Classical, shared memory concurrency control

- Concurrency control constructs in programming languages

  Can concurrent programming languages make concurrent programming easier than semaphore programming?

  We look at a number of different approaches in programming languages, as a logical and historical development.

Concurrency control – support for the programmer, 1

Developments in programming languages to support 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 explicitly)
synchronized methods and wait/notify (Java)

active objects: guarded commands, Ada select/accept and rendezvous

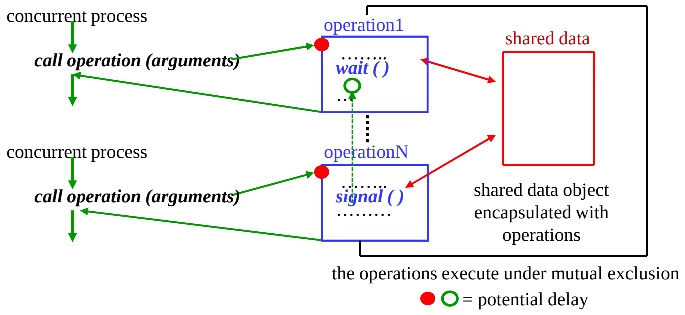
Edsger Dijkstra

Niklaus Wirth

Tony Hoare

Concurrent programming paradigms and models: Passive objects

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


concurrent process
call operation (arguments)
operation1
……..
wait ( )
shared data
concurrent process
call operation (arguments)
operationN
……..
signal ( )
………
shared data object encapsulated with operations
the operations execute under mutual exclusion
● ○ = potential delay

- 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 (implemented by a semaphore) and are indicated by ●
- Condition synchronisation is provided in different ways and will be indicated by ○

Critical regions ....

Critical regions were proposed as a means of hiding the complexity of semaphore programming.

var v: shared <data-structure> \\ compiler assigns a semaphore to protect ***v***, ***Semv***, initially 1

\\ compiler inserts semaphore operations

region v do ● ***begin ……*** \\ ***wait(Semv)*** at begin

end \\ and ***signal(Semv)*** at end

But this is only **mutual exclusion**.

Conditional critical regions (CCRs) add condition synchronisation

…….. and conditional critical regions

Condition synchronisation was added to CCRs
by including (note that this is *NOT implemented by a semaphore*):

await < some condition on shared data >

If the condition is true, the process continues.
If the condition is false the implementation ensures that

- the region is unlocked
- and the process executing ***await*** is blocked until the condition becomes true.

When it is selected to continue in the region the implementation again acquires the lock for it.

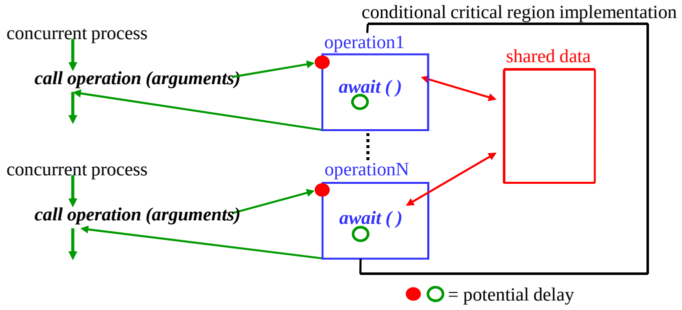
Note that the programmer must leave the data structure in a consistent state before executing ***await***, as well as before exiting the region.

CCRs are difficult to implement and impose too much overhead.
Programmers may invent any condition on the shared data e.g. items < N, spaces > 0 …….
All conditions (arbitrary expressions) have to be tested when any process leaves the region.

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

Illustration of CCRs

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



- shared data is encapsulated with operations in a passive object, called by concurrent processes
- operations execute under mutual exclusion
- implemented by a mutex semaphore ● (associated with the shared data)

- conditional critical regions (CCRs) are illustrated above.
- note that ***processes do not have to signal explicitly*** (unlike semaphore implementations of condition synchronisation)

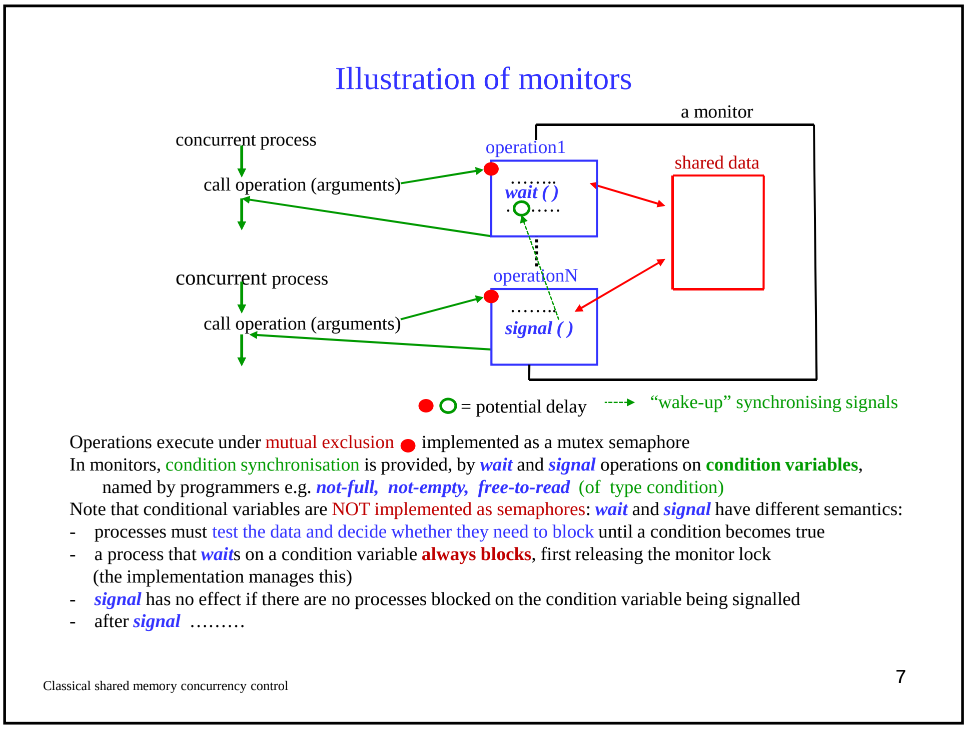


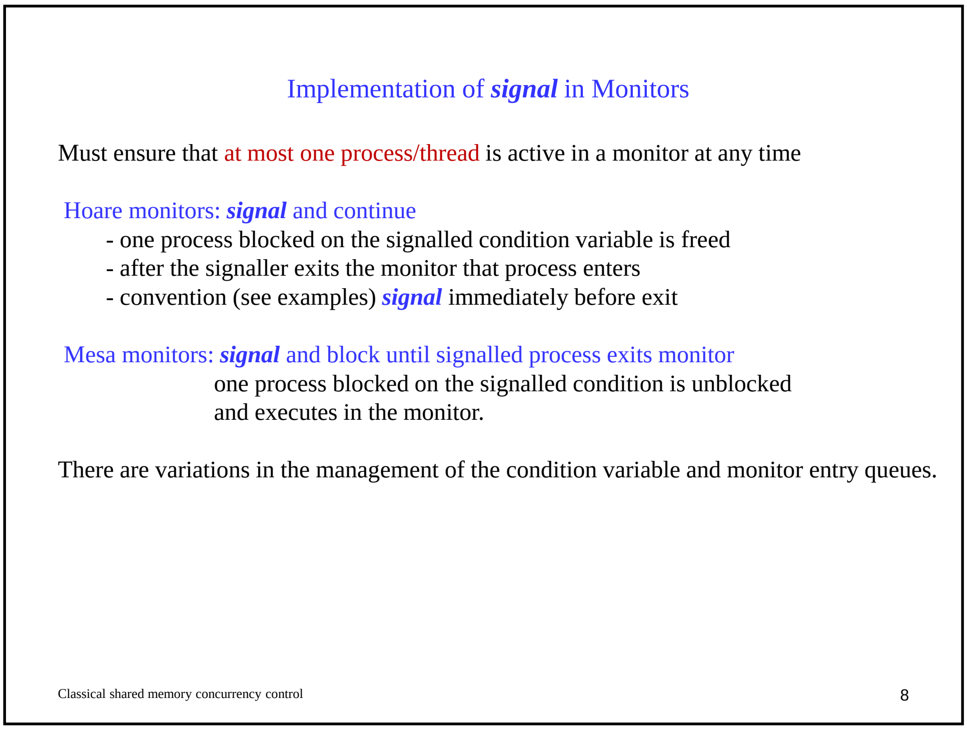
Illustration of monitors

Operations execute under mutual exclusion ● implemented as a mutex semaphore

In monitors, condition synchronisation is provided, by ***wait*** and ***signal*** operations on **condition variables**, named by programmers e.g. ***not-full, not-empty, free-to-read*** (of type condition)

Note that conditional variables are NOT implemented as semaphores: ***wait*** and ***signal*** have different semantics:

- 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*** ………



Implementation of ***signal*** in Monitors

Must ensure that at most one process/thread is active in a monitor at any time

Hoare monitors: ***signal*** and continue

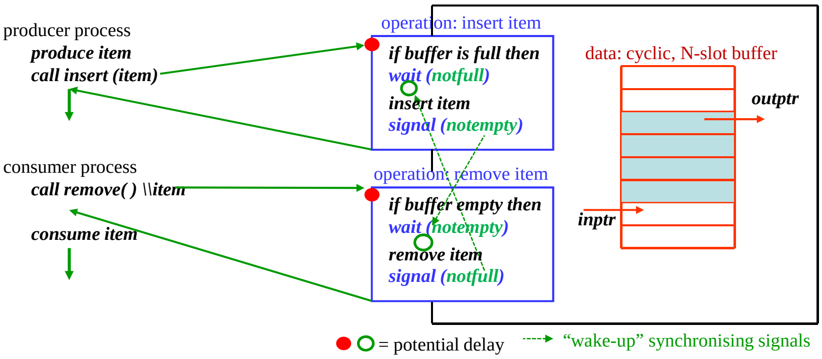
- one process blocked on the signalled condition variable is freed
- after the signaller exits the monitor that process enters
- convention (see examples) ***signal*** immediately before exit

Mesa monitors: ***signal*** and block until signalled process exits monitor

one process blocked on the signalled condition is unblocked
and executes in the monitor.

There are variations in the management of the condition variable and monitor entry queues.

Monitor example: Producers and consumers (bounded N-slot buffer)



producer process
produce item
call insert (item)

consumer process
call remove( ) \\item
consume item

operation: insert item
if buffer is full then
wait (notfull)
insert item
signal (notempty)

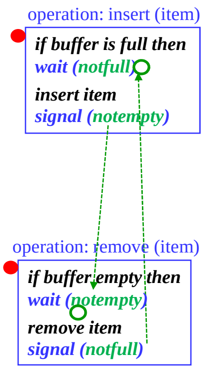
operation: remove item
if buffer empty then
wait (notempty)
remove item
signal (notfull)

data: cyclic, N-slot buffer
outptr
inptr

● ○ = potential delay
"wake-up" synchronising signals

monitor operations are executed under exclusion – can't exploit single producer/consumer concurrency
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

Monitor for producers/consumers - 2



operation: insert (item)
if buffer is full then
wait (notfull)
insert item
signal (notempty)

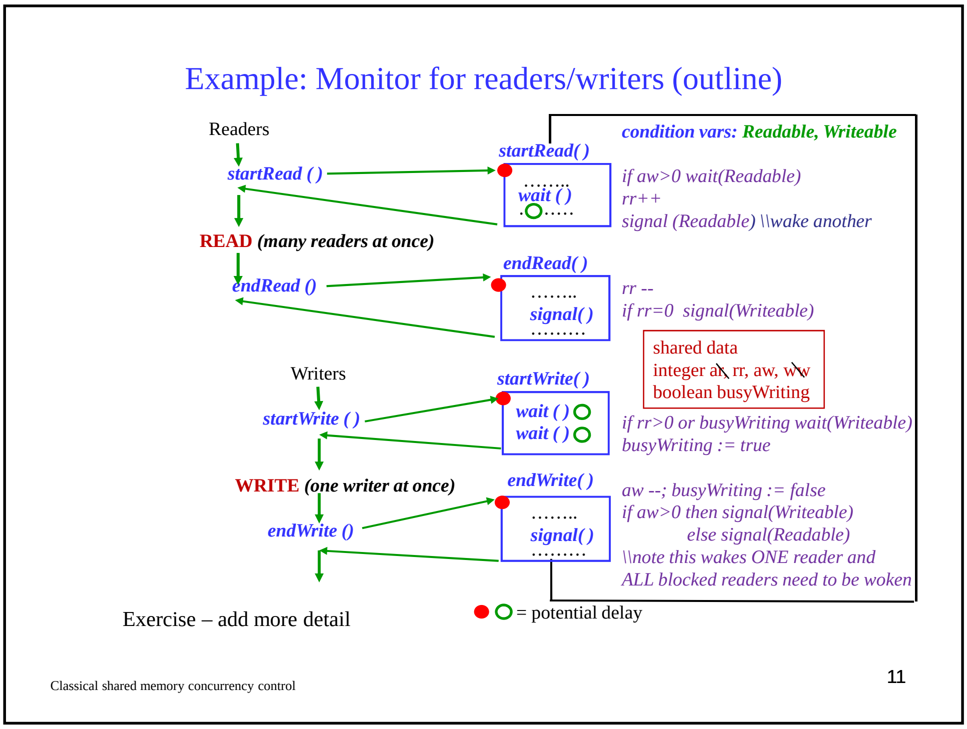
operation: remove (item)
if buffer empty then
wait (notempty)
remove item
signal (notfull)

Filling in more detail:

```
count = 0   \\ initialise count to zero items in buffer (maximum  = N)

insert (item)
    if count = N then wait (notfull)  \\ if count < N process continues without delay
    insert item
    increment inptr to point to next empty slot in buffer
    count = count + 1
    signal (notempty)

remove (item)
    if count = 0 then wait (notempty)  \\ if count > 0 process continues without delay
    remove item
    increment outptr to point to next item in buffer
    count = count – 1
    signal (notfull)
    return (item)
```



Example: Monitor for readers/writers (outline)

Readers

startRead ( )

READ ***(many readers at once)***

endRead ()

Writers

startWrite ( )

WRITE ***(one writer at once)***

endWrite ()

startRead( )

……..
wait ( )
.○…..

endRead( )

……..
signal( )
………

startWrite( )

wait ( ) ○
wait ( ) ○

endWrite( )

……..
signal( )
………

condition vars: ***Readable, Writeable***

if aw>0 wait(Readable)
rr++
signal (Readable) \\wake another

rr --
if rr=0 signal(Writeable)

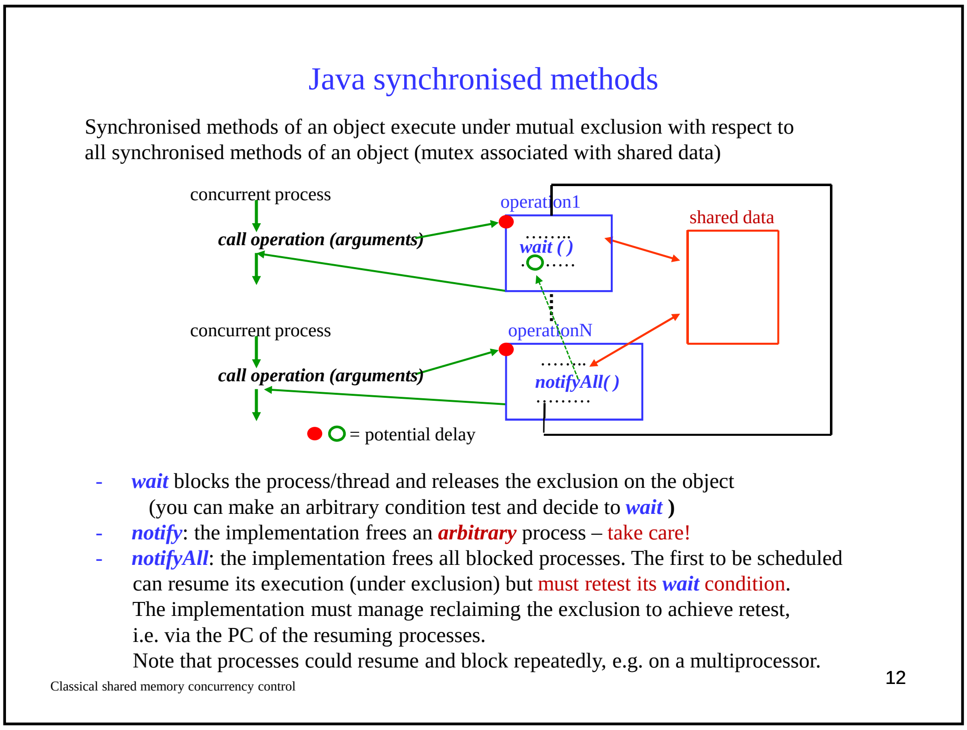
shared data
integer ar, rr, aw, ww
boolean busyWriting

if rr>0 or busyWriting wait(Writeable)
busyWriting := true

aw --; busyWriting := false
if aw>0 then signal(Writeable)
else signal(Readable)
\\note this wakes ONE reader and
ALL blocked readers need to be woken

Exercise – add more detail

● ○ = potential delay



Java synchronised methods

Synchronised methods of an object execute under mutual exclusion with respect to all synchronised methods of an object (mutex associated with shared data)

concurrent process

call operation (arguments)

concurrent process

call operation (arguments)

operation1

……..
wait ( )
.○…..

operationN

……..
notifyAll( )
………

shared data

● ○ = potential delay

- ***wait*** blocks the process/thread and releases the exclusion on the object (you can make an arbitrary condition test and decide to ***wait*** **)**
- ***notify***: the implementation frees an ***arbitrary*** process – take care!
- ***notifyAll***: the implementation frees all blocked processes. The first to be scheduled can resume its execution (under exclusion) but must retest its ***wait*** condition.
  The implementation must manage reclaiming the exclusion to achieve retest, i.e. via the PC of the resuming processes.
  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;
    }
}
```

Recall problems with semaphores – solved in CP languages?

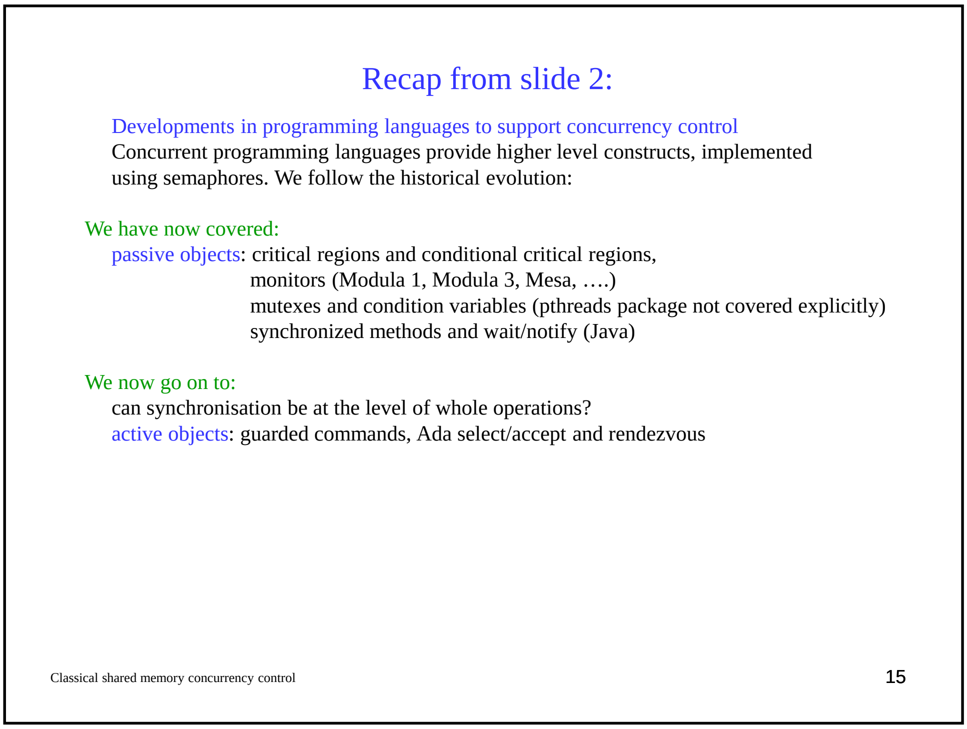
Difficult for programmers to use correctly – programs are complex
mutual exclusion and condition synchronisation have been made simpler by allowing programmer to synchronise without explicitly releasing a mutex – still hard to implement

Unconditional commitment to block
- as before - can sometimes use ***fork*** for parallel operation
- e.g. 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)



Recap from slide 2:

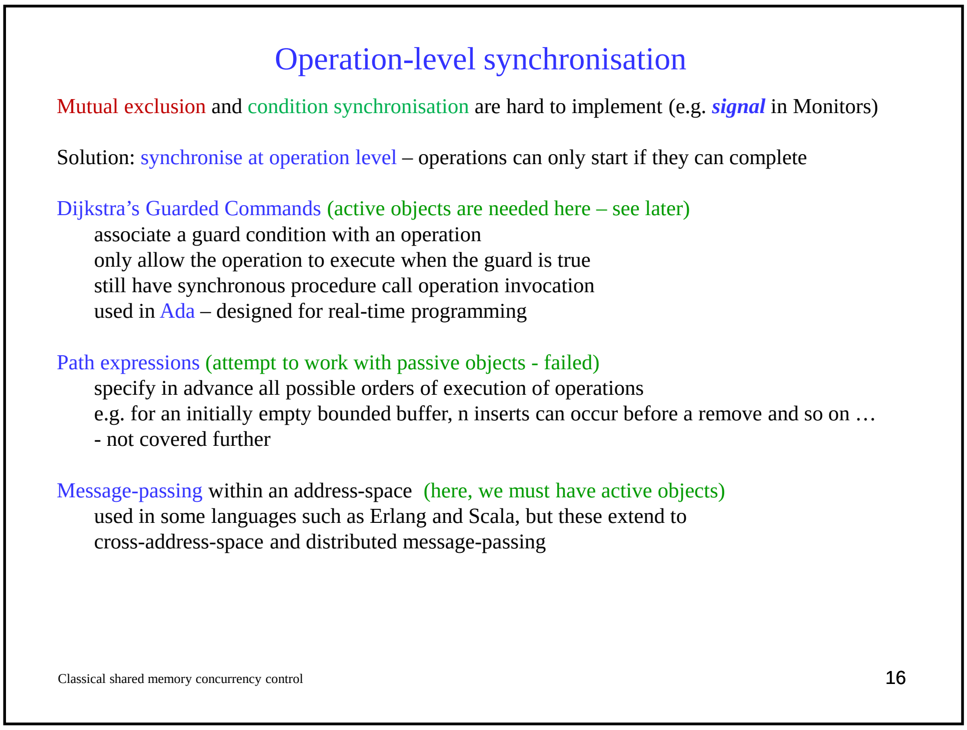
Developments in programming languages to support concurrency control
Concurrent programming languages provide higher level constructs, implemented using semaphores. We follow the historical evolution:

We have now covered:

passive objects: critical regions and conditional critical regions,
monitors (Modula 1, Modula 3, Mesa, ….)
mutexes and condition variables (pthreads package not covered explicitly)
synchronized methods and wait/notify (Java)

We now go on to:

can synchronisation be at the level of whole operations?
active objects: guarded commands, Ada select/accept and rendezvous



Operation-level synchronisation

Mutual exclusion and condition synchronisation are hard to implement (e.g. ***signal*** in Monitors)

Solution: synchronise at operation level – operations can only start if they can complete

Dijkstra’s Guarded Commands (active objects are needed here – see later)

associate a guard condition with an operation
only allow the operation to execute when the guard is true
still have synchronous procedure call operation invocation
used in Ada – designed for real-time programming

Path expressions (attempt to work with passive objects - failed)

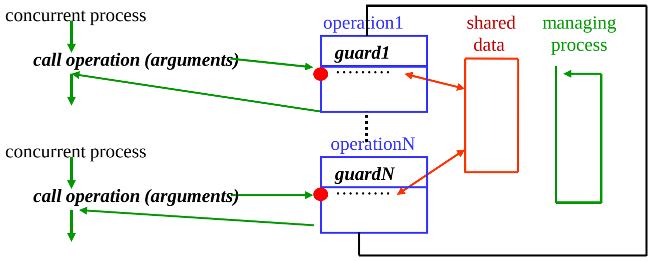
specify in advance all possible orders of execution of operations
e.g. for an initially empty bounded buffer, n inserts can occur before a remove and so on …
- not covered further

Message-passing within an address-space (here, we must have active objects)

used in some languages such as Erlang and Scala, but these extend to
cross-address-space and distributed message-passing

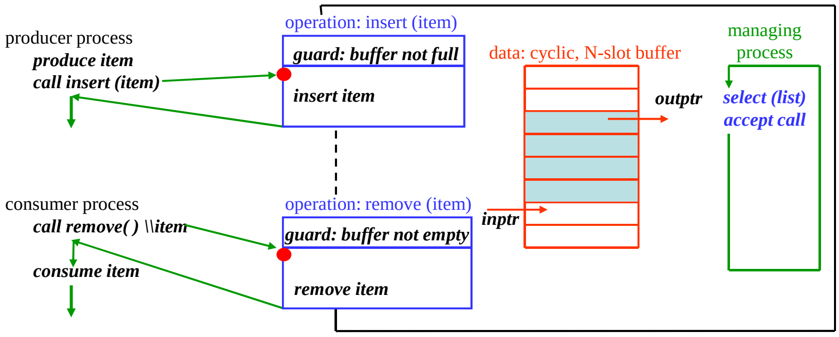
Active objects

Now, shared data is an active object managed by a process



- shared data is encapsulated with operations in an active object, managed by a process/thread
- 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
- which process (caller or manager)? executes the accepted operation is implementation-dependent

Example: Producers/Consumers with Ada *select/accept*



- managing process ***select***s from operations whose guard evaluates to true (the select list)
- and ***accept***s 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
- Greater concurrency than e.g. monitors – active process can manage counts

Active objects: Discussion

Motivation for active objects came from operation level synchronisation,
but we are on the way to a Client-Server Model:
Avoid shared data altogether – define a server to manage the data
Send the server a request to perform an operation on the data.
- what form does the request take?

Above - active objects in the same address space as the calling processes (Ada)
- callers execute synchronous procedure calls (heap/stack available)
- can't execute in parallel (unless they fork a child before the call)
- how does context switching overhead compare with other approaches?

The idea generalises to cross-address-space (next lecture) and distributed programming (next term)

Historical note

By the 1980s we had networks – fast LANs and slower WANs
Researchers and companies were working on:

Remote Procedure Call to distribute programs on LANs
e.g. Bruce Nelson PhD thesis 1982 CMU and XEROX PARC (available online)
e.g. SunRPC, MayFlower RPC (Cambridge), ANSA RPC ………
Paradigm: synchronous method invocation
client-server, in general

Message-passing to abstract above communication packets in WANs
- to connect existing clients, databases etc.
e.g. various research systems and languages (RIG, …..)
e.g. IBM MQseries evolved into
WebSphere with JMS interface
- far bigger market than RPC
Paradigm: asynchronous communication (send and continue)
don't expect an instant response – get on with work and pick up reply later

Both are needed in general systems design – see Distributed Systems course.

Summary

Concurrency control in programming languages.

- What can the implementation provide for the programmer?
- Do they make it easier to write correct programs?
- Passive objects:
  - conditional critical regions - programmers just ***await*** - no signalling needed
    - too much for implementation, too high overhead
  - monitors - programmers have to ***signal*** as well as ***wait*** on condition synchronisation
    - but can do it inside a critical region (unlike semaphore programming).
    - semantics of ***signal*** vary
  - Java - compose monitor-like structures.
    - no general condition specification, just ***wait, notify, notify-all***
    - (please check recent Java releases)
- Active objects – test conditions (guards) BEFORE executing the CRs
  - is this client-server style a good general solution?
  - how does context-switching overhead compare with other methods?

Next

cross-address-space inter-process communication