Distributed deadlock

- Everything gets just a bit more complicated when dealing with distributed systems.
- We can still prevent deadlock by resource ordering.
- Or we can prevent deadlock by process ordering, only allowing processes with higher priorities to wait for resources. Processes with lower priorities get rolled back.
- But this quickly leads to starvation.
- Or we can avoid deadlock by the Banker's algorithm, use one process as the banker:
  - Even more expensive.

More processes, more resources, all requests have to be checked by a Banker process.

Time-stamp prevention methods

- We prevent a cycle by only allowing older processes to wait for resources held by younger ones or vice versa. Rather than resource ordering this is process ordering.

wait-die

- Process A requests a resource held by process B
  If process A is older than process B it waits for the resource.
  Otherwise process A restarts, process B (the older) continues.
- Older processes hang around in the system (they have done more work). Age has its privileges.
- Younger processes may have to restart multiple times, the resource might still be busy (but they eventually get old too, they retain their original timestamps).

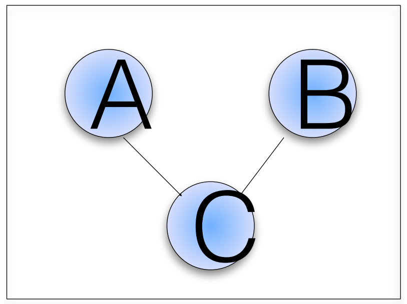
Another time-stamp method

wound-wait

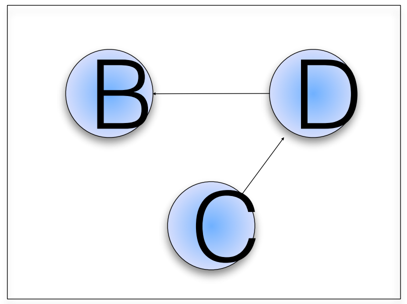
- Once again A wants a resource held by B.
- If A is older than B it takes the resource and B restarts.
  Otherwise A (the younger) waits for B to release the resource.
- Old processes never wait for anything. Age really has its privileges.
- Less restarts.
- In both cases processes keep their timestamps even when restarted.
  Eventually they are really old and will not have to restart.
- Either way lots of unnecessary restarts.

Distributed deadlock detection

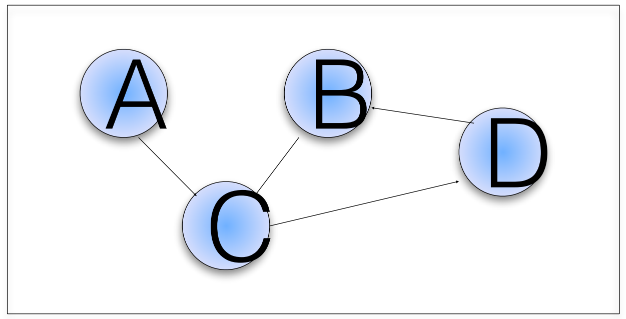
- Each processor keeps track of the resource allocation graph to do with its local resources.
  May include remote processes.
  Cycles don't just occur locally.
  Need to check the union of resource allocation graphs.



Site 1



Site 2



Global deadlock

DDD (cont.)

Centralized deadlock detection process.

- Information may have changed by the time the data from the last machine is gathered, the data from the first machine is probably out of date.
- Graph is only an approximation of the real allocation of resources and requests.
  If there is deadlock it will be detected, but it is possible to detect deadlock when it doesn't exist.
- Timestamps can be used to avoid false deadlock detection.

Avoiding false cycle detection

- When process *A* at site *1*, requests a resource from process *B*, at site *2*, a request message with a timestamp is sent.
- The edge $A \rightarrow B$ with the timestamp is inserted in the local graph of *1*. The edge with the timestamp is inserted in the graph of *2* only if *2* has received the request message and cannot immediately grant the requested resource.
- The deadlock detection controller asks all sites for their wait-for graphs.
- For requests between sites, the edge is inserted in the global graph if and only if it appears in more than one local graph (with the same timestamp).

Distributed approach

- There is an extra node (P_{ex}) in each local wait-for graph.
- All local processes waiting on any external processes point to P_{ex}.
- Any local processes waited on by an external process are pointed to by P_{ex}.
- If a cycle with P_{ex} involved is found we have a possible deadlock and information is sent to the site waited on.
- If a deadlock is then found then it is handled.
- If another possible deadlock is found involving P_{ex} another message is sent to another site etc.
- Until either deadlock is detected or there is no cycle.

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

<u>Message Passing</u>

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.

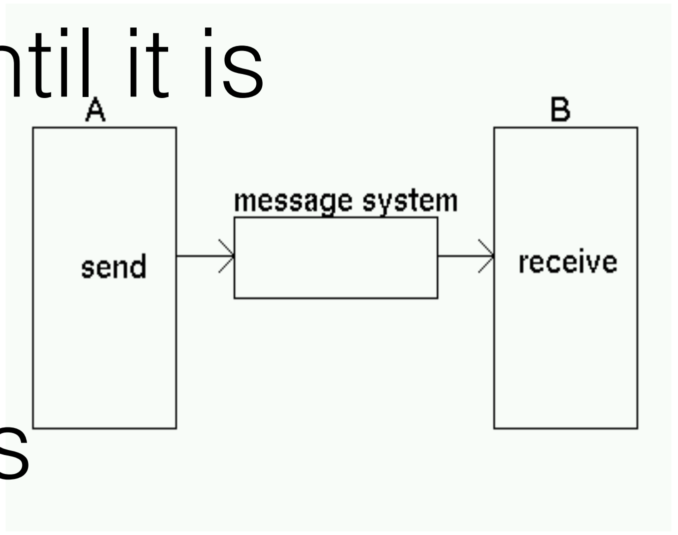
```
message = receive(type1,type2, ...)
```

- 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)

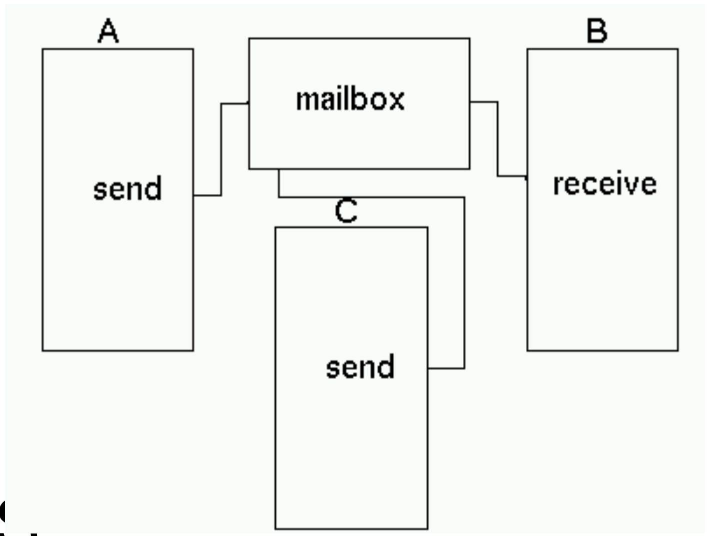
Direct communication

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
- <u>Disadvantages</u>
  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

UNIX process communication

Signals – software interrupts - text 4.6.2

`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

Unix 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);
```

- The system call 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 (65536 bytes on recent Linuxes)

Pipes (cont.)

Broken pipes

- A process waiting to read from a pipe with no writer gets an EOF (once all existing data has been read).
- A process writing to a pipe with no reader gets signalled.
- Writes are guaranteed to not be interleaved if they are smaller than the PIPE_BUF constant.
  This must be at least 512 bytes and is 4096 on Linux.

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).

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)

Before next time

- Read from the textbook

  3.8.1 Sockets

  3.5 IPC in Shared-Memory Systems

  3.7.1 POSIX Shared Memory