Process communication Lecture 1

Something interesting to do with PCBs

Messages - 4.6

Events - signals 19.3.3

PCBs and processors

Hardwired PCBs cause problems for processor designers

MMX instructions and floating point registers

- didn't want to alter the OS

Solutions:

- parameterized PCB
- device driver type save and restore routines

Messages

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

Shared resource or message passing

Fundamentally:

- send(message)
- receive(message)

We need

- method to address the message
- (possibly only at open time - then use ordinary file reads and writes)
- method to store the message

Should…

- the sender block? - usually not, similar to a file write
- the receiver block? - usually, similar to a file read
  - if both block we have synchronous communication - rendezvous
  - asynchronous when receives happen at a different time from sends
- communication be one way or two way?

<u>Storing the message</u>

don't worry, move it straight from the sender to the receiver's address space

- pass a pointer (sender cannot alter until it is received)
- we usually want to minimise the amount of copying - slows communication noticeably

buffer the message in the system

- fixed size - reject or block senders
- any size - stored as a file

How big should the messages be?

fixed size - easier to implement, harder to use

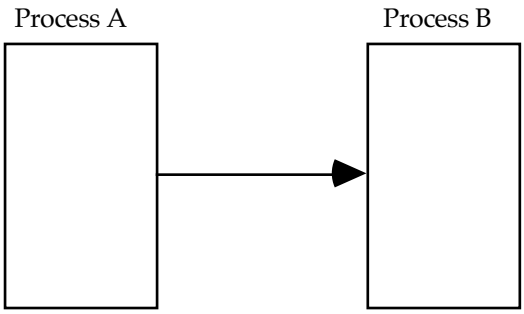
any size - harder to implement, easier to use

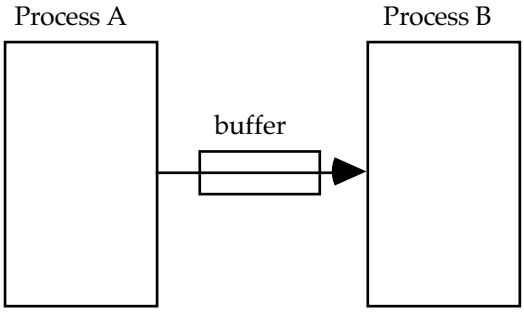
Process to Process - direct communication

address - name or id of the other process

send(toProcess, message)

receive(fromProcess, message)





one link between each pair of processes

receiver doesn't have to know the id of the sender it can receive it

i.e. the fromProcess parameter gets filled in with a value

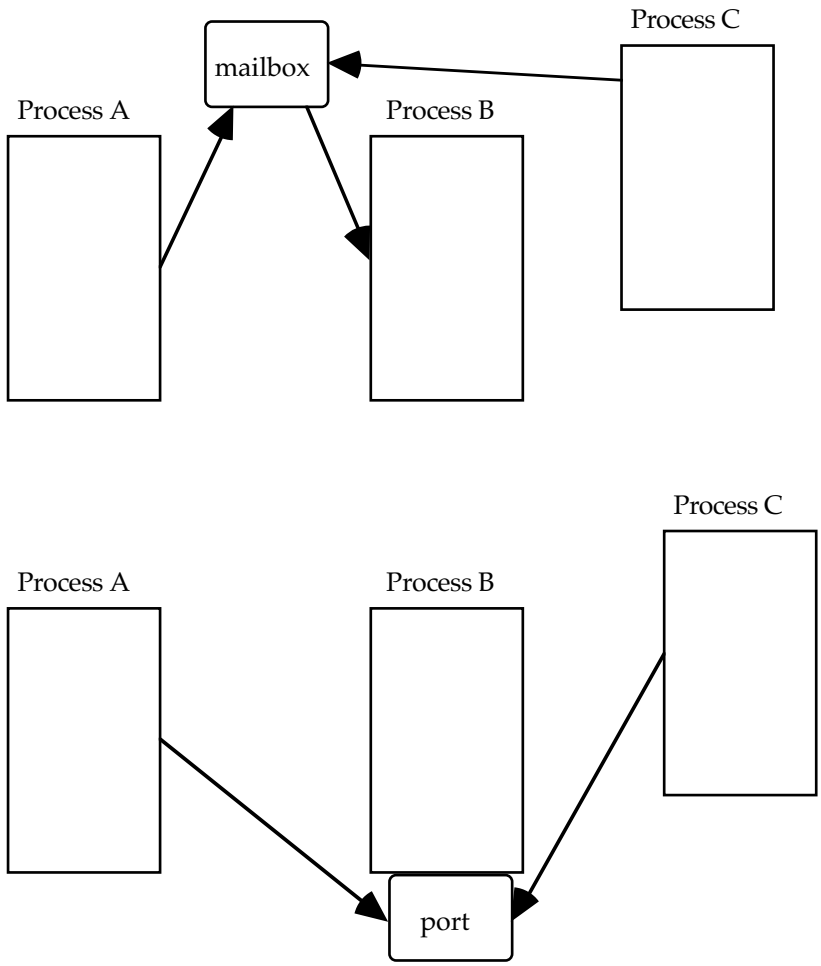
So a server can receive from a group of processes

Can't easily change the names of processes

could lead to multiple programs needing to be changed

Mailboxes or Ports - indirect communication

messages sent to and received from the mailbox



can also receive from a group of processes - useful for a server

can be several mailboxes used between the same pair of processes

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

everything done via ports even system calls and exception handling

only one receiver from each port - we know who gets the message

giving another process the right to receive from a port means the original can no longer receive

How can you get multiple receives i.e. same message read by different processes in this scheme?

Network wide

Probably use mailboxes

could use direct addressing - name must be unique - possibly machine:pid

client/server architecture

Problems

sender not available - has it gone or hasn't it started yet?

OS should notify or remove.

receiver not available - has it gone or hasn't it started yet?

with non-blocking sends it doesn't matter

lost messages

timeouts - sends always get replies

must cope with multiple messages

Sometimes we just need to know that something has happened

Event information

Just what semaphores and condition variables and event counters and locks provide

but the sending (signalling, notifying) process doesn't know who it is sending to

Can send such information with ordinary messages - a send acts like an unlock

the waiting process is waiting with a receive

How can the sender (which doesn't block on send) know when the message has been received?

UNIX 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) - actually the POSIX sigaction

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

When a signal is sent the kernel sets a flag in the destination process's PCB.

Just as this process is about to return to user running it jumps off and handles the signal instead of returning to its saved context.

so the signal is handled in the context of the process signalled

in a single processor - it cannot be handled immediately

Just a reminder

Traditional UNIX and preserving system integrity

Critical sections of the kernel are preserved by stopping preemptive scheduling.

Not the same as stopping interrupts - we still want these to occur.

Modern versions use semaphores or similar locks.

Kernel can have multiple active threads operating inside it.

Race condition in early versions

the field indicating the handler was cleared by the OS when the handler was called

the process had to call signal again to set the handler up

but a signal could come before the new call to signal

killing the process

No longer the case - signals can be blocked while being handled.