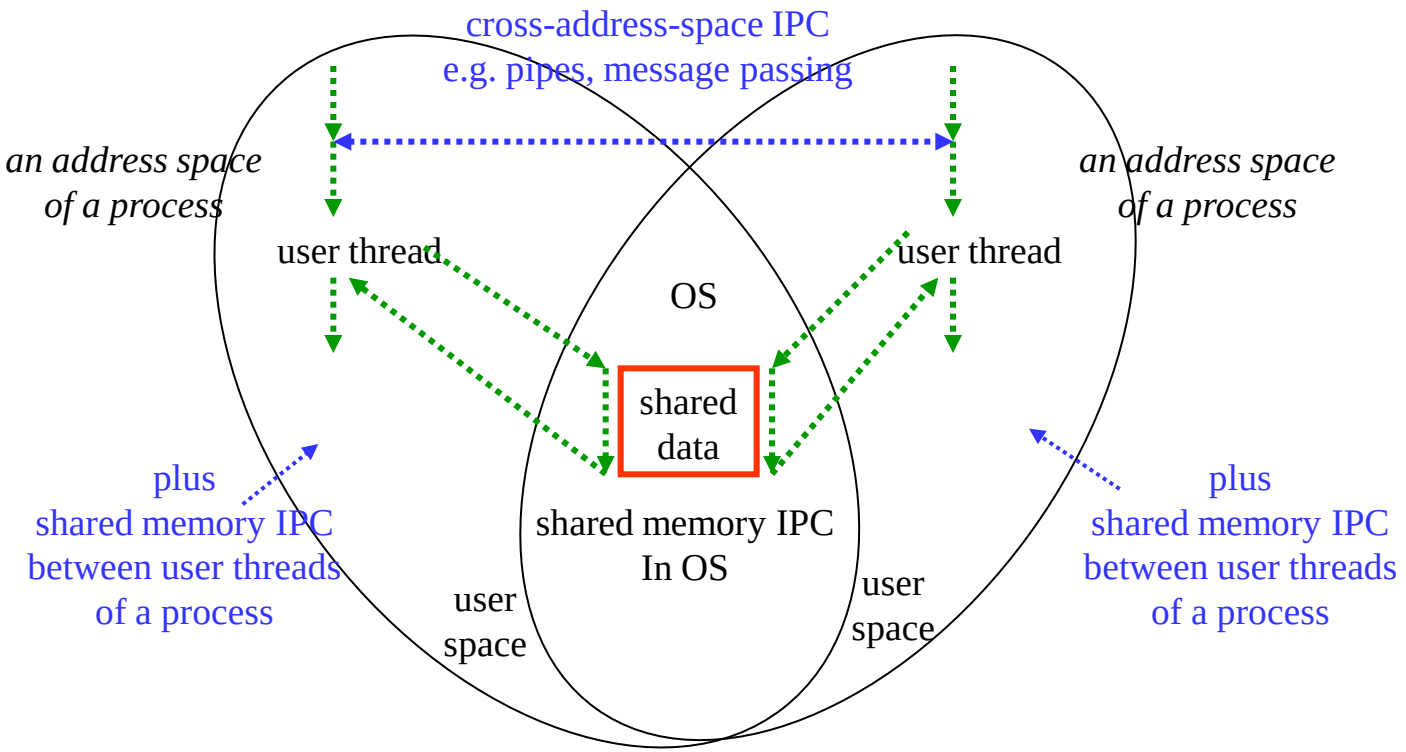
Inter-process communication (IPC)

- We have studied IPC 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 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
3. lock-free programming

We now consider: Cross-address-space IPC

Recall UNIX pipes – covered in Part 1A case study

Message passing – asynchronous – supported by all modern OS
programming language example: Erlang
tuple spaces e.g. Linda
Kilim – a Java extension for shared memory or cross-address-space message passing.

Message passing – synchronous e.g. occam

Consider which of these might be used 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.

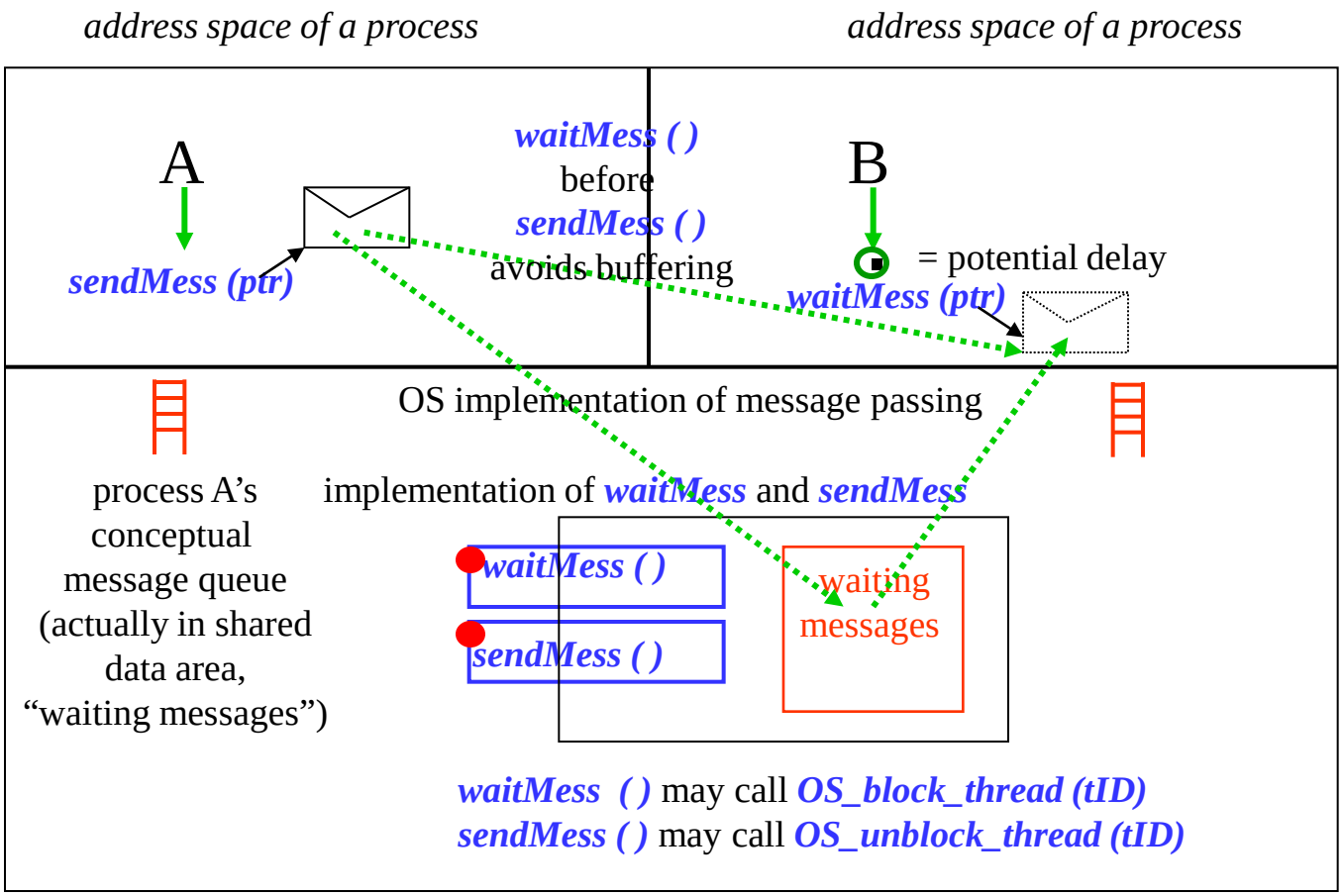
(recall – a pipe is implemented as an in-memory buffer in the file buffer-cache.
The UNIX designers attempted to unify file, device and inter-process I/O).

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.

Asynchronous message passing - 1



Asynchronous message passing -2

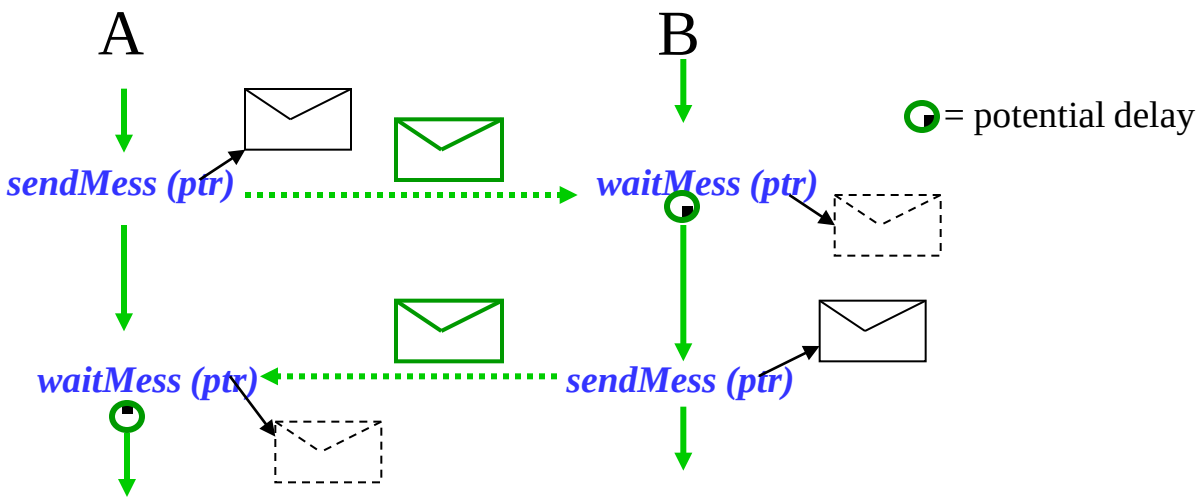
Note no delay on ***sendMess*** in asynchronous message passing (OS buffers if no-one waiting)

Note cross-address-space IPC *implemented* by shared memory IPC in OS

Details of message header and body are system and language-specific e.g. typed messages.
At OS-level message transport probably sees a header plus unstructured bytes.

Need to be able to wait for a message from "anyone" as well as from specific sources
e.g. server with many clients

Client-server interaction easily set up
e.g. needed for system services in microkernel - structured OS.



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) and 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 is used for telecommunications applications.

Erlang continued – 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 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 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*

Tuple spaces

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

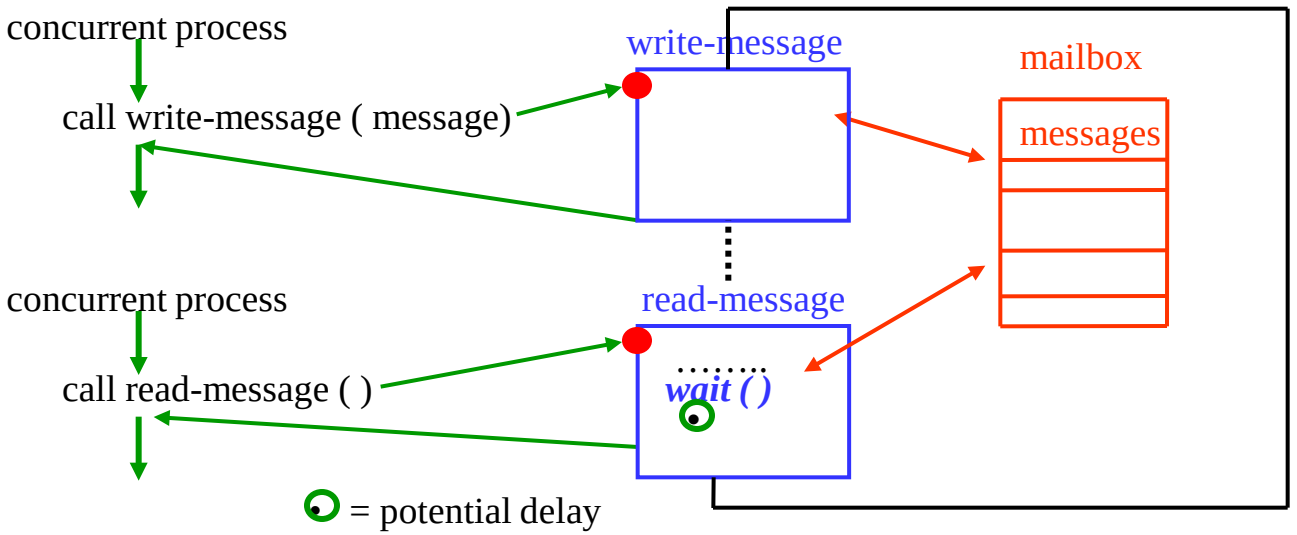
Even in a centralised implementation, scalability is a problem:

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

Several projects have tried to extend tuple spaces for distributed programming
e.g. JavaSpaces within Jini, IBM Tspaces, 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.

Kilim – *shared address-space* message passing

Kilim extends Java via annotations and static checking. A mailbox paradigm is used.



- after writing a message into a mailbox the sending process/thread *loses all rights to that data*
- the receiver/reading-process gains rights. Based on linear type theory.
- in the current shared memory implementation, threads share an address space
  messages are not physically copied but pointers are used i.e. mailboxes contain pointers
- good performance demonstrated for large numbers of threads cf. Erlang
- the motivation for Kilim is that in classical concurrency control, methods that execute under exclusion may make library calls, making it impossible for a compiler to carry out static checking of exclusive access to data.

Kilim further information

Sriram Srinivasan's PhD research
After some 12 years' experience of developing web servers e.g. WebLogic, Ram wanted more efficient, but safe, concurrent programming with large numbers of threads.
The work is inspired by Ada and Erlang, but extending Java gives a better chance of acceptance and use.

For an overview, publications and download, see:
http://www.malhar.net/sriram/kilim/

Ben Roberts' PhD is to distribute Kilim, ... among other things ...

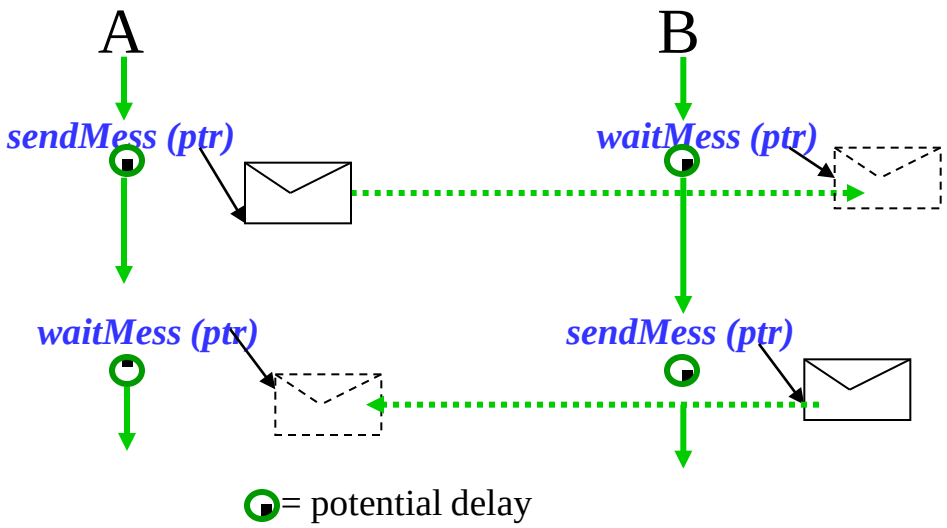
Synchronous message passing -1

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

How to avoid busy servers being delayed by non-waiting clients (on sending answer)?
buffers could be built at application-level
but synchronous message passing is not appropriate for client-server programming.



Synchronous message passing example – occam

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 holds the variable and the source process evaluates the expression and communicates its value:

destination process (***?*** = input from channel)
channel ? variable
e.g. ***channelA ? x***

source process (***!*** = output to channel)
channel ! expression
channelA ! y+z

input, output and assignment statements may be composed sequentially using SEQ or in parallel using PAR

```
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 distribution are needed.

See your theory courses for modelling concurrency and distribution.