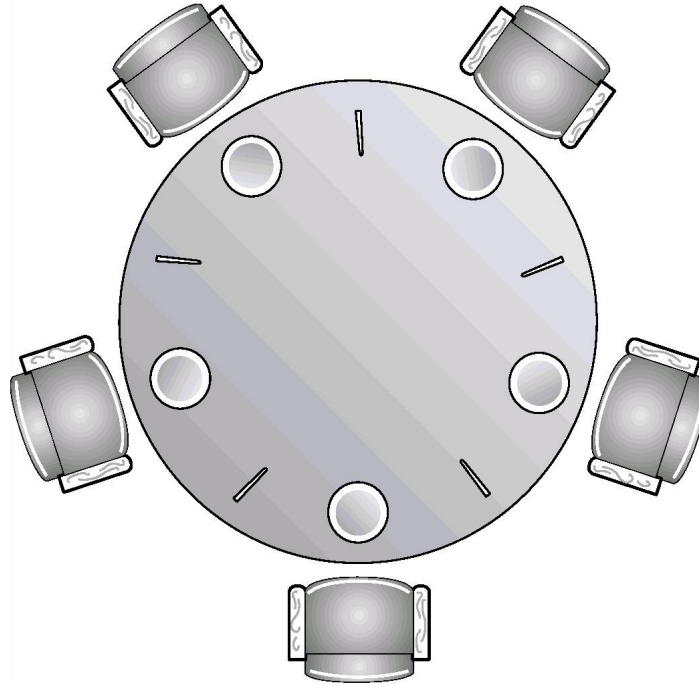


The Dining Philosophers



- 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.

```
Semaphore[] fork = new Semaphore[5];  
// assume all are constructed  
  
void philosopher (int name) {  
    // numbered 0 to 4  
    while (true) {  
        think();  
        fork[name].semWait(); // right fork  
        fork[(name + 1) % 5].semWait();  
        eat();  
        fork[name].semSignal();  
        fork[(name + 1) % 5].semSignal();  
    }  
}
```

See badDiningphilosophers.rb.

What goes wrong?

Second solution

```
while (true) {  
    think();  
    simultaneousWait(fork[name],  
                     fork[(name + 1) % 5]);  
    eat();  
    simultaneousSignal(fork[name],  
                       fork[(name + 1) % 5]);  
}
```

The simultaneousWait 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:

- use of states for the philosophers - thinking, hungry, eating
- a hungry philosopher gets preference over one which is thinking
- 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

Ruby simultaneous wait

Put this eat routine into the
 badDiningPhilosopher.rb file.
It *usually* works, but is not guaranteed.

```
# Time to eat
def eat
  @status = "waiting"
  loop do
    @right.lock
    break 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 (e.g. in Ruby)

```
require 'monitor'
```

```
class Semaphore < Monitor
```

```
  def initialize(count)
    super
    @s = count
    @queue = new_cond()
  end
```

```
  def sem_signal
    synchronize do
      @s += 1
      if @s < 1
        @queue.signal
      end
    end
  end
```

```
  def sem_wait
    synchronize do
      @s -= 1
      if @s < 0
        @queue.wait
      end
    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.

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

Need:

- 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.

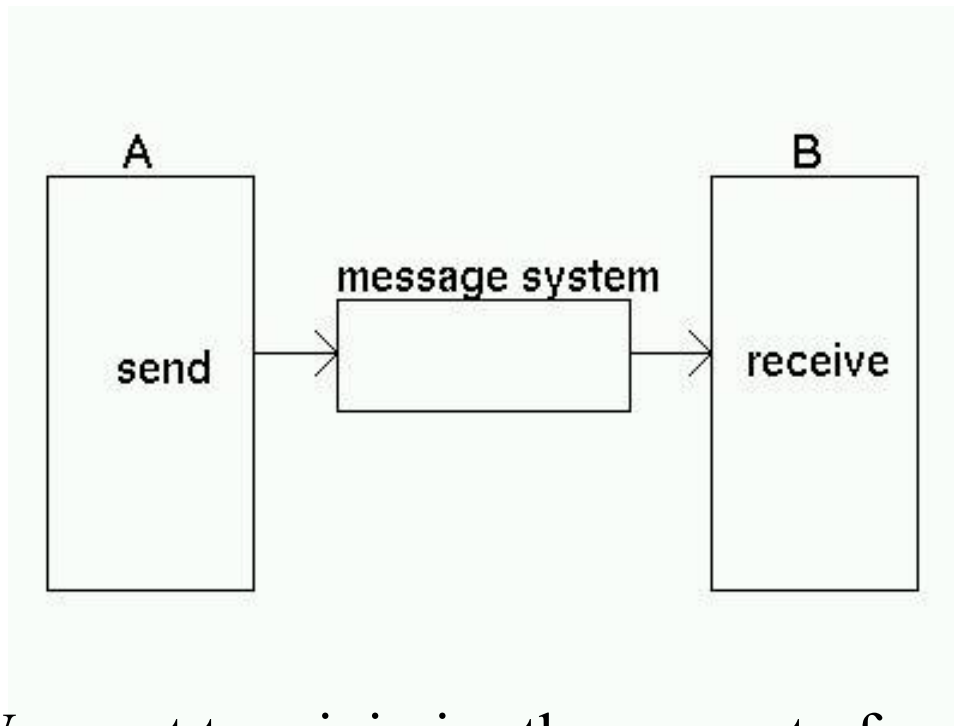
```
send(destination, type, data)
```

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

- fixed size - easier to implement, harder to use
- any size - harder to implement, easier to use

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

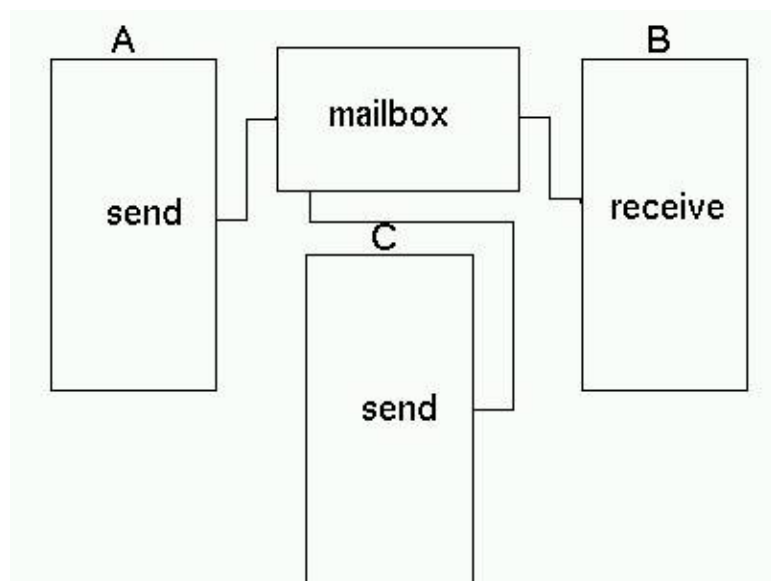
Can't easily change the names of processes.
could lead to multiple programs needing to be changed.

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 (and it normally gives the bootstrap process send rights via a message)
- programmers don't usually work at that level, they use RPC instead (more on that later)

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

Pipes

Data gets put into the pipe and taken out the other end

- implies buffering mechanism
- what size pipe?
- what about concurrent use - can writes interleave? etc

In UNIX it starts as a way for a process to talk to itself.

```
int myPipe[2];
```

```
pipe(myPipe);
```

System call which creates the pipe. Returns two UNIX file descriptors.

```
myPipe[0] to read, myPipe[1] to write  
e.g. write(myPipe[1], data, length);
```

Empty and full pipes

- Reading processes are blocked when pipes are empty
- Writing processes are blocked when pipes are full

Pipes (cont.)

Broken pipes

- A process waiting to read from a pipe with no writer gets an EOF.
- A process writing to a pipe with no reader gets signalled.

Writes are guaranteed to not be interleaved if they are smaller than the size of the pipe. The pipe size used to be 4096 but is now variable.

Limitation

- Can only be used to communicate between related processes. (Named pipes or FIFO files can be used for unrelated processes.)
 - The file handles are just low integers which index into the file table for this process.
 - The same numbers only make sense in the same process (or in one forked from it).

Sockets

Interprocess connection in a distributed environment.

Socket communication domains

- UNIX domain (can be used to implement pipes)
names are filenames
- Internet domain
names are IP addresses, names or numbers
plus port number
- NS domain (Xerox communication protocols)
- ISO OSI protocols

types

- stream – bidirectional, reliable, sequenced, unduplicated. No record boundaries. Just like pipes.
- datagram – bidirectional, but not reliable, sequenced or unduplicated. Record boundaries are preserved. (Packet switched networks like Ethernet.)
- raw – access to the underlying protocols which support sockets
- sequenced packet – like a stream socket but with record boundaries preserved.

Socket calls

Setting up a socket

socket - make a socket, specify the domain and protocol

bind - associate a name with the socket

listen - now ready to get connections

accept - gets a connection and returns a new socket (used for the actual communication)

another process (for the other end of the socket):

socket - make a socket

connect - makes the connection between this socket and the named one

Then normal read and write operations can be performed on the socket.

Only one process bound to each port.

select – can be used to read from multiple sockets when data becomes available.

Before next time

Read from the textbook

7.6.3 The Dining-Philosophers Problem

4.5 Interprocess Communication

5.3.3 Signals (*20.3.3*)

4.6.1 Sockets (*15.1, 20.9.1*)