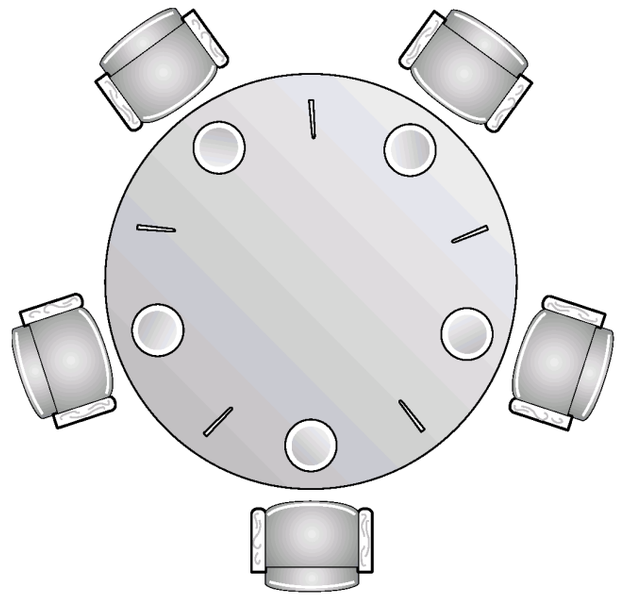
The Dining Philosophers

- A philosopher thinks and eats.
- 5 philosophers sitting around a table.
- 5 forks - 1 shared between each pair of philosophers.
- A philosopher needs the fork on either side in order to eat.
- We don't really want philosophers starving to death.



First solution

First attempted solution with semaphores.

`left` and `right` are semaphores for this philosopher's forks.

- A philosopher does this:

```
loop do
  think
  eat
end

def eat
  status = "waiting"
  right.wait
  left.wait
  status = "eating"
  right.signal
  left.signal
end
```

- What goes wrong?

Second solution

```
def eat
  status = "waiting"
  simultaneous_wait(left, right)
  status = "eating"
  simultaneous_signal(left, right)
end
```

- The simultaneous wait and signal operations are supposed to be atomic and block the thread until both forks are free.
- No more deadlock. But the problem is still not solved.

Some solutions:

- only allow 4 philosophers to pick up forks at any time
- even philosophers pick up their right forks first, odd philosophers pick up their left
- forks must be picked up lowest number first - so philosopher 4 waits for 0 first then 4 (i.e. right then left)

Simultaneous wait

We can implement a simultaneous wait with the “try lock” operation and a way of breaking out of a loop.

A “try lock” operation tries to gain the lock. If it fails the thread doesn’t block. It returns true only if the lock is granted.

`left` and `right` are simple locks (Mutexes)

```
  # Time to eat
def eat
  status = "waiting"
  loop do
    right.lock
    exitloop if left.try_lock
    right.unlock
  end
  status = "eating"
  left.unlock
  right.unlock
end
```

So just to be safe

1. Assume that threads can be interleaved at any point. Protect all access to shared data with synchronization.
2. Do not require that threads be interleaved at some point. If you need guaranteed progression between different threads you must code it explicitly using synchronisation.

Equivalence of solutions

To show that one concurrency construct (e.g. semaphores) is equivalent to another (e.g. monitors) we need to build each using the other.

e.g. semaphores can be easily implemented with monitors

```
monitor Semaphore

  def initialize(count)
    s = count
    queue = new_condition_var
  end

  def signal
      s += 1
      if s < 1
        queue.signal // condition variable
      end
  end

  def sem_wait
      s -= 1
      if s < 0
        queue.wait // condition variable
      end
  end

end
```

And the other way around

- A semaphore initialised to 1 is used to guard entrance to the monitor.
- Wait on entry, normally signal on exit.
- Condition variables complicate things
- Associate a semaphore with each condition variable.
- Only signal the semaphore when something is actually waiting.
- Need some way of querying the semaphore queue - a common addition.
- Or else keep track of this ourselves as well (no worries about mutual exclusion).
- If we wake up a thread waiting on a condition variable we don't signal the entrance semaphore as we leave.

Lock-free algorithms

- Another approach is to code so that we don't need locks.
- This is difficult – so we use standard lock free libraries of stacks, queues, sets, etc.
- They usually rely on a compare and swap type instruction (similar to test and set).

```
cas(address, old, new)
```

- If the current value at the address is the same as the old value then it replaces it with the new value and returns true.
- It returns false if the current value is not the old value.

Lock free modification

```
add_to_balance(increase):
  previous_amount = balance
  while (!cas(&balance, previous_amount,
         previous_amount + increase))
        previous_amount = balance
```

- No matter how many threads are accessing this code they will never block.
- This is not a wait-free algorithm as it is possible that a thread may stay looping indefinitely.
- There are ways of making sure the wait is bounded. Then it is wait-free as well as lock free.
- Many wait-free algorithms increase in memory size as the number of threads increases.
- There is also the hope of Transactional Memory.

Deadlock

- Multiple resources + locks introduce the danger of deadlock.
- *A circle of processes each holding a resource wanted by another process in the circle.*
- It is a local phenomena
  but it can easily spread.
- Can it be cured?
  - Not without hurting some process.
- At least one process must be forced to give up a resource it currently owns
  - (or provide a resource e.g. a message, which another process requires).

Conditions for deadlock

Havender's conditions for deadlock(1968) – 8.3.1

- There is a circular list of processes each wanting a resource owned by another in the list.
- Resources cannot be shared.
- Only the owner can release the resource
- A process can hold a resource while requesting another.

One reason deadlock is tricky is that testing may not discover it. It depends on the order of requests and allocations.

Deadlock detection

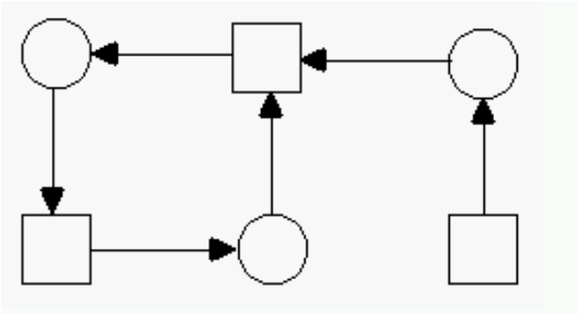
Resource graphs – 8.3.2

- All resources and processes are vertices in the graph.
- Allocations and requests are edges.

Cycles in the graph indicate deadlock (if each square holds one copy of the resource). It gets more complicated with multiple resources of the same type.

When should the deadlock detection algorithms run?

- How often do we expect deadlock?
- How many processes are usually affected?



circles are processes, squares are resources

Someone has to suffer

What do we do when deadlock is detected?

- Remove a process
  - One of the processes in the circle can be selected and removed. Its resources are returned and the deadlock is broken.
  - We could use priority or age to select the process.
  - It may not solve the problem (deadlock may occur again immediately)
    - Remove all processes involved
    - Overkill but certainly solves the problem
- Force a process to restart (or rollback to some safe state)
- If we want to rollback, the system needs to maintain checkpoints where the processes can be restarted from.
- We must ensure that the same process is not selected for restarting repeatedly.

Deadlock prevention

We make sure at least one of the conditions will not be met. i.e. It is impossible for deadlock to occur if we use prevention.

- *There is a circular list of processes each wanting a resource owned by another in the list.*
- All resources must be issued in a specific order
  if you have one of these you can't go back and request one of those.
- an alternative
  - allow requests from earlier in the ordering if all resources later than this are returned first.
- *Resources cannot be shared.*
  Make them sharable?
  Virtual devices - printer spooling

Deadlock prevention

Only the owner can release the resource

- Forcibly remove - but that causes damage

or

- if a new resource is currently busy release all currently held resources and try to get them back with the new one as well

or

- if removing the resource temporarily does no harm, e.g. a page of memory or use of the CPU (state is saved and restored)

A process can hold a resource while requesting another.

- Only allow one resource at a time?

or

- Return a group of resources before requesting another group

or

- Allocate all resources at once
  can't ask for more as the process runs

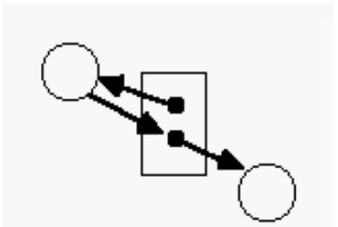
Deadlock avoidance

- Before granting requests we check if deadlock could occur if we allocate this resource to this process. “I can see deadlock might happen if I allow that. So I won’t allow that.”
- This may stop a process from getting a resource even though the resource is available.
  But doing so leads to a situation which could cause deadlock later.
  So avoidance prevents deadlock too - but dynamically as the processes run.
- System knows who has what.
  But doesn't usually know who wants what - has to use a very conservative strategy.
  Worst case scenarios of future resource requests.
- See section 8.6.2 for a simple algorithm (one of each type of resource).

Deadlock avoidance (cont.)

- e.g. Two processes P and Q and two units of the same resource R.

  Trouble only develops if both P and Q both require 2 Rs.



- Obviously no problem if they each only want one R.

  If P wants two Rs and Q has one (and doesn't want anymore), then P has to wait until Q releases its R.

  Deadlock only occurs when they both have one and both want one more.

- In this case the avoidance algorithm should not allow an allocation to the other process when one process already has an R.

The Banker's algorithm

- Dijkstra invented a deadlock avoidance algorithm known as the Banker's algorithm.
- suppose that the request has been granted
- repeat until no more processes can be finished
  - search for a process which can be given all its resources
  - return all that process's resources to the system
- If all the processes can be removed then the state is safe and the allocation can go ahead.

Banker's example

	Resource A	Resource B
P's maximum demand	1	2
Q's maximum demand	2	1
Initial state	P0, Q0	P0, Q0
P requests A	P1, Q0	P0, Q0
P requests B	P1, Q0	P1, Q0
Q requests B		
Q requests A	P1, Q1	P1, Q0
P releases A	P0,Q1	P1, Q0
Q requests B	P0,Q1	P1, Q1

e.g.

Two processes and two types of resource.

Two units of each resource.

Disadvantages

We don't usually know the maximum resource requirements of each process.

Even if we do, this algorithm has to be performed every time a process requests a resource.

Before next time

- Read from the textbook

3.4 Interprocess Communication

3.7 Examples of IPC Systems

3.7.4 Pipes

4.6.2 Signal Handling