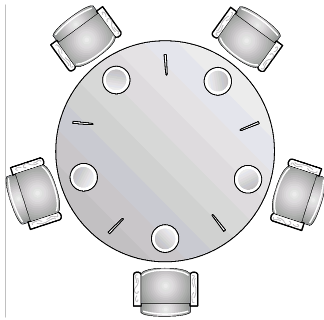
The Dining Philosophers



- •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.

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)
- These last two solutions seem to break the rules to me.

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.

<u>Condition variables complicate things</u>

- 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.

Messages

Passing messages can also be used to control concurrency.

Two (main) ways to send information from one process to another

1. Shared resource
2. Message passing

Message Passing

Needs:

- Some way of addressing the message.
- Some way of transporting the message.
- Some way of notifying the receiver that a message has arrived.
- Some way of accessing or depositing the message in the receiver.

May look like:

```
send(destination, message)
receive(source, message)
```

Or:

```
write(message)
read(message)
```

In this case we also need some way of making a connection between the processes, like an open call.

Design decisions

Should the sender block until the message is received? Should the receiver block until the message is received?

What are advantages and disadvantages of blocking or not blocking?

- Blocking reader seems natural.
- Blocking writer slows writer thread.
  - But doesn't require message buffering.
- Non-blocking writers might have to be blocked in some cases.
- If both block we have synchronous communication - rendezvous.

Should communication be one way or two way?

- Client/server requires two way

Design decisions

Should the system have message “types”. i.e., The sender can specify the type of message it is sending and the receiver can specify the type of message it wants to receive.

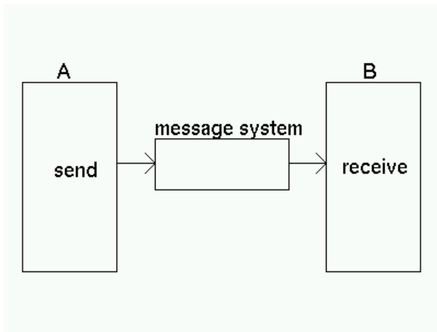
```
send(destination, type, data)
```

Should the receiver be able to wait for more than one type simultaneously.

```
message = receive(type1,type2, ...)
```

Some systems include extra conditions on message reception as well, normally known as “guards”.

Storing the message



- We want to minimise the amount of copying.

Move it straight from the sender to the receiver's address space.

Pass a pointer (sender cannot alter until it is received).

- Buffer the message in the system.

if a fixed size - reject or block senders

Message size

- should there be a fixed size? (packet or page size)

Direct communication

Process to process

- address - name or id of the other process
- one link between each pair of processes
- receiver doesn't have to know the id of the sender (it can receive it with the message)

So a server can receive from a group of processes

Disadvantages

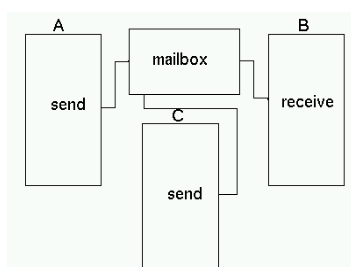
Can't easily change the names of processes. could lead to multiple programs needing to be changed.

Indirect communication

Mailboxes or Ports

Mailbox ownership

- Owned by the system
  - survives even without processes
- Owned by a process
  - the one which created it - usually the process which can receive from it
  - the creator can pass on the ability to receive
  - mailbox is removed when the process finishes



Mach ports

Underneath Mac OS X

- everything done via ports even system calls and exception handling
- only one receiver from each port
- can pass the right to receive
- when a process is started it is given send rights to a port on the bootstrap process (the service naming daemon) (and it normally gives the bootstrap process send rights via a message)
- programmers don’t usually work at that level (they can use the standard UNIX communication mechanisms)

UNIX process communication

Signals – software interrupts

```
kill(pid, signalNumber)
```

originally for sending events to the process because it had to stop.

`signalNumbers` for:

- illegal instructions
- memory violations
- floating point exceptions
- children finishing
- job control
- broken communication
- keyboard interruption
- loss of terminal
- change of window size
- user defined etc

But processes can catch and handle signals with signal handlers.

```
signal(signalNumber, handler)
```

Can also ignore or do nothing.
If you don't ignore or set a handler then getting a signal stops the process.

One signal can't be handled - 9 SIGKILL

Before next time

Read from the textbook

3.6.1 Sockets