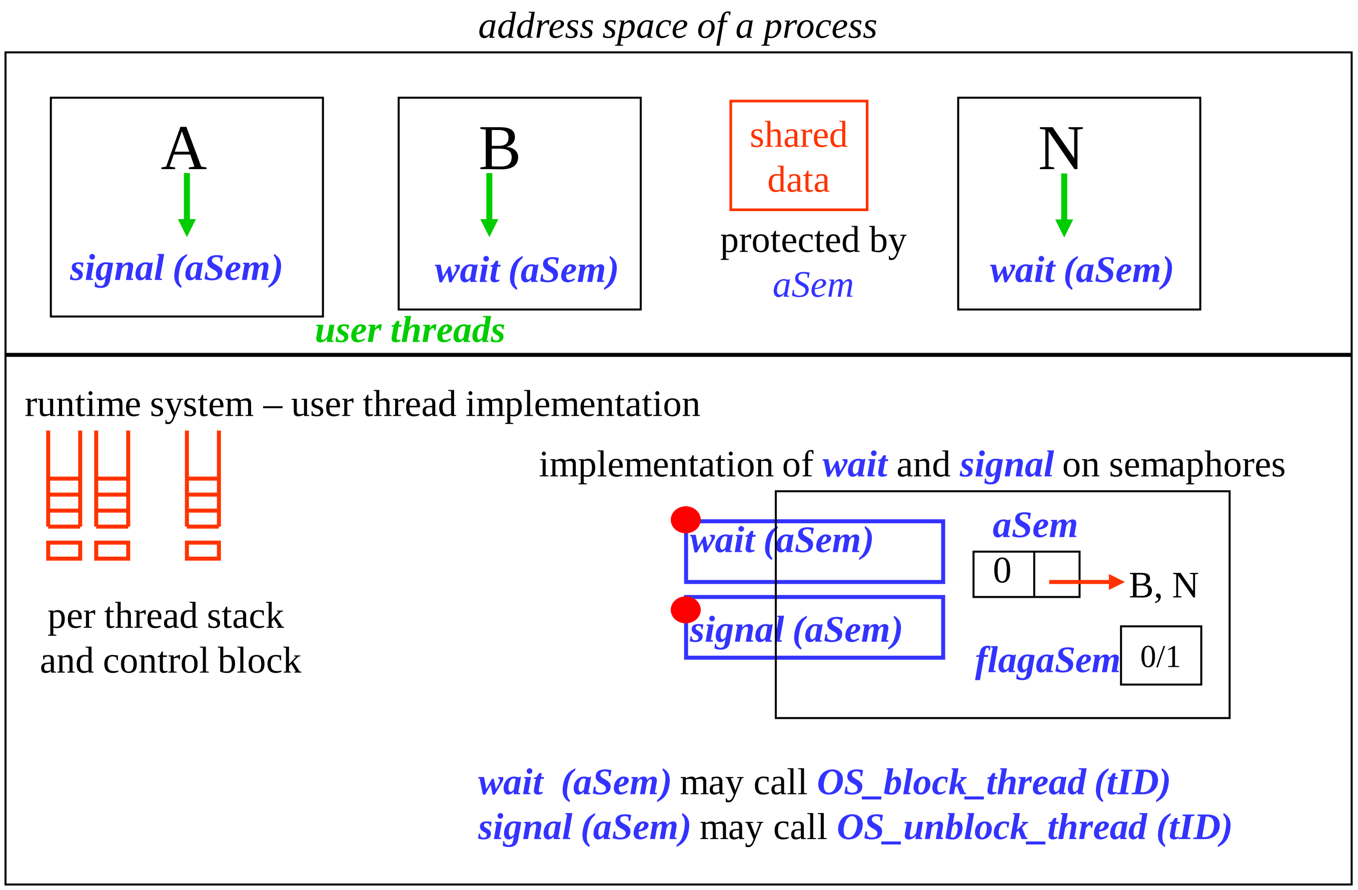
Suppose there are N instances of a resource.

Control its allocation using a semaphore ***resSem*** initialised to N.

Each time a process executes ***wait (resSem)*** the semaphore’s value is decremented.

When the value is 0, after N ***wait***s, all subsequent processes executing ***wait (resSem)*** are queued on it until freed by a current user of the resource executing ***signal (resSem).***

Implementation of semaphores - 1


address space of a process
A
signal (aSem)
B
wait (aSem)
shared data
protected by *aSem*
N
wait (aSem)
user threads
runtime system – user thread implementation
per thread stack and control block
implementation of *wait* and *signal* on semaphores
wait (aSem)
signal (aSem)
aSem
0
B, N
flagaSem
0/1
wait (aSem) may call *OS_block_thread (tID)*
signal (aSem) may call *OS_unblock_thread (tID)*

Implementation of semaphores -2

For user-threads only (OS sees a single-threaded process) the runtime system does all semaphore and user thread management

When user threads are mapped to kernel threads, ***wait*** and ***signal*** must themselves be ***atomic operations.*** This is clearly the case for a multiprocessor, and also for a uniprocessor with preemptive scheduling**.**

Associate a flag with each semaphore, and use an atomic instruction such as ***read-and-clear*****.**

This also applies to kernel threads executing the OS and using OS- managed semaphores for mutual exclusion and condition synchronisation.

The need for concurrency control first came from OS design. We now have concurrent programming languages and OSs support multi-threaded processes.

Semaphore programming

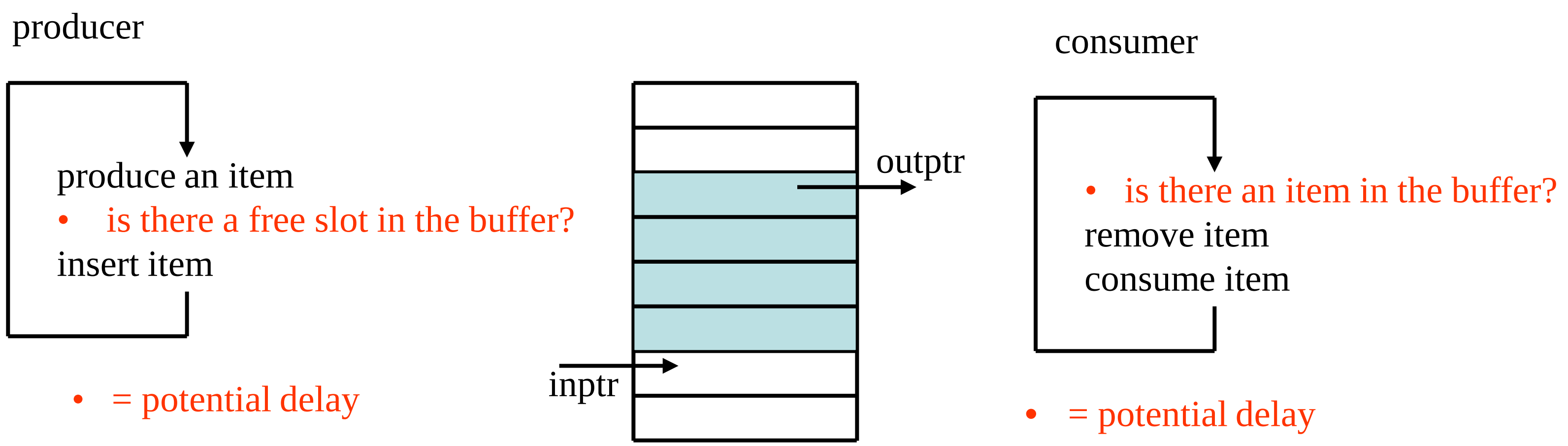
We now develop some concurrent programs that use a number of semaphores for mutual exclusion and condition synchronisation.

1. Two processes communicate through an N-slot cyclic buffer.
   One process inserts, the other removes, records of fixed size.
   Condition synchronisation is needed for when the buffer is full and empty.

2. We now have any number of producer and consumer processes communicating via the buffer. We now need to ensure mutually exclusive access to the buffer.

3. We note that processes that only read shared data can read simultaneously, whereas a process that writes must have exclusive access to the data.
   We develop a solution that gives priority to writers over readers,
   on the assumption that writers are keeping the data up-to-date.

N-slot cyclic buffer, single producer and consumer - 1

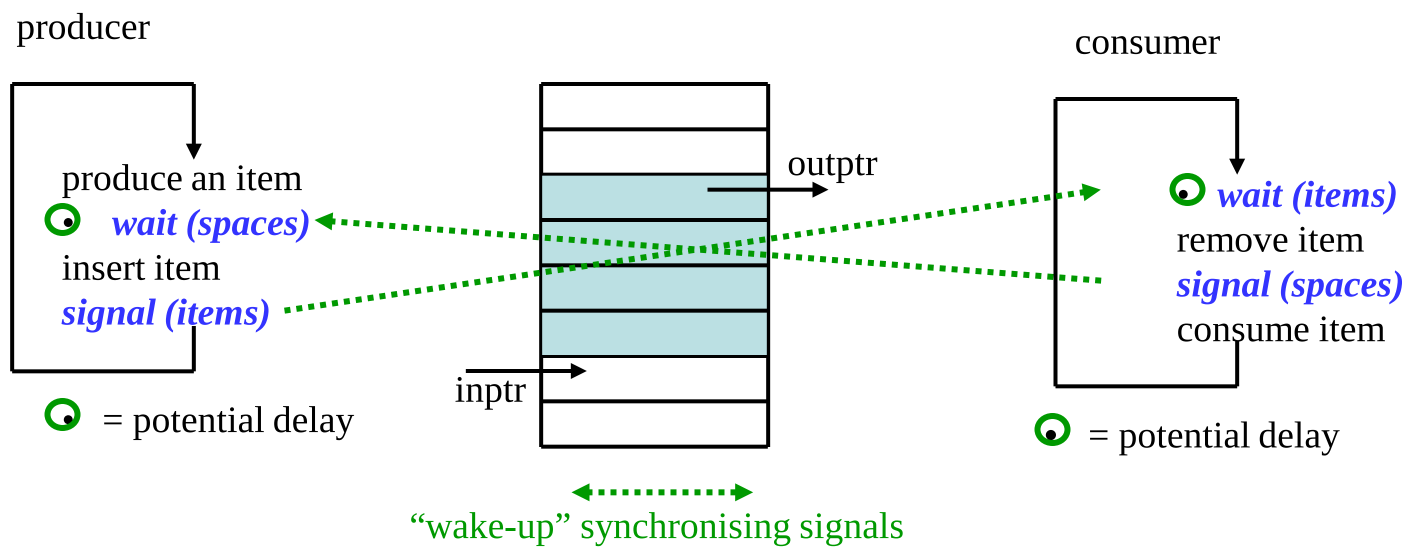


two semaphores are needed

- for the producer to block on when the buffer is full
- for the consumer to block on when the buffer is empty
- note: blocked processes must be unblocked via signals on semaphores

N-slot cyclic buffer, single producer and consumer - 2

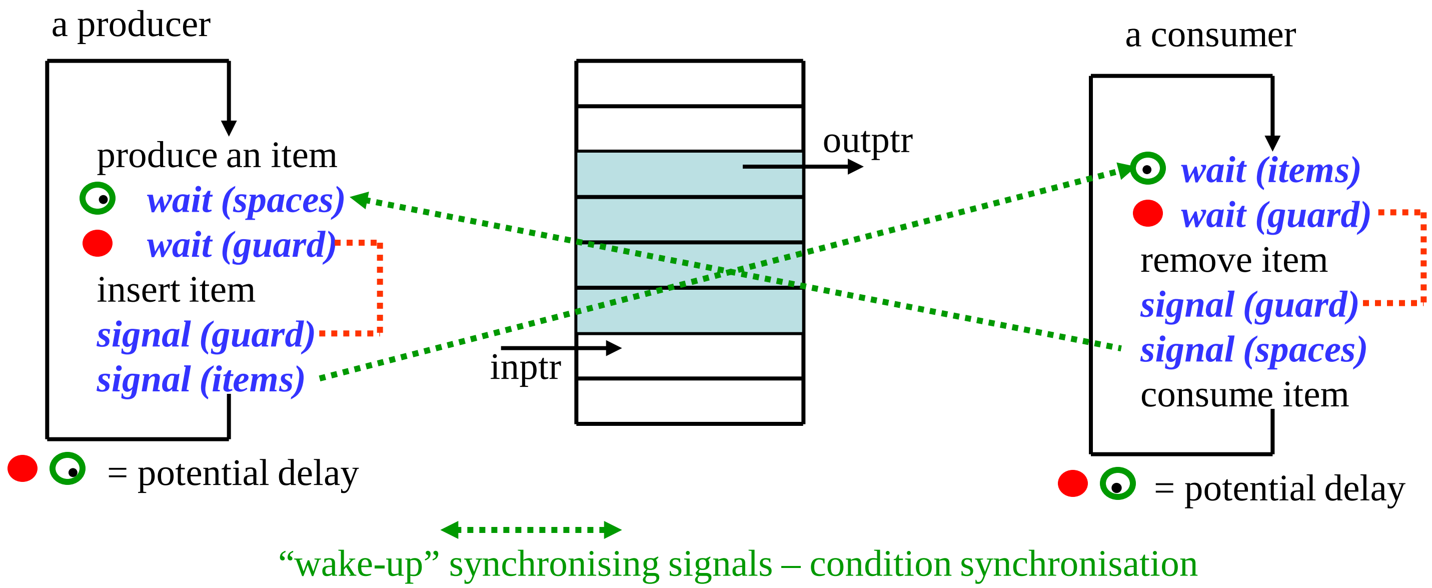
details of pointer manipulation are not shown – we focus on condition synchronisation



two semaphores are needed

- for the producer to block on when the buffer is full
  spaces = N // initially N spaces in buffer
- for the consumer to block on when the buffer is empty
  items = 0 // initially no items in buffer

N-slot cyclic buffer, many producers and consumers



three semaphores are used

- for the producer to block on when the buffer is full
  spaces = N // initially N spaces in buffer
- for the consumer to block on when the buffer is empty
  items = 0 // initially no items in buffer
- to ensure mutually exclusive access to the buffer ●
  guard = 1 // initially the buffer is free

variation: – allow one producer and one consumer to access the buffer in parallel – left as an exercise

Multiple readers, single writer concurrency control -1

Many readers may read simultaneously, a writer must have exclusive access
Assume writers have priority – to keep the data up-to-date.

counts:

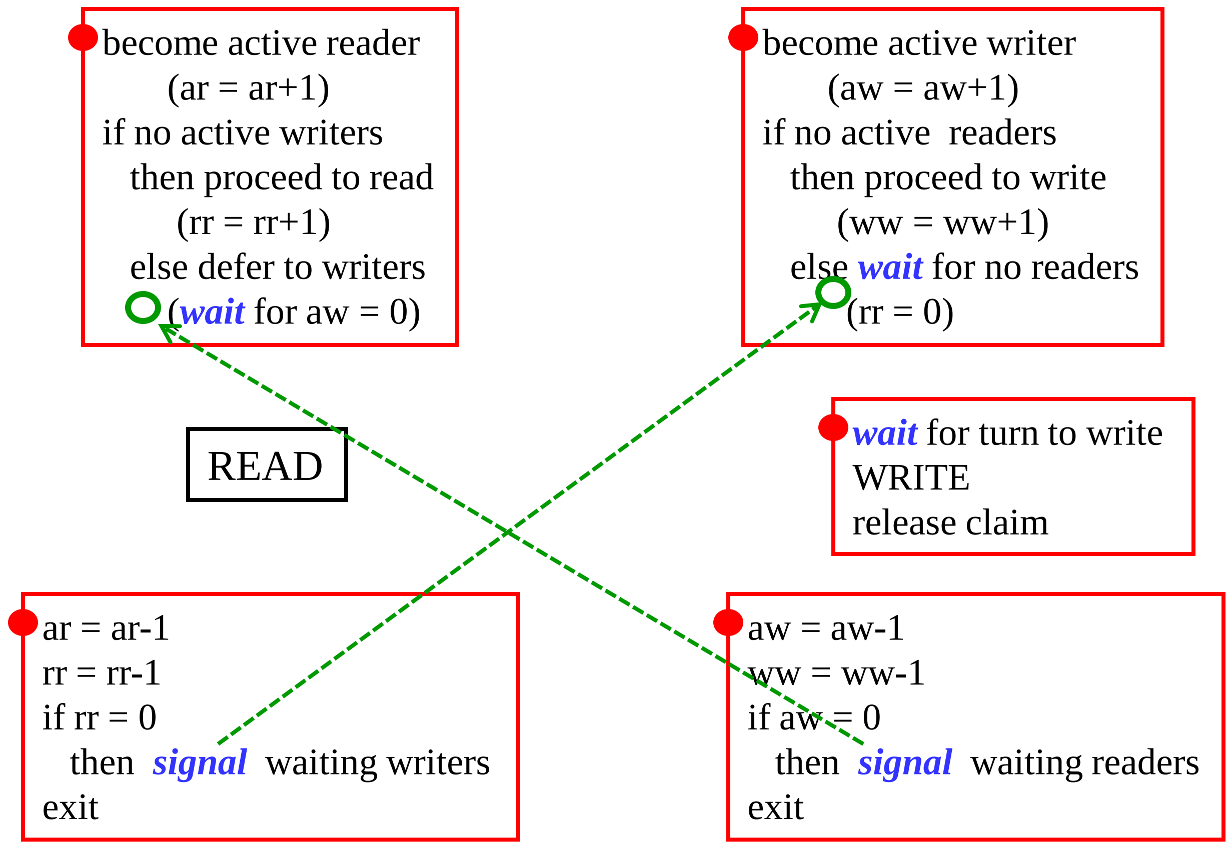
ar = active readers
rr = reading readers (active readers who have proceeded to read)
aw = active writers
ww = writing writers (active writers who have proceeded to write)
but they must wait to write one-at-a-time

Semaphores are needed:

for mutual exclusion ●
1. to test and update the above counts under exclusion
2. to ensure writers write under exclusion

for condition synchronisation ○
1. readers must wait for aw = 0 and must be woken up after blocking
2. writers must wait for rr = 0 and must be woken up after blocking

Multiple readers, single writer concurrency control -2



```
● become active reader
      (ar = ar+1)
  if no active writers
    then proceed to read
      (rr = rr+1)
    else defer to writers
    ○ (wait for aw = 0)
```

```
READ
```

```
● ar = ar-1
  rr = rr-1
  if rr = 0
    then signal waiting writers
  exit
```

```
● become active writer
      (aw = aw+1)
  if no active  readers
    then proceed to write
      (ww = ww+1)
    else wait for no readers
    ○ (rr = 0)
```

```
● wait for turn to write
  WRITE
  release claim
```

```
● aw = aw-1
  ww = ww-1
  if aw = 0
    then signal waiting readers
  exit
```

● mutual exclusion - to access shared counts
- for write access

-----> ○ condition synchronisation

Multiple readers, single writer concurrency control - 3

Complete the program as an exercise. Solutions are in textbooks.

Note that a ***signal*** unblocks only one blocked process. The values of the counts indicate how many signals to send. The last writing writer must unblock all blocked readers. The last reading reader must unblock all waiting writers.

Take care not to ***wait*** while holding the semaphore that protects the shared counts. That would cause deadlock.

Semaphores - discussion

Semaphores are a widely used mechanism underlying concurrency control in operating systems and concurrent programs

Difficult for programmers to use correctly – programs are complex

- can forget to wait and corrupt data
- can forget to signal and cause deadlock

Unconditional commitment to block

- but can fork new threads for concurrent activity.

Unbounded delay on wait.

Priority inversion and convoy effect (see 30 for further discussion)

- low priority process with lock can hold up higher priority processes (note scheduling)
- a long lock-hold can hold up a lot of potentially short ones.

Programming language support

We now follow two developments for concurrency control in shared memory.

1. Programming language support for concurrency control
   Concurrent programming languages provide higher level constructs, implemented using semaphores. We follow the historical evolution:
   passive objects: critical regions and conditional critical regions,
   monitors (Modula 1, Modula 3, Mesa, ….)
   (mutexes and condition variables (pthreads package) not covered)
   synchronized methods and wait/notify (Java)
   active objects: guarded commands, Ada select/accept and rendezvous

2. Concurrent composite operations in main memory.
   We started from ensuring exclusive access to a single item of shared data.
   In general, programmers need to create operations that involve related operations on multiple data items.
   The last two examples we saw both used several semaphores
   producers/consumers involved multiple buffer slots
   readers/writers involved the resource and various integer counts.

We then consider an alternative approach to achieving concurrency control:
lock-free programming.

Critical regions and conditional critical regions

Critical regions and conditional critical regions (CCRs) were proposed as a means of hiding the complexity of semaphore programming.

var v: shared <data-structure> \\ compiler assigns a semaphore, initially 1
region v do begin …… end \\ compiler inserts semaphore operations

But this is only mutual exclusion. Condition synchronisation was added to CCRs
by including
await < some condition on shared data >
The implementation allows the region to be temporarily left/freed if the condition is false
and the process executing ***await*** blocked until the condition becomes true (and its turn comes).

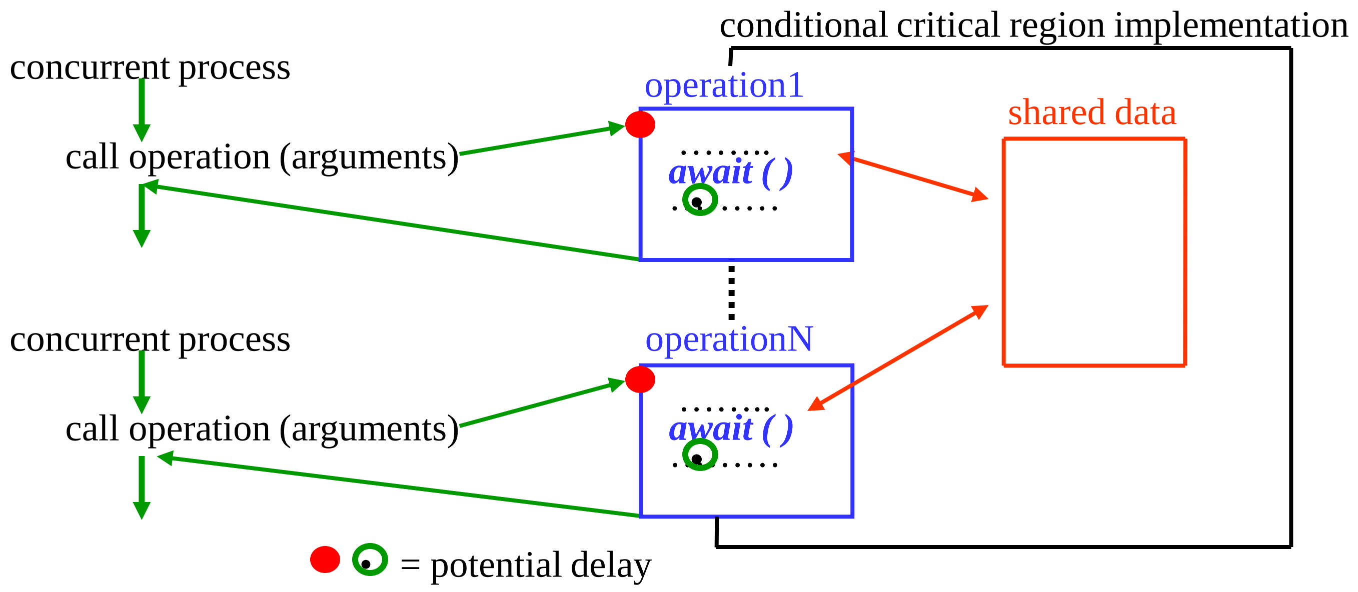
Note that the programmer must leave the data structure in a consistent state
before executing ***await***, as well as before exiting the region.

CCRs are difficult to implement. Programmers may invent any condition on the shared data.
All conditions have to be tested when any process leaves the region.

We now introduce an illustration of CCRs and the subsequent evolution of concurrency control

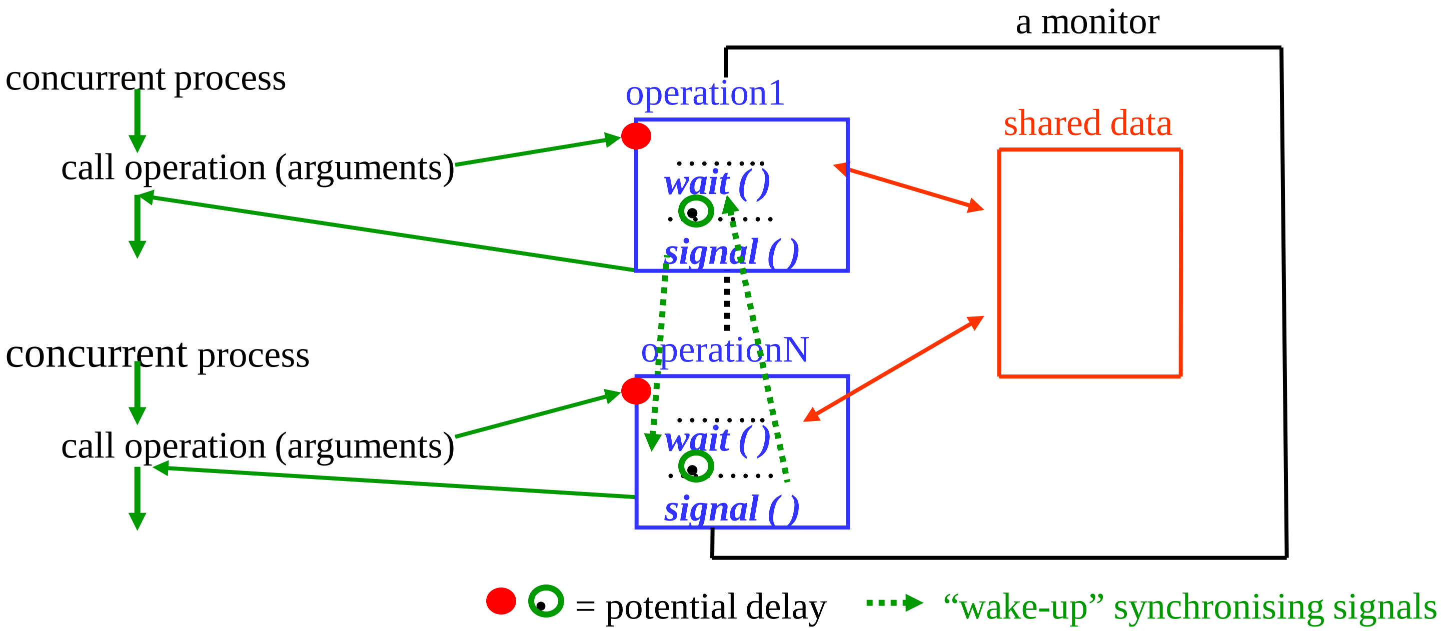
Concurrent programming paradigms and models - 1

1. shared data is a passive object accessed via concurrency-controlled operations



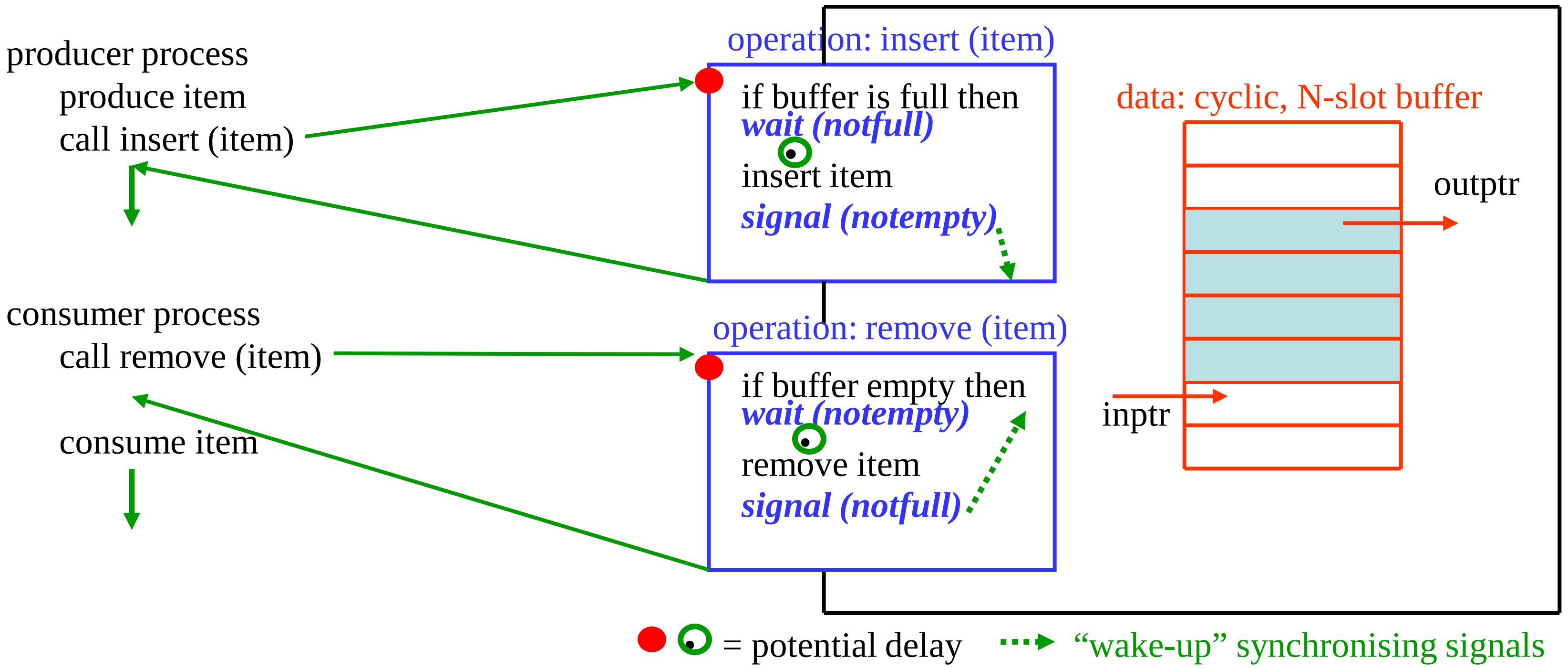
- we use a programming-language-independent, diagrammatic representation
- shared data is encapsulated with operations in a passive object, called by concurrent processes
- operations that read and/or write execute under mutual exclusion (semaphore implementation) ●
- in some languages (Mesa, Java), other operations may execute without exclusion
- conditional critical regions (CCRs) are illustrated above.
- note that processes do not have to signal explicitly (unlike semaphores)

Illustration of monitors



- operations that read and/or write execute under mutual exclusion (semaphore implementation)
- in monitors, condition synchronisation is provided, by ***wait*** and ***signal*** operations on condition variables, named by programmers e.g. ***not-full, free-to-read***
- processes must test the data and decide whether they need to block until a condition becomes true
- a process that ***wait***s on a condition variable always blocks, first releasing the monitor lock (the implementation manages this)
- ***signal*** has no effect if there are no processes blocked on the condition variable being signalled
- after ***signal*** the monitor lock must be re-acquired for an unblocked process after the signalling process has left the region (the implementation manages this)

Passive object example: monitors and condition variables



monitor operations are executed under exclusion

condition variables (***notfull, notempty***) are defined for synchronisation,
operations on them are ***wait*** and ***signal***

data is tested in the monitor before a ***wait*** operation, semantics of ***wait***: process is always queued

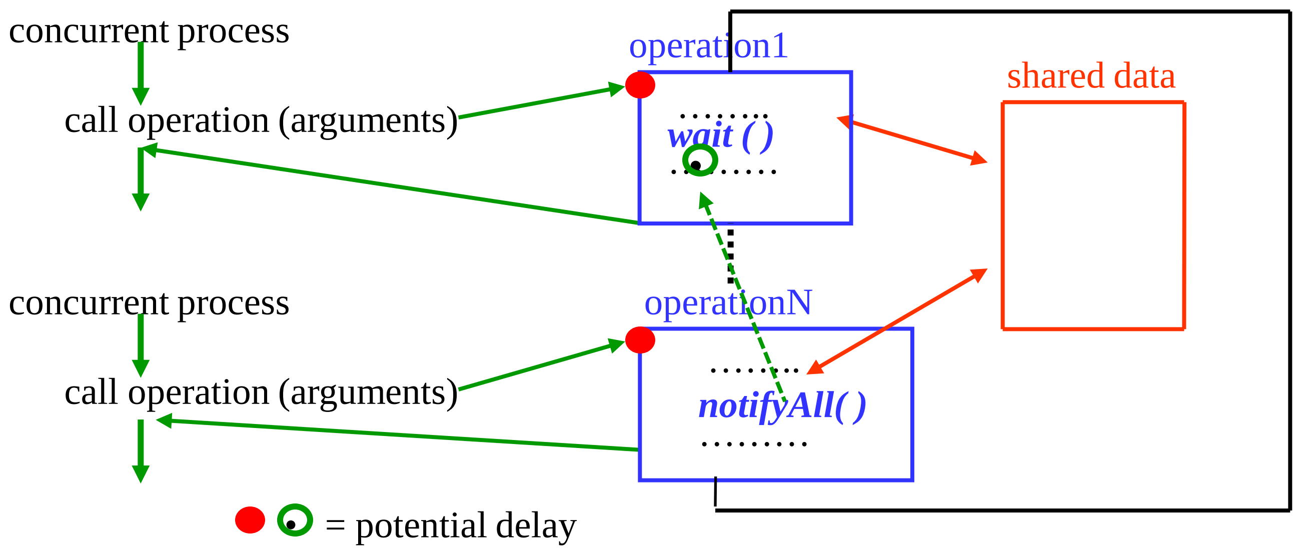
semantics of ***signal***: if there is no blocked process – no effect
if there is a queue, wake up ONE process

note: only one process can ever be active inside a monitor (mutual exclusion property)

after ***signal***, should it be the signaller or the signalled process? (implementation decision)

Java synchronised methods

Synchronised methods of an object execute under mutual exclusion with respect to all synchronised methods of an object.



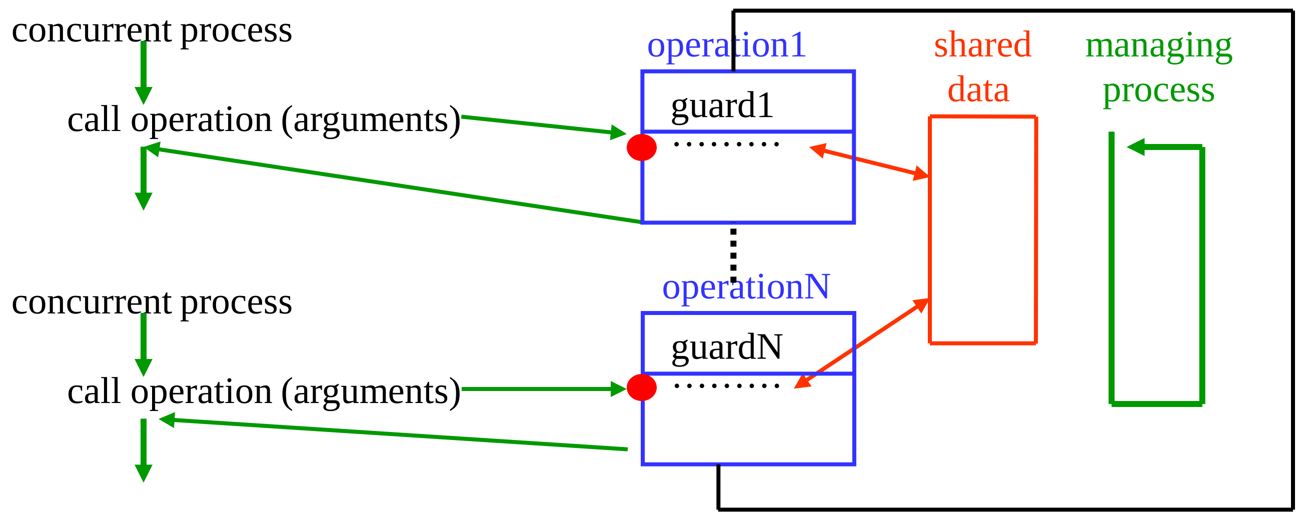
- condition synchronisation is similar to the pthreads package
- ***wait*** blocks the process/thread and releases the exclusion on the object
- ***notify***: the implementation frees an arbitrary process – take care!
- ***notifyAll***: the implementation frees all blocked processes. The first to be scheduled may resume its execution (under exclusion) but must retest the wait condition. The implementation must manage reclaiming the exclusion, i.e. the PC of the resuming processes to achieve retest. Note that processes could resume and block repeatedly, e.g. on a multiprocessor.

Java example, buffer for a single integer, Bacon and Harris section 12.2.4, p369

```
public class Buffer {
     private int value = 0;
     private boolean full = false;

     public synchronized void put (int a)
                                                    throws InterruptedException {
     while (full)
            wait ( );
     value = a:
     full = true;
      notifyAll( );
    }
public synchronized int get ( )
                                                    throws InterruptedException {
    int result
    while (!full)
           wait( );
     result = value;
     full = false;
     notifyAll( );
     return result;
     }
}
```

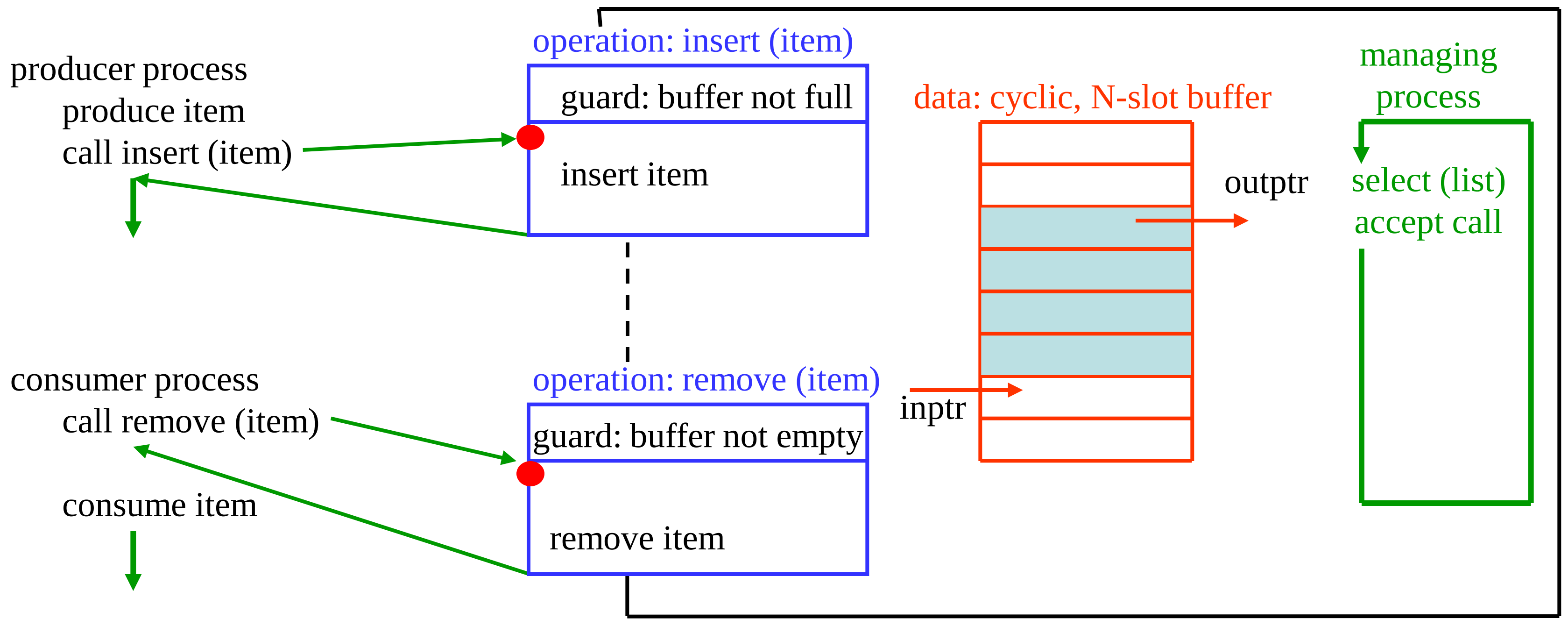
Concurrent programming paradigms and models - 2

1. shared data is an active object managed by a process



- shared data is encapsulated with operations in an active object, called by concurrent processes
- the managing process performs condition testing, and ..
- .. only accepts calls to operations with guards that evaluate to true
- mutual exclusion and condition synchronisation are ensured by the managing process
- note that synchronisation is at the granularity of whole operations (note that *path expressions* also have this feature)
- which process (caller or manager)? executes the accepted operation is implementation-dependent

Active object example: Ada select/accept



- managing process selects from operations whose guard evaluates to true
- and accepts a call from the select list
- a "rendezvous" occurs between the managing process and the calling process
- one of them (not defined, implementation-specific) carries out the call and return
- note that the operation programming is simplified because the active managing process carries out both mutual exclusion and condition synchronisation

Recall problems with semaphores (18) – solved?

Difficult for programmers to use correctly – programs are complex
waiting and signalling have been made easier for the programmer than with semaphore programming.

Unconditional commitment to block
- as before - can sometimes use ***fork*** for parallel operation
- pthreads offers ***test lock*** as well as ***wait*** - but there can still be race conditions between them

Unbounded delay on wait
- pthreads offers time-limited waits – for mutual exclusion, not for condition synchronisation

Priority inversion (these points also apply to semaphore implementations)
- queues of blocked processes need not be FCFS
- suppose process/thread priority can be known to the implementation of semaphores etc.
- implementations can re-order the queues of blocked processes according to priority
- raise the priority of the lock-holder to the highest priority waiting process

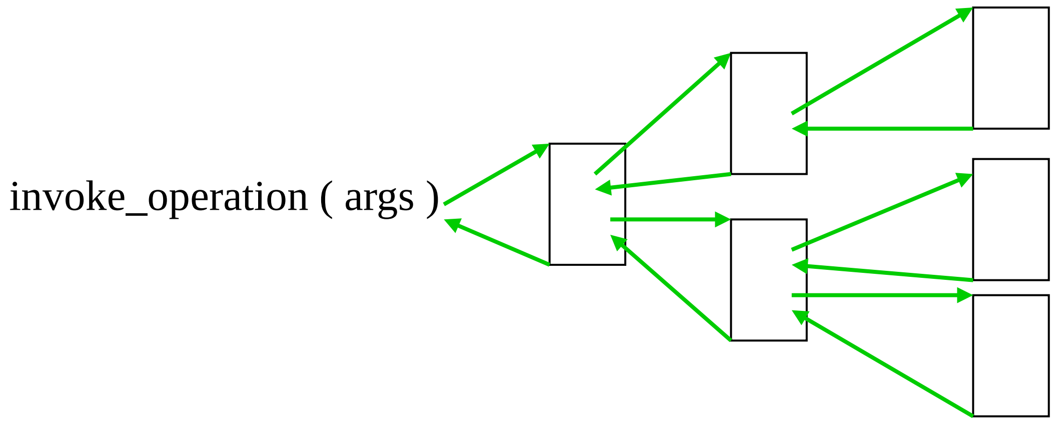
Convoy effect - a long lock-hold can hold up a lot of potentially short ones.
- try to program with fine-grained locking (components rather than whole structures)

Library calls - a universal problem! Static analysis of code executed under mutual exclusion
becomes impossible when these operations make extensive use of library calls.
(motivation for Java+Kilim – see later)

Composite operations in main memory - 1

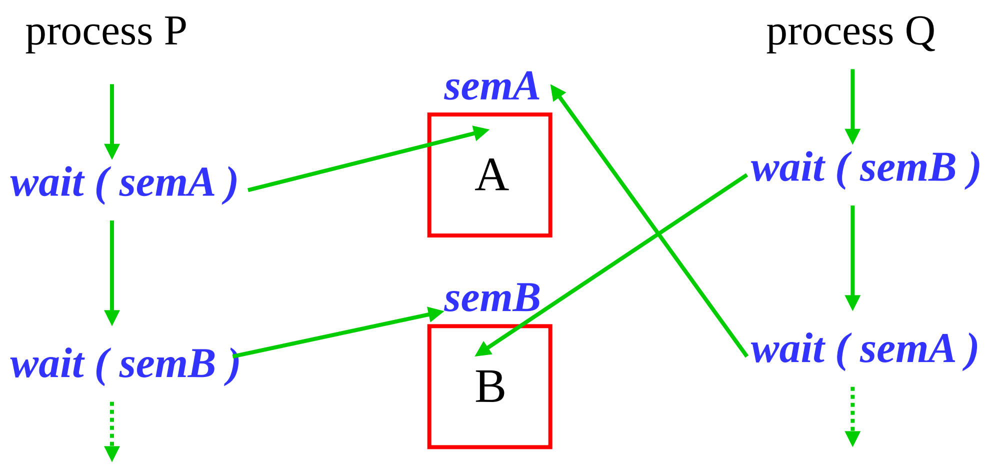
We have studied how to make one operation on shared data atomic in the presence of concurrency and crashes.
Now suppose a meaningful operation comprises several such operations:

- e.g. transfer: subtract a value from one data item and add the same value to another.
- e.g. test some integer counts to decide whether you can write some shared data;
  proceed to write if there are no existing readers or writers



Composite operations in main memory - 2 - example

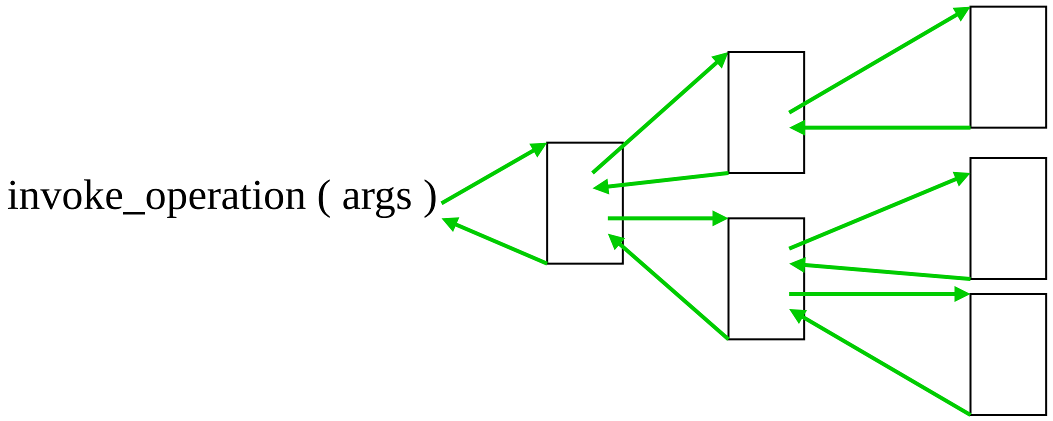
The sequence below may work correctly over a long period then unfortunate timing may cause deadlock. We illustrate this using semaphores – easily generalisable to higher level constructs (that are probably implemented using semaphores).



At this point we have deadlock. Process P holds ***semA*** and is blocked, queued on ***semB***
Process Q holds ***semB*** and is blocked, queued on ***semA***
Neither process can proceed to use the resources and signal the respective semaphores.
A cycle of processes exists, where each holds one resource and is blocked waiting for another, held by another process in the cycle.

Deadlock: systems that allocate resources dynamically are subject to deadlock.
We later study the policies and conditions necessary and sufficient for deadlock to exist.

Composite operations in main memory – 3 – solutions?



Concurrency control: why not lock all data – do all operations – unlock?
But contention may be rare, and “locking all” may impose overhead and slow response (e.g. Python)

Crashes?: in main memory everything is lost on a crash – no problem! unless any externally visible effects have occurred. These could be output, or changes to persistent state.
We’ll consider persistent store later. Assume that output generated by concurrent composite operations should be deferred until the operation completes successfully.

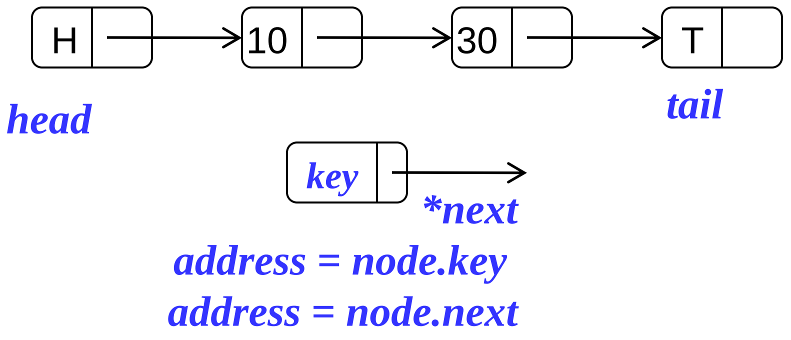
Atomicity: we first solved how to make a single operation on an object atomic/indivisible.
DEFINITION: a composite operation is atomic if either all of its component operations complete successfully, or no operation has any effect, i.e. all data values are unaffected by the composite operation and have the values they had before it started and failed.
Note that it must be ensured that no concurrent process can see any intermediate state of the data.

Lock-free programming1

Concurrent programs are difficult to develop correctly, particularly for large-scale systems.
Problems such as priority inversion, deadlock and convoying have been highlighted.

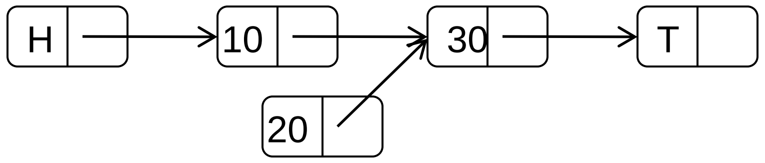
Lock-free programming became established as a research area from the late 1990s
We'll introduce it this year and develop it in next year's courses.
We'll use a non-blocking linked list as an example,
from Tim Harris's paper
A Pragmatic Implementation of Non-Blocking Linked Lists
DISC 2001, pp. 300-314, LNCS 2180, Springer 2001

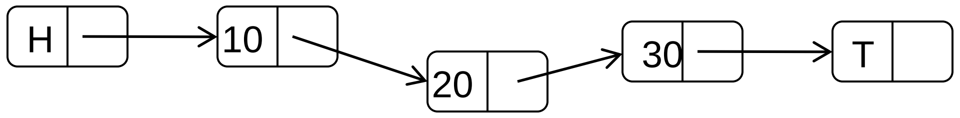
Consider an ordered list of integers as follows:



Lock-free programming 2

Insertion is straightforward. First, the list is traversed until the correct position is found.
Then a new cell is created and inserted using ***CAS*** (compare and swap)





CAS (address, old, new) atomically compares the contents of ***address*** with the ***old*** value and, if they match, writes the ***new*** value to that location.

Note that if the ***CAS*** fails, this means that the list has been updated concurrently by other thread(s) and the traversal must start again to find the correct place to insert.

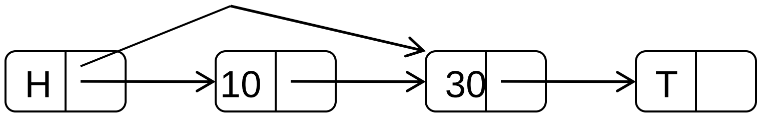
Contrast this with the “spin lock” approach for claiming a semaphore (mutex lock).

In this case the “***read-and-clear***” repeats until it succeeds (ref slide 5).

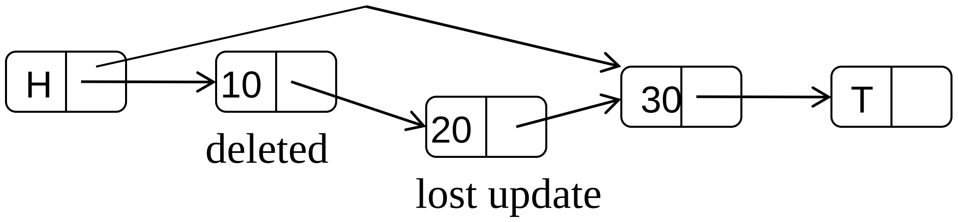
How would you program an ordered list using semaphores? Lock the whole list?

Lock-free programming 3

Correct deletion is more difficult, consider:

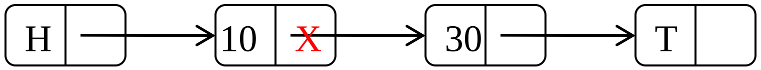


CAS (address, old, new) could be used to change ***node.next*** in the head to point to 30,
after checking the old value points to 10 (so there were no concurrent inserts between H and 10)
But concurrent threads could have inserted values between 10 and 30, after 30 was selected
for the new pointer from H.
Those inserts would be lost:

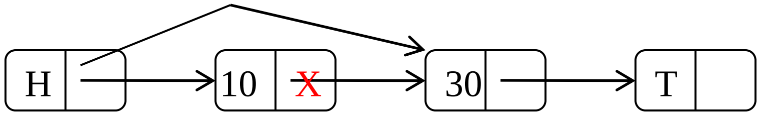


Lock-free programming 4

Correct deletion:



atomically **mark** node for deletion (X)
The node is "*logically deleted*" and this can be detected by concurrent threads
that must cooperate to avoid concurrent insertions/deletions at this point
A marked node can still be traversed.



The node is "*physically deleted*"

The algorithms are given in C^{++} - like pseudo-code in the paper, as is a proof of correctness

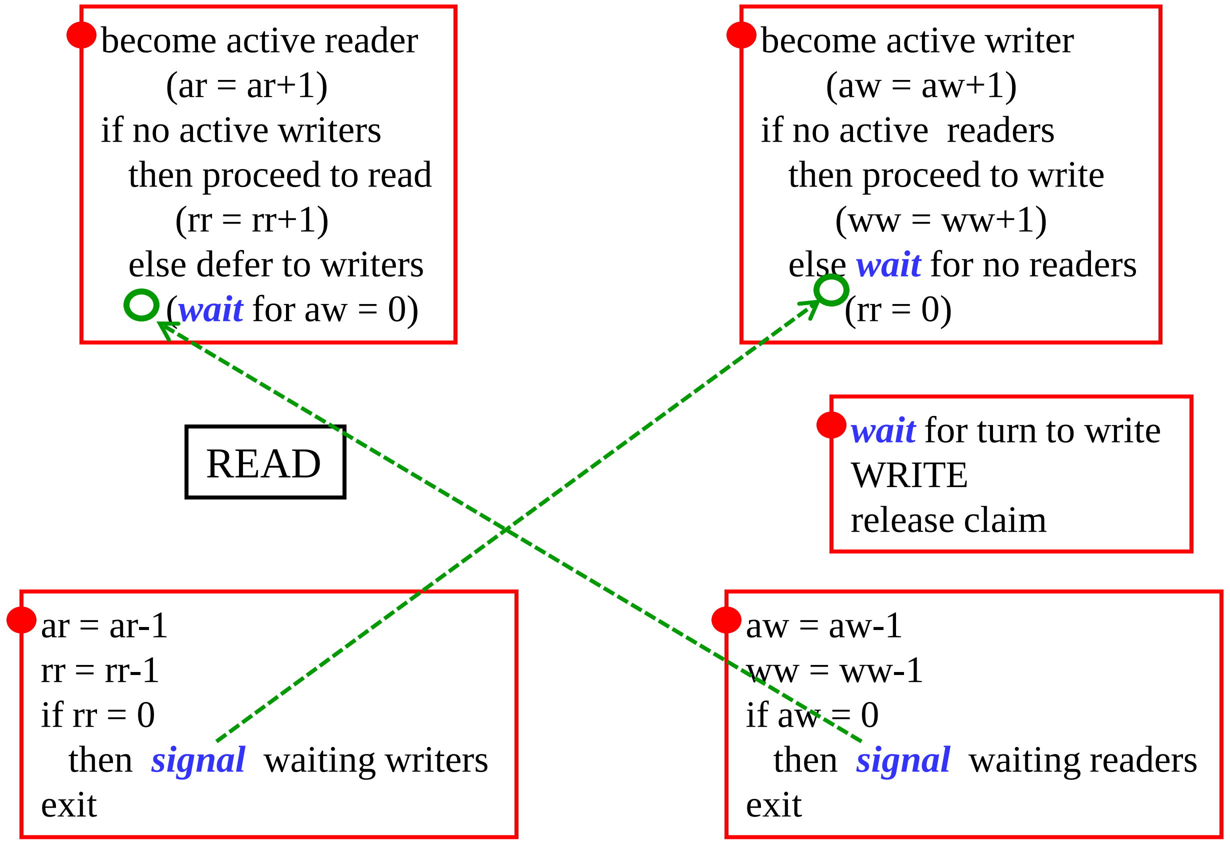
Lock-free programming 5

Selected further work:

Keir Fraser,
Practical Lock Freedom, 2004.
PhD thesis (UK-DD winner), UCAM-CL-TR-579

Keir Fraser and Tim Harris
Concurrent programming without locks
ACM Transactions on Computer Systems (TOCS) 25 (2), 146-196, May 2007

Multiple readers, single writer concurrency control -2



```
● become active reader
      (ar = ar+1)
  if no active writers
     then proceed to read
        (rr = rr+1)
     else defer to writers
     ○ (wait for aw = 0)
```

```
● become active writer
      (aw = aw+1)
  if no active  readers
     then proceed to write
        (ww = ww+1)
     else wait for no readers
       ○ (rr = 0)
```

```
READ
```

```
● wait for turn to write
  WRITE
  release claim
```

```
● ar = ar-1
  rr = rr-1
  if rr = 0
     then  signal  waiting writers
  exit
```

```
● aw = aw-1
  ww = ww-1
  if aw = 0
     then  signal  waiting readers
  exit
```

● mutual exclusion - to access shared counts
 - for write access

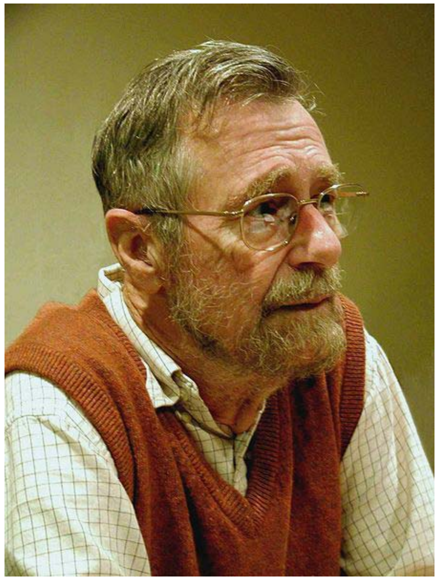
- - - - -> ○ condition synchronisation

```
● become active reader
        (ar = ar+1)
  if no active writers
    then proceed to read
        (rr = rr+1)
    else defer to writers
     ○ (wait for aw = 0)
```

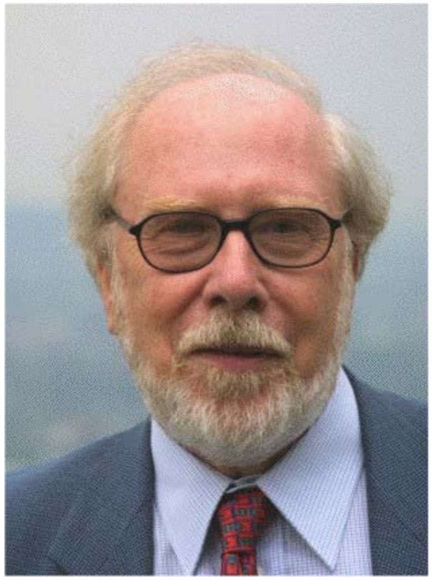
```
● wait (CountGuard-sem)






  signal (CountGuard-sem)
```

```
● wait (CountGuard-sem)
  ar = ar+1
  if  aw=0 then rr = rr+1
      else wait (Rsem) ○      deadlock! blocking while holding a semaphore
  signal (CountGuard-sem)
```

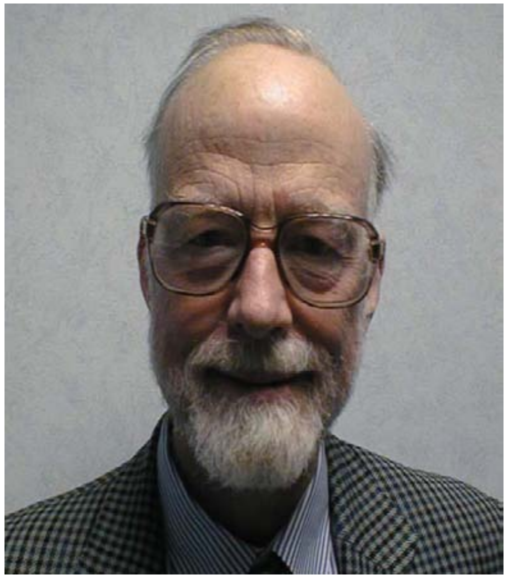
So the programmer has to program to avoid deadlock



Edsger Dijkstra



Niklaus Wirth



Tony Hoare