Chapter 2

Parallel Techniques

All distributed systems present similar problems for the system analyst.

Parallel execution …

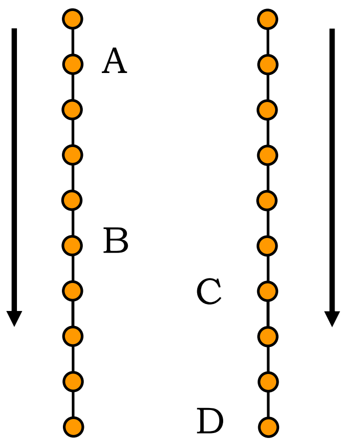
Data movement (eg Cloud)

Synchronisation problems between different resources.

In order to make use of eg Cloud resources it is necessary to understand how to break up a problem into a number of processes which can execute independently.

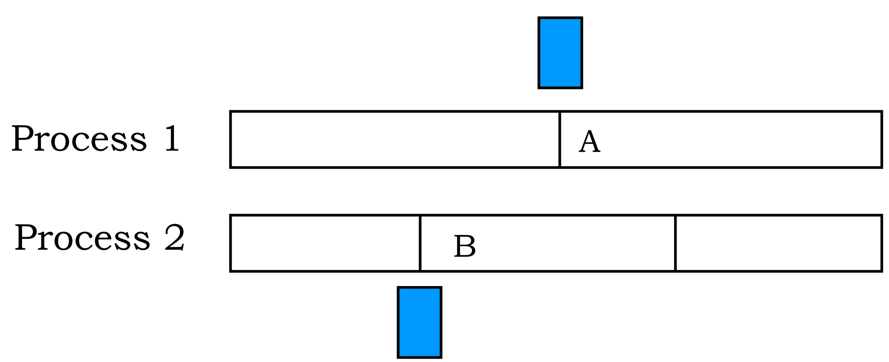
Spend some time discussing this ….

Techniques



Parallel / Distributed Programming has a serious difficulty.

Synchronisation
Parallel processes on one machine lead to the idea Of *non-determinism*
In the absence of any explicit synchronisation there is no order in which instructions in different processes are executed and in particular the order may change between invocations of the task.
(Partial order) There is a partial ordering in that ordering is predictable in a single process. And order between processes is weakly ordered.
A will always occur before B and C before D.
Also if for a particular run B occurs before C then D will occur after B
What if process 2 should not proceed beyond point B until 1 reaches point A



Multi threads are a special instance

Interprocess synchronisation

Flags

We might imagine a flag – integer variable accessible by two or more processes.

For grid computing we need to worry about network connections. Failure to write

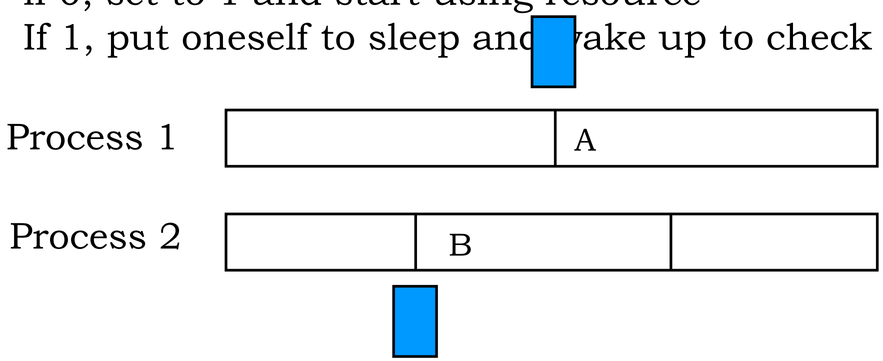
Synchronisation. 2 has to wait for 1.

Repeated checking is wasteful. Go to sleep

When 1 arrives it sets the flag to one and continues
When 2 arrives it checks the flag.
    If 1 continues
    If 0 – puts itself to sleep for some period, before waking up and checking again.

Also good for exclusive access to a resource

Exclusive access
Initialise to 0.
When you want exclusive access set to 1
When you have finished set to 0.
On arrival check flag
    if 0, set to 1 and start using resource
    If 1, put oneself to sleep and wake up to check again



Interprocess synchronisation

Problem with flags

There are serious problems with flags in the checking and incrementing operation.

Firstly more than 1 process may be waiting for the resource and which gets it is totally random

The polling is a consumer of resources

It does not guarantee exclusive use of the resource

Operation not atomic

```
Do while (flag==1) {
  wait(200)
}
flag++;
Use resource
```

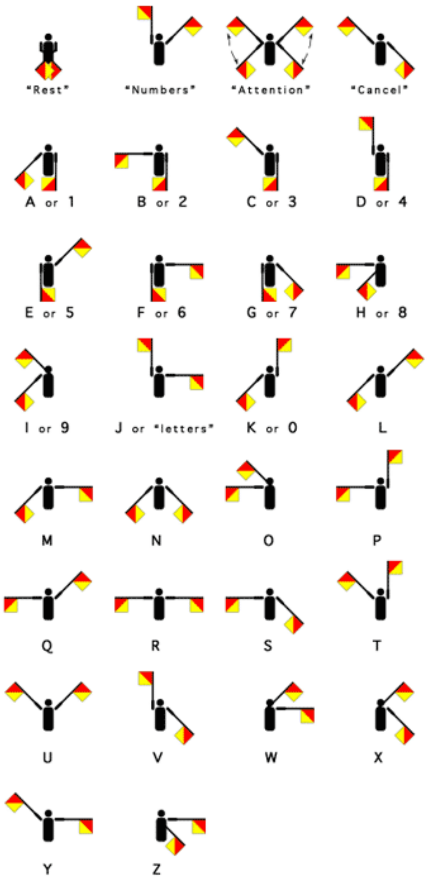
```
ld flag, r0
inc r0
st r0, flag
```

Single processor we have time slicing issues – losing the processor in the middle of the actions of checking and setting.

In grid computing the network latencies can be just as damaging.

The solution is a semaphore

Semaphore



Dijkstra (1965) developed semaphores

Common variable – but set and reset by a single atomic un-interruptible action

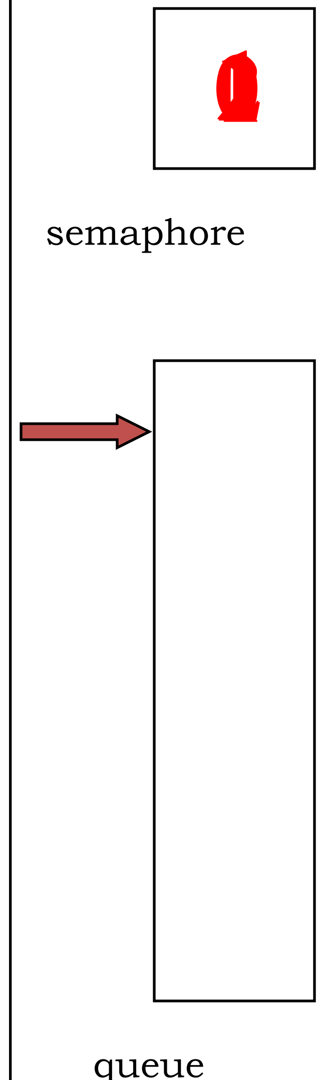
Semaphore is a non-negative integer
Operations are signal and wait

signal increments the semaphore
wait decrements the semaphore UNLESS the result of the operation would be to make the semaphore negative.
In this case the process is moved to a wait queue.

Look first at operation and then at implementation

Semaphores & Resources (i)*

1. A limited resource is available. Let us say two processes can use it.
2. OS then initialises the semaphore to 2.
3. Process A wishes to access the resource. It waits on the resource semaphore. Sets it to 1 an runs.
4. Process A finishes and signals the semaphore.
5. Process B waits on the resource, and runs.
6. Before B returns process C waits on the resource. The semaphore is now 0.
7. If B or C return before D waits on the semaphore then all runs smoothly. But suppose D arrives when B and C are still using the resource



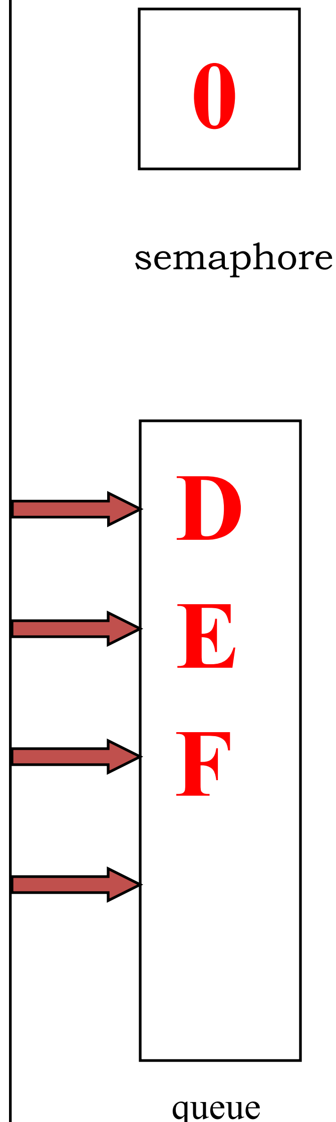
Semaphores & Resources (ii)*

1. Suppose B and C are running. Process D comes along. It does a wait on the semaphore. Unable to run it is put in the queue for this resource.

2. The same thing happens when process E arrives – and indeed anymore processes

3. When B or C signals that it is finished nothing happens to the semaphore but the first process in the queue is removed from the queu and put into a runnable state

4. As long as there are processes in the queue – a signal from a process which has finished with resource had no effect on the semaphore but leads to processes being removed from the queue



Semaphores & Synchronisation

A arrives first

Semaphore set to zero	0	Queue
A arrives at 2 ***waits***	0	A Queue
B arrives at 1 ***signals*** & continues	0	A Queue
A removed from queue and runs	0	Queue

B arrives first

Semaphore set to zero	0	Queue
B arrives at 1 ***signals*** & continues	1	Queue
A arrives at 2 ***waits*** & runs immediately	0	Queue
	0	Queue

Semaphores & Handshaking

Handshaking

If 2 cannot proceed until 1 reaches A **and**

If 1 cannot proceed until 2 reaches B

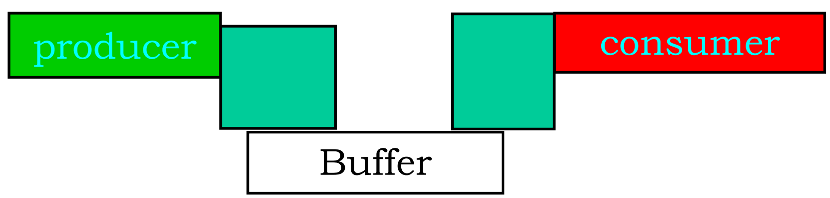
We need two semaphores
A signals **A** and then waits on B.
B signals **B** and then waits on A.

I'm here. Are you?

Note you must signal and then wait.
If you wait and then signal **deadlock** results.

Three way handshaking becomes rather tricky

Producer-Consumer



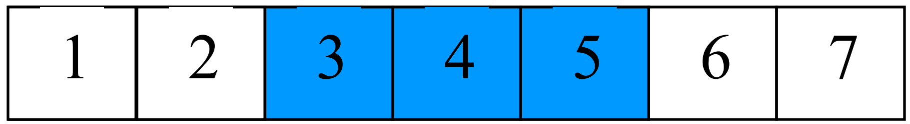
Car assembly line

Job broken sequentially. Finish one part; move to next
Room for more than one item is likely to help smooth flow of information (*especially in the case of network latencies).*
More than one leads to the idea of a circular buffer.

Circular Buffer

Area where data can be stored. Each slot is not a memory location but enough memory to store one "object"

Objects are placed in the buffer and removed in the same order

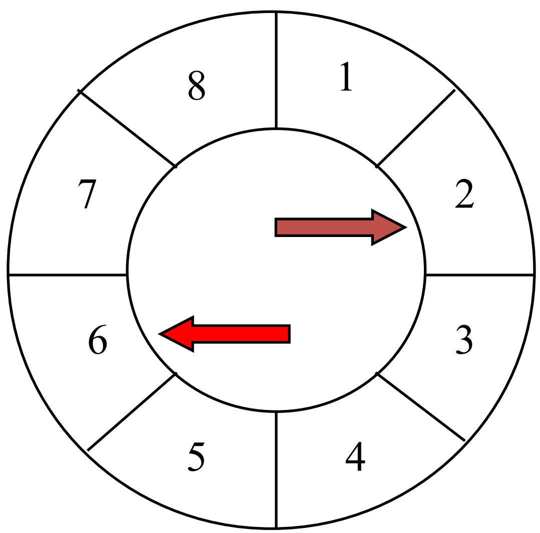


What happens when we get to the end? Loop to back to beginning. Conceptually **wrap round**. Actually pointers.

Next free slot and next full slot.

Synchronisation done by semaphores

SlotFree	initialised to the length of the buffer
ItemAvailable	initialised to 0




```
Produce Item

WAIT (SlotFree)
Put item in at NextIN
Increment NextInm
SIGNAL(ItemAvailable)
```

PRODUCER

```
WAIT (ItemAvailable)
Get item in at NextOut
Increment NextOut
SIGNAL (SlotFree)
```

CONSUMER

Buffer provides a reservoir to help maintain continuous flow

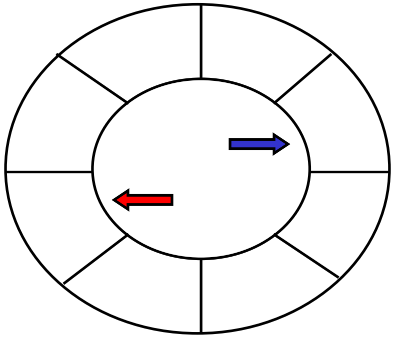
Multiple Producer Consumer

Nicely symmetrical implementation

Producer	**Consumer**

`Waits SlotFree`
`Stores in NextIn`
`Increments NextIn`
`Signals DataAvailable`
	`Waits DataAvailable`
	`Stores in NextOut`
	`Increments NextOut`
	`Signals SlotFree`

Does not work for multiple producers or consumers.



Introduce a semaphore BufferFree
Or
Two semaphores BufferFreeRead and BufferFreeWrite

Problems

1. Their use is not enforced, they can be missed by accident (or design).
2. Incorrect use can lead to deadlock
3. Semaphore can not be used to pass data.
4. Blocking is indefinite, you cannot wait for a certain length of time and then timeout.
5. Cannot wait on the and/or of more than one semaphore

Deadlock

A set of processes is in a deadlock state when every process in the set is waiting on a resource which is being held by another process in the set.

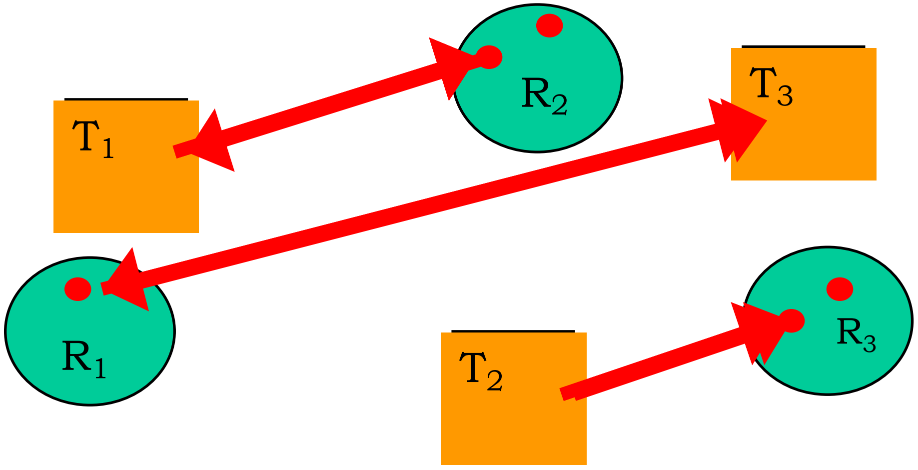
Note it must be **ALL** processes. If even one is runnable the deadlock may be breakable.

The general idea is that A is waiting for B is waiting for C is waiting for ….. is waiting for A.

A tool to discover deadlock is the resource allocation graph.

Resources are vertices of the graph.
Threads are vertices of the graph.
Request is a line from a thread to a resource
Allocation is a line from a resource to a thread

When a request is fulfilled the direction of the arrow is reversed.

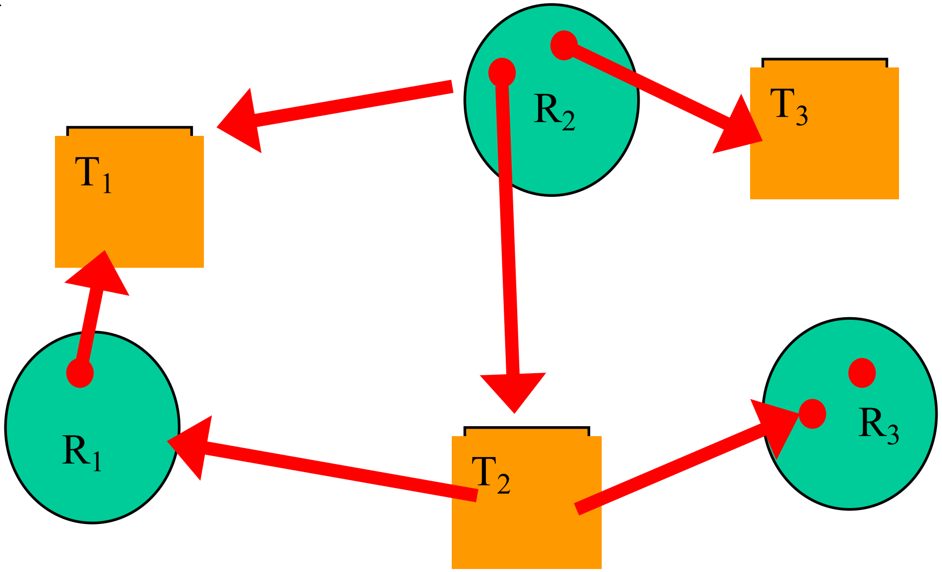


Resources which can supply more than one instance show multiple dots.

Resource Allocation Graphs

Allow you to identify the possibility of a deadlock.

If there is a closed cycle on the graph a deadlock is possible



The presence of a cycle indicates the possibility of a deadlock does not prove its existence.

The connections are arrows and have a direction.
All the arrows in a cycle have to point the same way to establish the possibility of deadlock

Can be used by the OS to detect deadlocks and break them – or stop them forming in the first place

Concurrency properties

Douglas Lea: Concurrent Programming in Java.

Lea provides a list of questions which are relevant for a concurrent application. They are worth including in documentation *and*

Good check list
Provides some guide to the problems that may occur when you try to implement concurrent programmes.

Safe
Will the method always produce its intended effect if it is called with no further checks.
this method of this object, or another method of this object may being called.
It is normally assumed that a thread-safe method means it works in a multi-threaded context, but remember
For a method to be safe implies that the caller doesn't do anything unsafe with the reference.

Thread safe is not an absolute guarentee

ConstructionSafe
Some methods not safe, but is the constructor? Must the thread constructing the object call some special initialization method to make it safe?
Singleton object.
if object exists return pointer
else create object and then return pointer.
If the "if" is much faster than the "else" we may return a pointer to a half constructed object.

Concurrency properties

Douglas Lea: Concurrent Programming in Java.

Read/Write Locks
If the object is accessed via a wrapper class which guarantees no unsynchronised access. Is it safe and live. What must be read locked and what write locked?

OwnerSafe
Is this method safe only when invoked by the thread that created it? If not are there ways of making it safe for others.

RequiresState
Safety of method conditional on it being in a state created by some sequence of operations.

RequiresLock
Safety of this method conditional on the caller holding a particular lock?

FailureSafe
Exception in this method, will subsequent calls still be safe?
Is there a way to recover the state of this object, or create a new one?
Can any exception leave the object in an unadvertised unusable state?

Concurrency properties

Atomic
Are the state changes and other effects produced by this method atomic with respect to all other methods?
Which ones are?
Are only some of the effects atomic?
Is there anything I can do do ensure atomicity with respect to those other methods or effects?

Stateless
Is this method a function?

A given message always returns the same result

Asynchronous
Do some of the effects of this method occur in other threads that need **not** complete upon method return?
Is there a way I can wait these out if I need to?

ObjectReturn (AccessorConsistent)
Are objects returned by method guaranteed ***not to be***
Stale
reveal transient illegal values?
If not what can I do to avoid these problems?

DataBase Views (ViewConsistent)
An object needs information from an (some) objects.
Is that information
synchronous - guaranteed up to date.
snapshot – correct at the time of creation
weak – at least as good as snapshot. Somethings better.
fastfail – provide snapshot if accurate, if not fail.

Even if Doug Lea uses them I don’t think you should.

Concurrency properties

WaitFree
method guarantees never to block, and not to loop more than a finite (and small) number of steps, in all circumstances?

LockFree
methods guarantee never to block, and additionally to only contain loops that will eventually terminate in all circumstances? Guarantee no good if OS provides no resources.

BoundedLocking
method guarantees not to block except due to lock contention with other threads. To use locks to cover only loopless, recursionless code & so hold them only for finite (and short) periods?

Fair
method guarantees eventual progress in the face of unbounded thread contention?
If provided with CPU.
Stronger fairness such as FIFO?

Can it be cancelled from another thread.

Cancellable
method checks interrupt or cancellation state and aborts cleanly.

SaturationLive
method complete (in some manner) somehow complete even when bounded resources are exhausted. Liveness under saturation includes aborting, shedding work, or preventing other processes producing work too slow.

TimeoutBlocking
Does this method give up after a timeout?
If so, is there any way to control the timeout value?

ConditionPolling
Does this method repeatedly poll/spin until some condition or result holds?
What can or must I do to minimize or eliminate spinning? For instance reduce the spin rate if immediate notification is not required.

TimeSensitive
Does this method have a (soft) real-time deadline?
What happens if it is not met? Fail, throw away work, reduce guarantees when it falls behind?
Dealing with deadlines is an important part of distributed computing. Includes hard deadlines for real-time control

IO
Does method block waiting for IO? Can it time-out and fail? If so, can it be retried or must it be aborted? Does the IO affect the state of local objects?

Mutex

A *mutual exclusion lock* is a way of ensuring on one thread can access something at any one time.

Can be implemented as a simple object
but with permits=1 a semaphore acts as a mutex – called a *binary semaphore* in this context.

Bounded buffers
Semaphores useful for implementing bounded buffers.
BBs are much used in Producer-Consumer.
Buffer is used to smooth out flow rate fluctuations.
BoundedBuffer makes sure the buffer doesn't overflow memory.

Buffer size *n* has *n* put permits and 0 take permits.
take must acquire a take permit and release a put permit.
put must acquire a put permit and release a take permit

Order is important

Latches: variable or condition is one which eventually receives a value from which it never again changes.
Also a *one shot.*
Uses
Completion indicators
Timing thresholds – trigger threads at a particular date
Event Indicators – some condition must be fulfilled

Bakery Algorithm

Process that wish to enter a critical section take a ticket.
Value is greater than that of all outstanding tickets
Process has its own ticket.

Value=0	does not wish to enter the section
Value>0	wishes to enter the section.

Process waits until it has the lowest number ticket.

It is (a complicated) implementation of a **mutex**
It is also free from starvation.
It is not much used because the check of lowest numbers means each waiting process has to ascertain the number of all other processes.

Introduced because it leads to a distributed mutex.

In a single machine the numbers can be directly compared. Here they must be sent in a message.

There problem of getting numbers from central repository is solved by letting every process choose their own number with the proviso it is greater than any number it knows about.

Supposedly what you do at a bakery (US?)

What about wrap around?

Actually maximum of n(n-1)/2 if halt when lower found

Main

```
loop forever
Non critical
myNum <- chooseNumber
for all other nodes
  send(request, N, myID, myNum)
await reply
critical section
For all nodes N in deferred
  remove N from defered
  send(reply, N, myID)
```

Chooses a number.
Sends to all other nodes.
When gets a reply from all
enters critical section
Finishes critical section and
sends message to all
deferred nodes.

Receive

```
Integer source, requestedNum
loop forever
  receive(request, source, requestedNum)
  highestNum = max(highestNum, requestedNum)
  if (requestedNum < myNum
    send(reply, source, myID)
  else add source to deferred
```

Node recieves request.
Is request lower?
Yes – send a reply
No – stay silent
Keep list of higher numbers

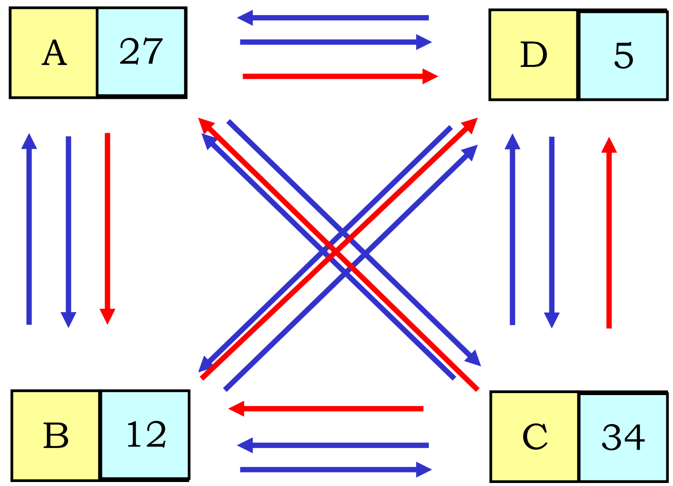
Everyone has to know about everyone else.

Sending node has to receive a reply from **all** nodes before entering critical section.

This algorithm creates a virtual queue.

Virtual Queue*

request reply



A

node	def	from
B		
C	1	1
D		

B

node	def	from
A	1	1
C	1	1
D		

C

node	def	from
A		
B		
D		

D

node	def	from
A	1	1
B	1	1
C	1	1

Virtual Queue D: 3 replies, B: 2 replies, A: 1 reply

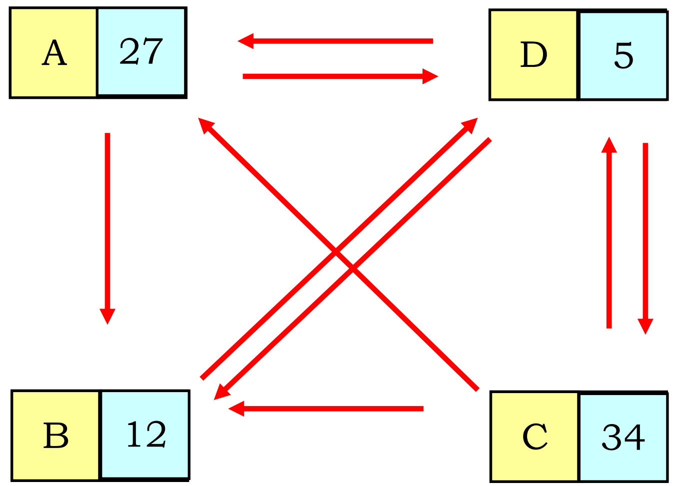
D enters critical section

Chooses a number.
Sends to all other nodes.
When gets a reply from all enters critical section
Finishes critical section and sends message to all deferred nodes.

Node recieves request.
Is request lower?
Yes – send a reply
No – stay silent
Keep list of higher numbers

...

request reply



Chooses a number.
Sends to all other nodes.
When gets a reply from all enters critical section
Finishes critical section and sends message to all deferred nodes.

A

n o d e	def	fro m
B		1
C	1	1
D		1

B

no de	d ef	fro m
A	1	1
C	1	1
D		1

C

n o de	d ef	fro m
A		
B		1
D		1

D

no de	de f	fro m
A	1	1
B	1	1
C	1	1

Node recieves request.
Is request lower?
Yes – send a reply
No – stay silent
Keep list of higher numbers

D exits critical section and sends replies to all on the deferred list
B has now replies from everyone.
Executes the critcal section and sends notifications to those on its critical list

Complications

In the situation shown the system will work.

There are a couple of refinements to get rid of possible problems.

Before a node chooses a number its number is zero.
It lower than all other numbers and so it will not send a reply.

If a node chooses a number but does not enter the critical section.
Other nodes number will pass this number and again no replies will be sent.

Solution add a flag.
Just before choosing a number set a Critical Section Flag.
Then choose number and immediately enter the critical section *when allowed.*
On exit from critical section clear flag.

The algorithm can be proved to provide
Mutual exclusion
Freedom from starvation and therefore deadlock.

It is not very efficient – too many messages.

Not trivial – but not over complex

Critical Section Problem

Each of N processes is executing in an infinite loop a sequence of instructions divided into
critical section and *non-critical section.*

It is required they satisfy the following constraints
Mutual Exclusion: statements from the critical section of two or more processes must not interleave.
No deadlock: if some processes are trying to enter their critical section, then one of them must eventually succeed.
No starvation: if *any* process is trying to end its critical section then it must succeed eventually.

Infinite loops mean that there may always be more than one process trying to enter the critical section

Applied iteratively that means something is always happening

The critical section **must** progress. A process in the critical section must eventually finish.
The non-critical section need **not** progress.

In a single multi-threaded application we *may* be able to rely on the OS to ensure *fairness.*
In a distributed system the algorithm must ensure fairness.

Single OS (multiple cores allowed) – reasonable that a free for all will be fair except in strange circumstances.
For processes separated by network links of varying speed it is easy to believe that starvation at the end of low speed links will be the norm.

Performance of Mutex Algorithms

Message Complexity
Number of messages per Crit. Sec. execution

Synchronisation delay (SD)
Time between one processor leaving the Crit. Sec. and the next one entering

Response time
The time between the point at which the Crit. Sec. message is sent out and the time at which the processor exits the Crit.Sec. So the time to decide what message to send after a request for the Crit.Sec. arrives and the actual sending of the messages is ignored; as is the time for the system to decide that the process has finished with the critical section and make an appropriate response. It is the time between a the request being sent to the distributed system and the system fulfilling the request.

Response time does depend on the complexity of the critical section

System throughput
The rate at which the system executes Crit.Sec. requests. If the time to execute the Crititcal Section is CS, then throughput is 1/(CS + SD).

Distinguish **low load** (seldom more than one request for Crit.Sec. at a time) and **heavy load** (normally at least one pending request for the Crit. Sec.

Interruptions

Reasons for ending a job

With a distributed system cancelling jobs is far more complex than just killing a process.

We can just kill the process edg-grid-cancel <job> but that may not have the desired effect

User decides to end it.

Time limited activities: search for best solution in a problem space. Split the task up and run sub-tasks. Some will have finished, but the ones that haven't may have the best answer. Straight *kill -9* risks losing that.

Solution found elsewhere: Searching a solution by many tasks. One finds the answer. Stop the others

Errors: Some error may occur on a machine which makes further progress impossible. Stop further work or save state for a restart.

Shutdown: what to do about running work for a serice shutdown.

Pre-emptive destruction is rarely a good idea. So we must have some mechanism which allows us to communicate the fact the job is required to tidy up and stop.

....

Reasons for ending a job

All the errors which are connected with killing distributed jobs on a single site are still relevant. Plus network failures.

Fail to cancel the message may fail or be delayed. The job may continue to produce output – locally (probably OK) – to central repository. Could be a problem.
Fail to respond building a respond and cancel message if response message fails can lead to whole system hanging.

token passing

Permission based algorithm, first ones in the text books

But suffer from drawbacks
inefficient if there are a large number of nodes
time consuming in the absence of contention

Still needs to communicate with all other processes

Token passing: permission to enter the critical section conferred by possession of the token.

Mutual exclusion is trivially satisfied (clear from structure)
Only one message is needed.
Once in possession of the token a process can enter and leave the critical section as often as desired with no further overheads.

Create a token free from deadlock AND starvation

Token for single line working

Ricart-Agrawala token passing algorithm.

Assumes the communication channels are FIFO

FIFO – first in first out

Two message types
REQUEST: sent to all other processes for permission to enter the critical section.
REPLY: sent to a requesting process giving permission to enter the critical section

Still send N-1 messages, receives N-1 replies. Complexity 2(N-1)

Processes use a Lamport logical clock to timestamp the requests

Logical clock – see EE5531

Each process p_i maintains a Request-Defered array, with one slot for each processor in the system.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Logical clock – see EE5531

Initially all the RD arrays are filled with zeroes (and are all identical

When processor *i* sends a defers a request from processor *j*, it sets the corresponding flag to 1.

When processor *i* sends a REPLY to processor *j*, it resets the corresponding flag to 0

Requesting Critical Section access
Processor *i* wishes to enter the critical section. It sends a message to all other processors (broadcast) with a timestamp.

When processor *j* receives a request if it is in the critical section it defers replying. If it is waiting to enter the critical section and its timestamp is smaller (earlier) than *i* it also defers response. It sets the corresponding entry in the RD table to 1.
It is not in the critical section and it does not wish to enter it sends a REPLY. If it wants to enter the critical section but its time stamp is larger (later) than *j* it sends a REPLY.

Entering Critical Section
Processor *i* can enter the critical section when it has received a REPLY from all other processors.

Exiting the Critical Section
When processor *i* leaves the critical section it sends a REPLY to all processors in the RD table and resets their entries to 0.

When a processor receives a message with a timestamp t_i it resets its own clock to be greater than t_i.

If it doesn't it can generate a request with an earlier time than a request it has replied to. (See EE5531)

Operation

The system relies on a reliable message passing system. If a communication channel fails so that processor k fails to respond, then no process will be able to enter the critical channel.

Guard against this with each processor acknowledging the receipt of a REQUEST. Increases number of messages, but in a lightly loaded system not by much. 1 message.
If every processor is waiting for the critical section then it may generate an extra N-1 messages.

As long as no more than 1 REQUEST is in flight then all the clocks will agree on the order. When they REPLY they will set their own clocks to be later than the REQUEST.

If i generates a timestamp A and then j generates a timestamp B, but because of drift in their clocks B is actually less then A. (We will see that this is not necessarily a meaningful statement).
All other processors REPLY

i will receive B, will note it is less than A and sends a REPLY. j will receive A note that it is greater than B, defer its REPLY and set its clock to be greater than B, thus sybchronising with i so it cannot generate a second request less than A

Performance

Complexity is 2(N-1) for a system with no ack and between 2N-1 and 3(N-1) for a system with ack, depending on the load on the system.

Synchronisation delay – depends mostly on the network speed and radius (how far/many hops for delivery.
Time for exiting process to deliver REPLY and receiving process to respond
Mostly a function of the network bandwidth and distance.

Response time
Include the network bandwidth, but potentially is dominated by time to execute the critical section.
By looking at the Synchronisation delay and the response time you can decide how to improve the response of a slow system.

System throughput
A different way of mixing the network speed and the critical section speed.

If the number of critical section requests exceeds the throughput, the queues will grow without limit.

For a number of different critical sections may look at weighted average of the different execution times. Single critical section may itself have different execution times, depending on context and here again a weighted mean can be used.

Virtual structure

A drawback to the Ricart-Agrawala solution is that a data structure has to be sent with the token.

The message length increases with the number of processors.

Does not have good scaling properties

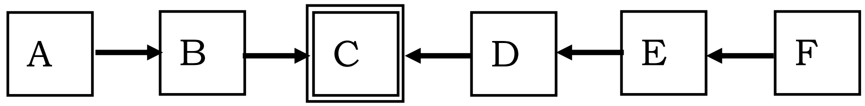
Create a distributed data structure, that doesn't have to be sent with the token.

Virtual tree

Interesting idea

Create a distributed data structure, that doesn't have to be sent with the token.

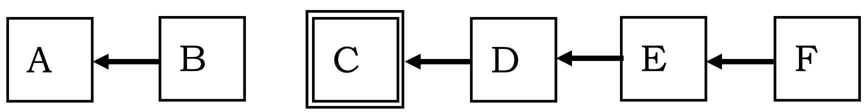
Starting anywhere following the arrows you arrive at the root.



C is the **root** of the tree
(the process with the token)

A wants to enter the critical section.
sends parent **B** a message (request, A, A)

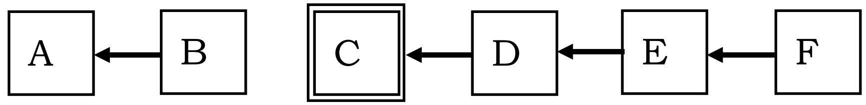
(request, message sender, message originator)



A zeroes the parent field since
A is now the root of the tree

B forwards the message to **C.**
sends parent **B** a message (request, B, A)
Changes the parent to **A**

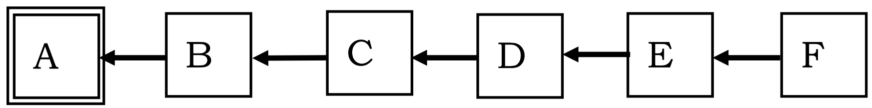
Virtual tree (ii)



A zeroes the parent field since
A is now the root of the tree

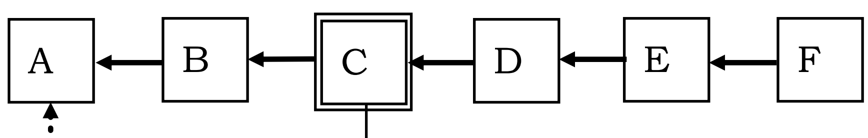
Sets parent to **B** and if **C** is not in the critical section sends token to **A** saying OK.

Token passed and root redefined



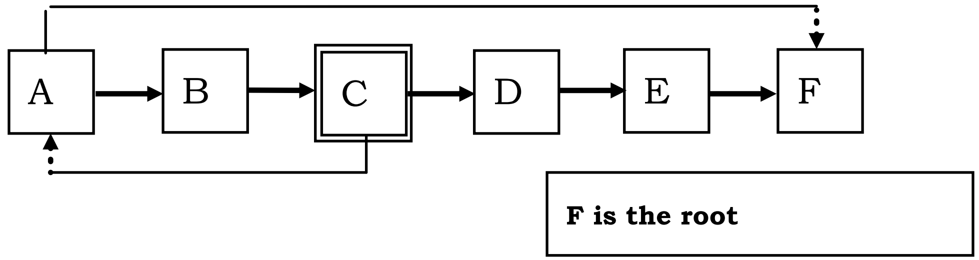
A **is now the root. All roads lead to A**

If **C** is in the critical section, sets the value of the deferred field to the originator of the message.



A **is the root**

F now requires to enter the critical section. So sends a message down the chain. This does not stop at **C** because **A** is the root

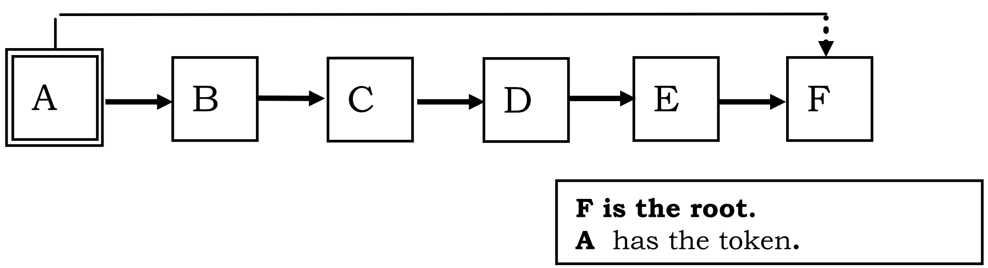


Root redefined

Message passes down the chain to **A**. **A** does not have the token so cannot return it.
A is waiting and so knows that it appears in the deferred field of some other process.
A sets deferred field to **F** and as normal makes **B** the parent.

Last requestor is the root

C exits critical section. Sends token to process in her deferred field, and zeroes the deferred field.



A now has the token and enters the critical section.
On completing can keep the token if there is nothing in the deferred field.
Actually send token to **F**, who can then enter the critical section.
The token has caught up with the root.

More efficient, maximum of N-1 hops to the root and on average less. The message is also much shorter on average.

The deferred fields create a virtual queue.

If no one is waiting a process can keep the key without further enquiry.

Idea of virtual data structure is a powerful one.

More efficient than Bakery or Ricart-Agrawala

If you want to make a queue, the natural idea is to have a centralised queuing mechanism.

This is a single point of failure and a potential bottleneck. It will certainly stop to system scaling at some point.

Here we create an effective queue, but without anything which you would recognise as a queue.

It has the behaviour of a queue without the normal queue structure.

Parallel computing may require a different method of thinking.

Compared to out original virtual queue, the size of the data structure on each machine does not increase with the number of machines

Total size of the structure increases linearly, but the amount of memory space also increases linearly with number of processors

Amount of message passing increases with number of machines.
Effect depends on the topology of the interconnection network

Critical Section

Place only one process can execute at once.
The critical section **must** progress. A process in the critical section must eventually finish.
In a distributed system can only be achieved by message passing – but message delays unpredictable and there is no place which has complete and up to date knowledge of the system

Require mutual exclusion and mutex algorithms must respect

Safety: only one process in the critical section
Live: No deadlock or starvation
Fairness if *any* process is trying to end its critical section then it must succeed eventually.

Algorithms are then judged by their efficiency – how many messages are sent. How long are the messages.

No such thing as a random number

Pseudo Random Numbers

For simulation need a set of random numbers.
But must be reproducible.

Algorithm
reproducible string of numbers
passes tests for randomness
equal probability, no correlations

A generator
Linear congruence generator
$X_{k+1} = (a X_k + c) \bmod m$

Gives numbers in the range 0 to m.
Divide X_k by m to give numbers between 0 and 1

Maximum of m numbers before repeat.

Don't *try to increase cycle length by ad hoc changes*

Parameter choice

X_0 = any positive integer
a = 16807
m = 2^{31} - 1 *or other large prime*
c = 0
Period is 2^{31} - 2 – maximum possible

X_0 = any positive integer
a = 8z + 5 *for any integer z*
m = 2^e *e positive integer*
c = 0
Period is m/4 = 2^{e-2} – normally m is the machine word size

Note if c=0 then
$X_{k+n} = (a^e X_k) \bmod m$

Which is the same form. Useful in parallel generation.

Centralized Generator

Server machine

One machine hands out random numbers to others.

Doesn't scale
Any attempt at multiple servers ends with same problem.

Not reproducible
Number delivered depends on order of arrival of request.

OK if simply one number per programme

Deliver all N required if N is known.

Random Tree Method

$L_{k+1} = (a_L L_k) \bmod m$
$R_{k+1} = (a_R R_k) \bmod m$

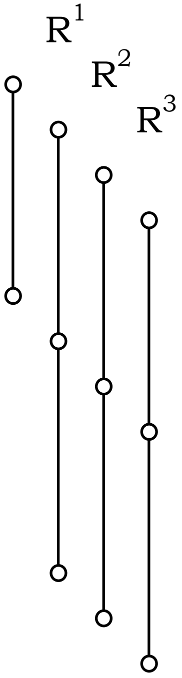
Using a single seed X_0 for both produces two random sequences.
The right generator is used for the numbers used in the calculation.
The left generator is used to generate starting numbers for the right generator.

Scaleable
Reproducible

Because the Left and Right sequences are at best length m. Left is essentially choosing a random starting point in the Right circuit.

Left and Right can clearly be interchanged

Can be correlated



By chance two starting values may be close together leaving to overlap

Leapfrog method

Leapfrog Method

If the number of generators (sequences of random numbers) needed for each is known in advance then

$$L_{k+1} = (a\,L_k) \bmod m$$
$$R_{k+1} = (a^n\,R_k) \bmod m$$

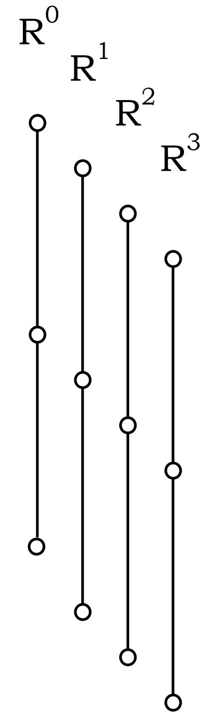
$R^i_1, R^i_2, R^i_3, R^i_4,$ is in fact
$L_i, L_{i+n}, L_{i+2n}, L_{i+3n},$ n sequences displaced by 1.

If the period of L is P then each sequence has at least P/n non overlapping values.

Also the subsequences are guaranteed to be disjoint for P/n values.

P/n should not be too short, or else the statistical properties of the numbers will no longer be random.

If n divides P it has exactly P/n values



If n is some value we can subdivide the sequence to get a hierarchy of generators

Modified leapfrog method

Modified leapfrog Method

If the maximum number of random numbers needed for each instance is known in advance **but not** the number of sequences use the modified leapfrog

$L_{k+1} = (a^n L_o) \bmod m$
$R_{k+1} = (a R_k) \bmod m$

$R^i_1, R^i_2, R^i_3, R^i_4,$ is in fact
$L_{in}, L_{in+1}, L_{in+2}, L_{in+3},$

We now have contiguous sequences of n random numbers.

Use reliable generators and use them as advertised
See
Donald Knuth, *The Art of Computer Programming: Semi-numerical Algorithms:* (Vol 2, 3rd Ed), Addison-Welsey, 1997, ISBN 0201896842