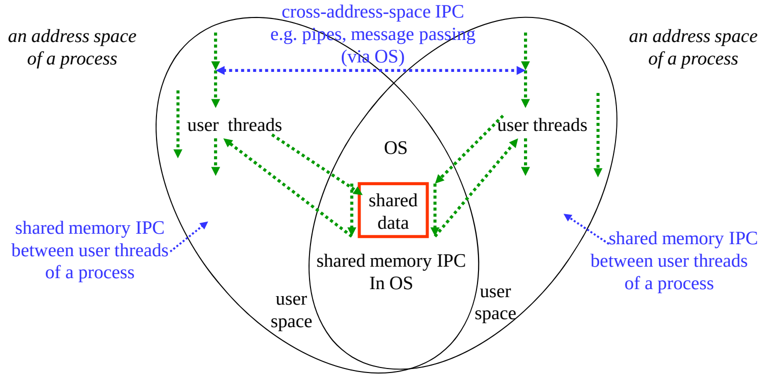
Inter-process communication (IPC)

- We have studied IPC (strictly, inter-thread communication) via shared data in main memory.
- Processes in *separate address spaces* also need to communicate.
- Consider system architecture – both shared memory and cross-address-space IPC is needed
- Recall that the OS runs in every process address space:



Concurrent programming paradigms – Overview

IPC via shared data – processes/threads share an address space – we have covered:

1. shared data is a passive object accessed via concurrency-controlled operations:
   conditional critical regions, monitors, pthreads, Java
2. active objects (shared data has a managing process/thread)
   Ada select/accept and rendezvous

We now consider: Cross-address-space IP

Recall UNIX pipes – covered in Part 1A case study

Message passing – asynchronous – supported by all modern OS

Programming language examples:

Tuple spaces (TS)

Erlang – message passing between isolated processes in shared memory
generalises to cross-address-space and distributed message passing

Scala (Michael Oderski) DEBS 2012 keynote
"Actors reloaded – from Scala Actors to Akka"

Kilim CL PhD 2010, TR 769 Sriram Srinivasan
"A server framework with lightweight actors"

Message passing – synchronous e.g. occam

Can these be extended to use for distributed programming?

UNIX pipes outline - revision

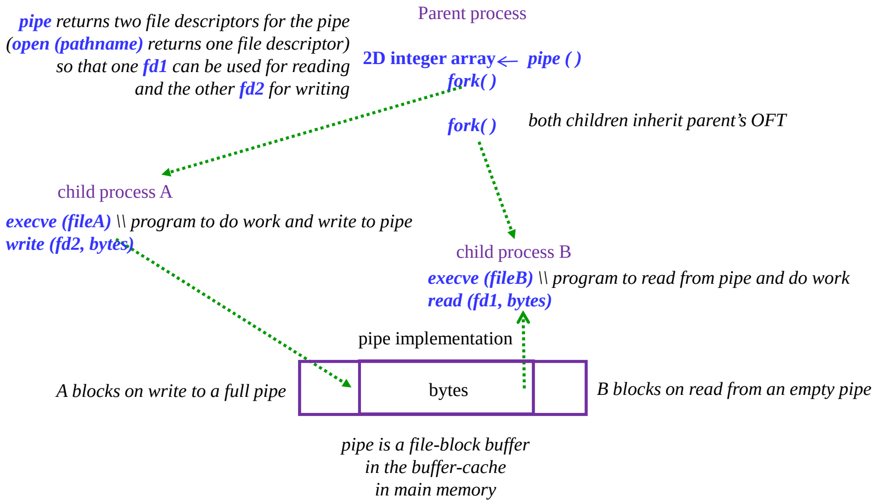
A UNIX pipe is a synchronised, inter-process byte-stream
A process attempting to ***read*** bytes from an empty pipe is blocked.
There is also an implementation-specific notion of a "full" pipe
- a process is blocked on attempting to ***write*** to a full pipe.
(recall – a pipe is implemented as an in-memory buffer in the file buffer-cache.
The UNIX designers attempted to unify file, device and inter-process I/O).

To set up a pipe a process makes a ***pipe*** system call and is returned two file descriptors in its open file table. It then creates, using ***fork*** two children who inherit all the parent's open files, including the pipe's two descriptors.

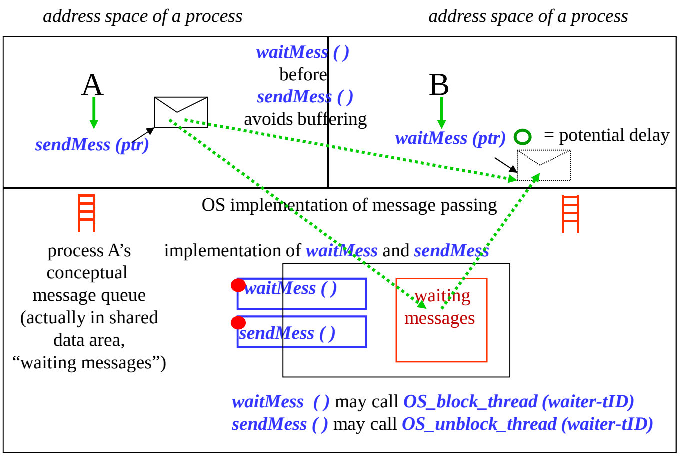
Typically, one child process uses one descriptor to ***write*** bytes into the pipe
and the other child process uses the other descriptor to ***read*** bytes from the pipe.
Hence: pipes can only be used between processes with a common ancestor.

Later schemes used "named pipes" to avoid this restriction.
UNIX originated in the late 1960s, and IPC came to be seen as a major deficiency.
Later UNIX systems also offered inter-process message-passing, a more general scheme.

Unix pipes visualisation



Asynchronous message passing implementation



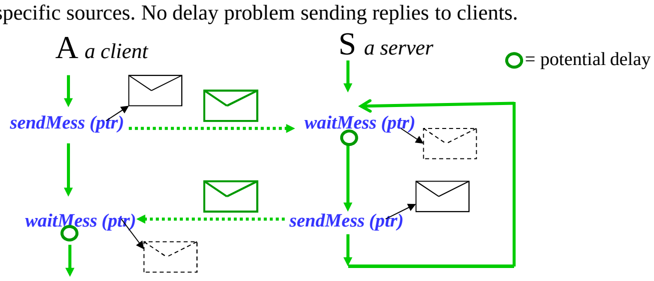
Asynchronous message passing: Client-Server

Note no delay on ***sendMess*** in asynchronous message passing (OS buffers if no-one waiting)

Note cross-address-space IPC *implemented* by shared memory IPC in OS (see previous slide)

Details of message header and body are system and language-specific
e.g. **typed messages at language level** (e.g. Java JMS)
At OS-level, message transport probably sees a header plus unstructured bytes.

Server with many (unknown) clients: Need to be able to wait for a message from ***"anyone"***
as well as from specific sources. No delay problem sending replies to clients.



Programming language example: Tuple spaces 1

Since Linda was designed (Gelernter, 1985) people have found the simplicity of tuple spaces (TS) appealing as a concurrent programming model. TS is logically shared by all processes.

Messages are programming language data-types in the form of tuples
e.g. ***("tag", 15.01, 17, "some string")***

Each field is either an expression or a formal parameter of the form ***? var***,
where ***var*** is a local variable in the executing process

sending processes write tuples into TS, a non-blocking operation
out ("tag", 15.01, 17, "some string")

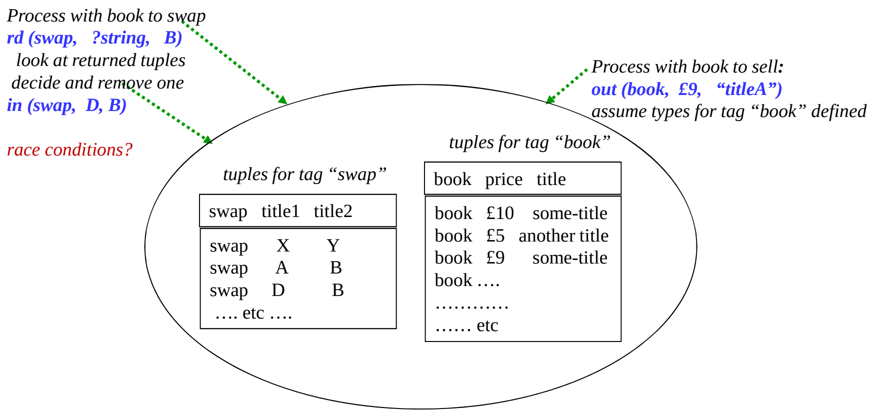
receiving processes read with a template that is pattern-matched against the tuples in the TS

reads can be non-destructive: ***rd ("tag", ? f, ? i, "some string"),*** which leaves the tuple(s) in TS

or destructive: ***in ("tag", ? f, ? i, "some string"),*** which removes the tuple from TS

A read blocks if a matching tuple is not found (setting a ***'notify'*** was added later)

Tuple space visualisation ?



Concurrency control? Does TS have a manager through which to single-thread messages?
Where is the TS? Main memory for performance – what about crashes?
Need to achieve persistence of results of ***"out" "in"*** (Transactions again – lectures 6, 7).
New subjects make TS obsolete - from DB: in-memory databases, column stores, key-value stores for "big data" ?
- ? publish/subscribe message passing for communication in distributed systems

Programming language example: Tuple spaces 2

Even in a centralised implementation, scalability is a problem:

- inefficient: the implementation needs to look at the contents of all fields, not just a header
- protection is an issue, since a TS is shared by all processes.
- naming is by an unstructured string literal ***"tag"*** - how to ensure uniqueness?

Several projects have tried to extend tuple spaces for distributed programming
e.g. JavaSpaces within Jini, IBM Tspaces, and various research projects.
Destructive reads are hard to implement over more than a single TS,
and high performance has never been demonstrated in a distributed implementation.

Programming language example: Erlang

Erlang is a functional language with the following properties:

1. single assignment – a value can be assigned to a variable only once, after which the variable is immutable

2. Erlang processes are lightweight (language-level, not OS) but share no common resources.
New processes can be forked (***spawn***ed), and execute in parallel with the creator:
Pid = spawn (Module, FunctionName, ArgumentList)
returns immediately – doesn't wait for function to be evaluated
process terminates when function evaluation completes
Pid returned is known only to calling process (basis of security)
Pid is a first class value that can be put into data structures and passed in messages

3. asynchronous message passing is the only supported communication between processes.
Pid ! Message
! means send
Pid is the identifier of the destination process
Message can be any valid Erlang term

Erlang came from Ericsson and was developed for telecommunications applications.
It is becoming increasingly popular and more widely used.

Erlang – 2: receiving messages

The syntax for receiving messages is (recall guarded commands and Ada active objects):

```
receive
    Message1 ( when Guard1) ->
        Actions1 ;
    Message2 ( when Guard2 ) ->
        Actions2 ;
    ..........
end
```

Each process has a mailbox – messages are stored in it in arrival order.

Message1 and ***Message2*** above are patterns that are matched against messages in the process mailbox. A process executing ***receive*** is blocked until a message is matched.

When a matching ***MessageN*** is found and the corresponding ***GuardN*** succeeds, the message is removed from the mailbox, the corresponding ***ActionsN*** are/is evaluated and ***receive*** returns the value of the last expression evaluated in ***ActionsN***.

Programmers are responsible for making sure that the system does not fill up with unmatched messages.

Messages can be received from a specific process if the sender includes its ***Pid*** in the pattern to be matched: ***Pid ! {self( ), abc}***

receive {Pid, Msg}

Erlang – 3: example fragment

Client:

PidBufferManager ! { self ( ), put, <data> }

PidBufferManager ! { self ( ), get, <pointer for returned data> }

Buffer Manager:

receive {PidClient, put, <data> } (buffer not full)

insert item into buffer and return

{PidClient, get, <pointer for returned data> } (buffer not empty)

remove item from buffer and return it to client

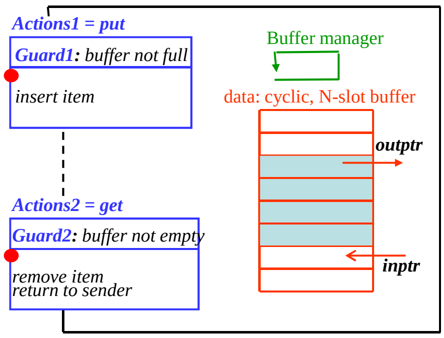
Example: Producers/Consumers with Erlang

Mailbox of Buffer Manager to ***receive*** from: (FCFS order)

PidBufferManager ! { self ( ), put, <data> }
PidBufferManager ! { self ( ), put, <data> }
PidBufferManager ! { self ( ), put, <data> }

PidBufferManager ! { self ( ), get, <ptr for returned data> }
PidBufferManager ! { self ( ), get, <ptr for returned data> }
PidBufferManager ! { self ( ), get, <ptr for returned data> }

self( ) gives sender's ***PID*** for any return message



managing process searches through mailbox (FCFS order)
- It matches ***put*** when ***Guard 1*** is true, matches ***get*** when ***Guard2*** is true
- Asynchronous message passing is the only IPC supported
- ACK may be returned in a ***message*** after a ***put***
- data is returned in a ***message*** after a ***get***

Erlang - 4: further information and examples

Part 1 of Concurrent Programming in Erlang is available for download from
http://erlang.org/download/erlang-book-part1.pdf

The first part develops the language and includes many small programs, including distributed programs, e.g. page 89 (page 100 in pdf) has the server and client code, with discussion, for an ATM machine.

The second part contains worked examples of applications, not available free.

A free version of Erlang can be obtained from
http://www.ericsson.com/technology/opensource/erlang

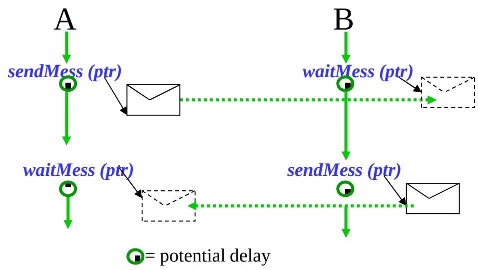
Erlang also works cross-address-space, and distributed.
e.g. Steve Vinoski's "favourite language of all time"
ACM Middleware conference keynote 2009

Synchronous message passing -1

Delay on both ***sendMess*** and ***waitMess*** in synchronous message passing

Sender and receiver "hand-shake" - OS copies message cross-address-space

Note no message buffering in OS



Synchronous message passing - 2

Designed for pipelines of processes, known statically e.g. in embedded systems.

For client-server?

How to avoid busy servers being delayed by non-waiting clients (on sending answer)?

- use ***fork***
- build buffers at application-level

(difficult to program – which way to synchronise, with next in or with next out?)

Not suitable for general client-server programming

Synchronous message passing example – occam 1

In occam communication takes place via named ***channel***s. IPC is equivalent to assignment from one process to another, so for ***variable := expression***, the destination process can hold the variable and the source process evaluates the expression and communicates its value:

destination process (***?*** = input from channel)   source process (***!*** = output to channel)

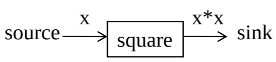
	channel ? variable	***channel ! expression***
e.g.	***channelA ? x***	***channelA ! y+z***

input, output and assignment statements may be composed

sequentially using ***SEQ*** or in parallel using ***PAR***

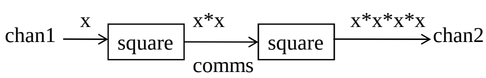
Synchronous message passing example – occam 2

```
PROC square ( CHAN source, sink )
    WHILE TRUE
      VAR x
       SEQ
          source ? x
          sink ! x*x
```



PROC is a non-terminating procedure that takes a value from channel ***source*** and outputs its square on channel ***sink***. We might then make a parallel composition:

```
CHAN comms:
  PAR
  square ( chan1, comms )
  square ( comms, chan2 )
```



Synchronous and asynchronous systems

Historically, synchronous systems were favoured for theoretical modelling and proof.
e.g. occam was based on the CSP formalism, Hoare 1978

occam enforces static declaration of processes - more applicable to embedded systems than general purpose ones: "assembly language for the transputer".
Current applications need dynamic creation of large numbers of threads.

In practice, asynchronous systems are used when large scale and wide distribution are needed.
Asynchronous message passing is more general and more widely used than Remote Procedure Call (RPC).

NEXT
Moving on to
liveness properties, deadlock,
composite operations and transactions