COMP2310 Introduction

Course co-ordinator: Richard Walker

E-mail: `richard@cs`
Room: N315
Phone: x53785

Text:

Jean Bacon, *Concurrent Systems*, second edition, Addison-Wesley, 1998.

Reference:

George Coulouris et al., *Distributed Systems: Concepts and Design*, second edition, Addison-Wesley, 1994. I recommend the fourth or fifth impression.

I will refer to other sources when necessary.

What are we doing today?

- What this course is about.
- What I expect you to know already.
- What is my job?
- What is your job?
- How will you be assessed?
- Some basic issues.
- What to do next.

What this course is about

How to understand, design, and build concurrent and distributed systems.

- concepts and abstractions
- what makes a good concurrent/distributed system
- how they can fail
- design choices
- language support for concurrent and distributed processing (C/Unix and Java)
- tools to help you to build concurrent and distributed systems
- how to diagnose problems and fix them

What I expect you to know already

(or learn by yourself)

- well-honed skills in Eiffel
- basic programming skills in C
- how to write well-documented programs
- software tools: `make` and `RCS`
- basic concepts of literate programming
- how to use Emacs
- how to write in coherent English

If the last point is a problem for you, *get help* from the Study Skills Centre.

What's my job?

What I must do for you:

- Help you to understand the concepts, issues, and skills in building concurrent and distributed systems.
- Make the course as relevant as possible. Use 'bleeding edge' technology!
- Make sure that we have a good time.

What I must do for the community:

- Assess and certify your competence in these areas.

What's your job?

- Be curious!
- Participate in the lectures, labs, assignments, and exams.
- Communicate: if you have a problem, question, or a suggestion, talk to me or your tutor as soon as possible.
- Plan how you will use your time.
  - Some people program many times faster than others.
  - Even slow programmers make good money.
  - But first you need to pass this course.
  - Know your own ability and plan accordingly.

Practical assessment

It is a course objective that you acquire certain practical skills.

- Writing skills.
- Programming skills.
- All programming will be literate.
- Two programming assignments (one done in pairs).
- One written assignment.
- Tutorial participation. Preparation and various skills.

Exam assessments

Exam assessment is needed to calibrate your achievement.

- Midsemester quiz (10%)
- Final exam (40%)

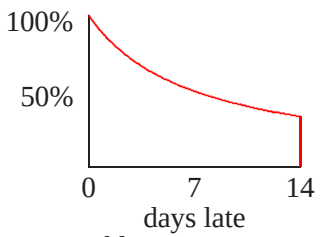
The exams will test your knowledge of basic facts, understanding of basic concepts, and ability to evaluate solutions.

What if I hand my assignment in late?

- You must hand in an acceptable solution within 2 weeks of the deadline to pass this course.

- Late assignments (less than 2 weeks) will be devalued as follows:

$$\text{finalMark} = \text{originalMark} \times \frac{1}{\frac{\text{daysLate}}{7} + 1}$$



- If you have a problem, come to see me *early*.

The assessment in detail . . .

- 50% exam component:
  - 10% redeemable mid-semester;
  - 40% final.
- 50% practical component:
  - 5% lab participation;
  - 15% C assignment;
  - 15% 'theory' assignment;
  - 15% Java assignment.
- Mid-semester compulsory (and you must get at least 40%).
- Concrete assignment specifications, a sample executable (for C), and lots of hints.

How your mark might be computed

```
public static int finalGrade(int theory, prac)
  {
    int worst = Math.min(theory, prac);
    int average = (theory + prac) / 2;
    int capped = Math.min(average, worst + 10);
    if (worst < 40) { /* Need 40% on both to pass. */
      return Math.min(capped, 44);
    } else {
      return capped;
    }
  }
```

Some basic issues

- How to declare concurrency.
- How to synchronize processors.
- How to communicate 'results'.
- How to prevent processing or communication errors.
- How to recover from them.
- How best to distribute the workload.

Systems we can study

- C/Unix and Java
- Message Passing Interface (MPI)
- Sockets and XTI
- Ada, occam, Linda
- RPC, RMI, CORBA, DCOM, . . .
- Anything else?

What to do next

- Register for a lab group.
- Get the reading brick ($5) before the next lecture.
- Get Bacon and read chapter 1.

Concurrency

- *Concurrent* - the word

  Dictionary meaning:

  At the same time.

  Is that clear?

  Each activity starts before the other finishes.

- Examples

  A person talking

  || a person listening

  Jack's program

  || Jill's program (on iwaki)

  What's the difference?

Concurrent Systems

- What are they?

  They handle concurrent activities

- Are they Hardware or Software?

  Either!

  Processor or process

- Coming up

  Inherently concurrent systems.

  Potentially concurrent systems.

Inherently Concurrent Systems (1)
Real-time systems

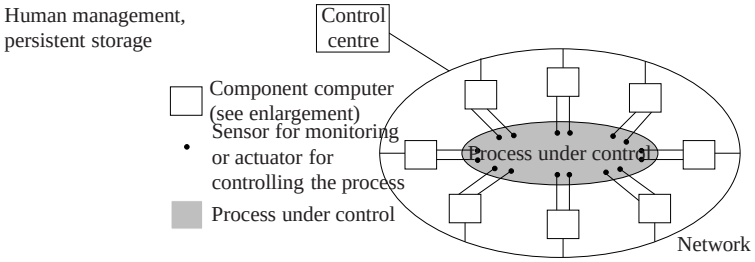
- Examples
  - *Chemical plants*
  - *Power Plants*
  - *Patient Monitoring systems*
  - *Vehicles*
  - *Robots*
  - *Telephone Network*
  - *Virtual reality*
- Significant Notions

  Hard real-time v. soft real-time

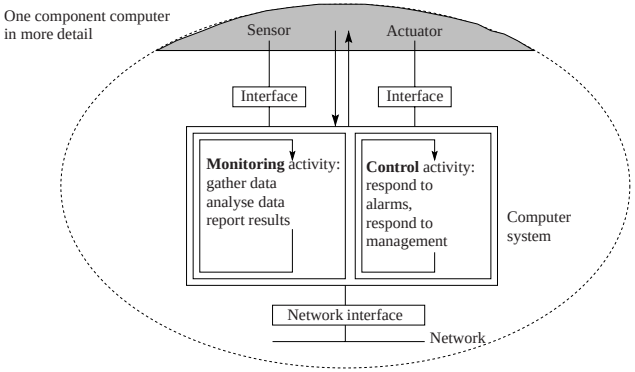
  Static and Dynamic systems

  Distribution

  Embedded systems

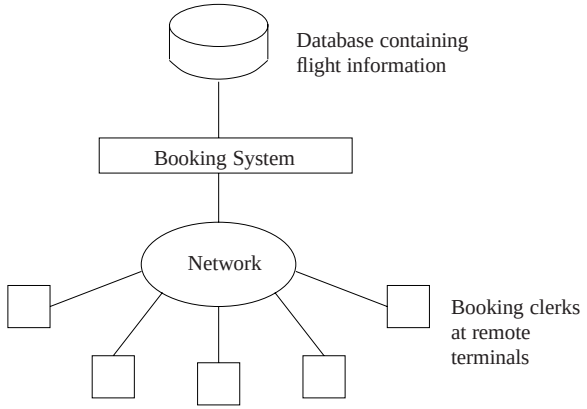


Example of a distributed process control system

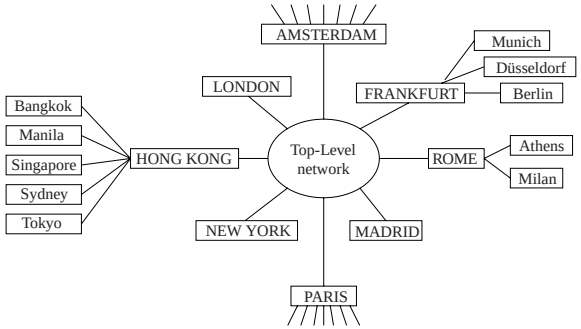


Inherently Concurrent Systems (2)
Database systems

- Examples
  - *Banking systems*
  - *Reservation systems*
  - *Inventory*
  - *Libraries*
  - *Personal records*
  - *CAD*
- Important Concepts
  - *Queries v. updates*
  - *Batch processing*
  - *Transaction processing*
  - *Distribution*



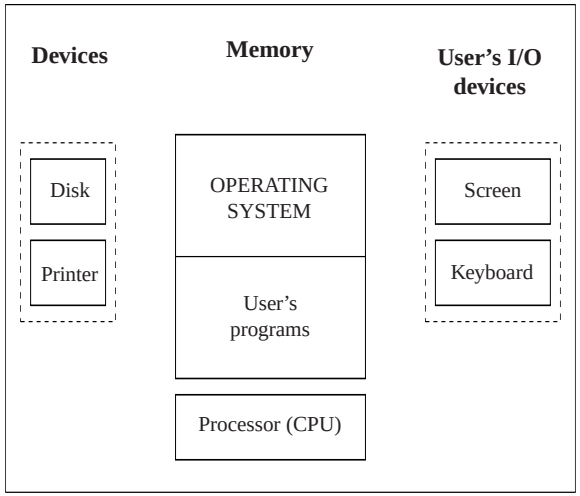
Components of an Airline Booking System



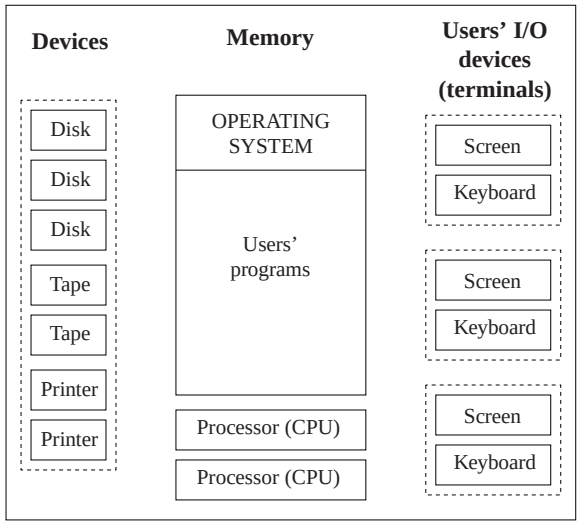
Part of the early SITA private network

Inherently Concurrent Systems (3)
Operating systems

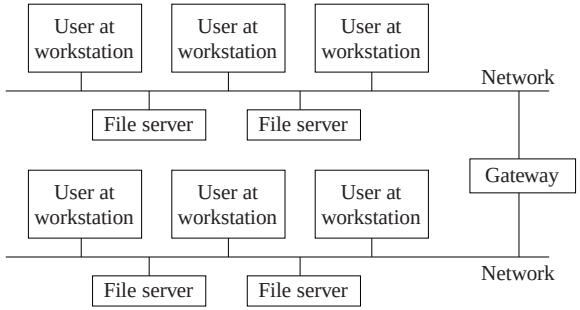
- Absolute basis:
  Peripherals are autonomous
- Where it started:
  Concept of concurrent processes was developed by OS designers
- Uniprocessor OS is concurrent
  At least conceptually
  Separating concerns
  Disparate activities are processes
  System level & user level
- Distributed OS
  Physical separation of activities



Components of a single-user system



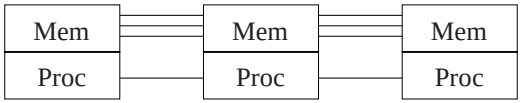
Components of a multi-user system



A simple distributed operating system

Potentially Concurrent Systems (1)
Parallel Machines (a)

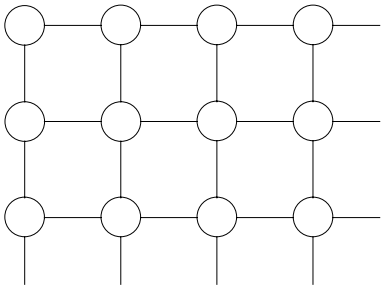
- Several or many processors



- Many possible architectures
  Usually homogenous
- **Application Areas**
  Physics, Chemistry
- **Grand Challenge Problems**
  Chemistry, Code Cracking, Astrophysics, Climate, ...

Parallel Machines (b)

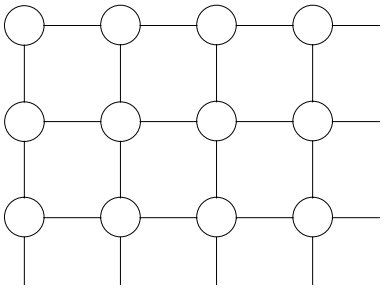
- **Local examples**
  - ANU/CSIRO
    Fujitsu,
    Thinking Machines,
    Maspar
  - DCS & Fujitsu
    AP1000 (CAP) [1990]



128 SPARC processors
16 MB memory each

Parallel Machines (c)

- **Current Local Machines**
  - AP1000+, AP3000 [1996]



12 SPARC processors
16 MB memory each

- **Significance**
  - Scalability experiments for visualisation, object stores and operating systems.

Parallel Machines (d)

Current Local Machines

- Bunyip [2000]
- http://tux.anu.edu.au/Projects/Bunyip/
- 'Beowulf' style
- 96 2-CPU 'commodity' PCs + 2 2-CPU front ends = 196 Pentium IIIs
- 384 MB memory in each box
- 13.6 GB hard disk in each box
- Very fancy networking (see web page) based on clusters

Architectural Jargon - Parallel Machines

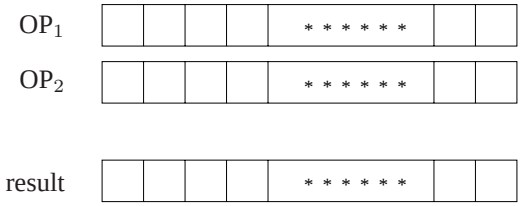
- **SISD / SIMD / MIMD**
  - A classification of architectures

S = Single	M = Multiple
I = Instruction	D = Data

- **SISD**
  - Single Instruction, Single Data
  - The Uniprocessor
- **MIMD** - Multicomputer
  - Loosely-coupled (network of workstations)
  - Tightly coupled (AP1000)

Vector Machines

- **SIMD** - Single Instruction, Multiple Data
  - Operations apply to vectors of data
  - generalisation of bitwise logical operations



  - One processor per element of vector
  - All processors in lockstep
  - Has conditional adds but no conditional jumps
  - ICL DAP (Distributed Array Proc.)
  - CM2 (Connection Machine)

Virtual SIMD

- Recall Notion of Virtual Machine
  - Any machine can be simulated
  - Remember "Virtual memory"
- Data parallel languages
  - All the primitive operations apply to vectors
    (Think of + on vectors)
  - Defined operations do too
- Examples
  - APL (1962)
    (A Programming Language)
  - SISAL (circa 1990)

Virtual MIMD

- Time-sharing
  - The case of X-terminals:
    1 processor, many processes
- General case
  - n processors and m processes (where m > n)
  - User may not be aware of how many CPUs are being used.

Parallel Machines Memory Organization

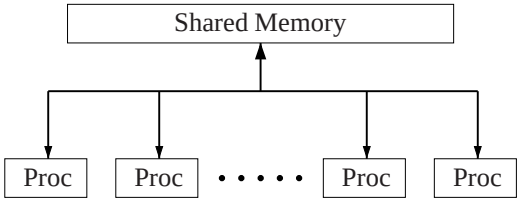
- **Shared memory**
  - Each processor has direct access to every word of one big memory (e.g. Sequent Symmetry, karajan)
- **Distributed memory**
  - Each processor has its own private memory
- **Virtual shared memory**
  - A layer of software can make a Distributed Memory Machine seem like it has shared memory

Memory Organization (1)
Shared Memory

- Logical View

Memory Organization (2)
Implementation

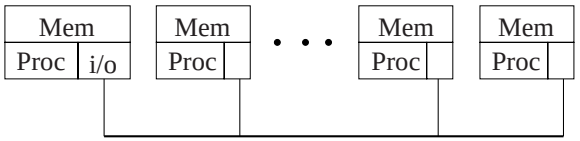
- Standard Design
  There'd be a bus.
  Can have just a few processors.



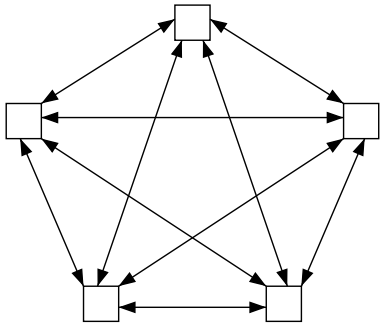
Hardware ensures that no two processors update one location at the same time.

Memory Organization (3)
Distributed Memory (a)

- Logical View

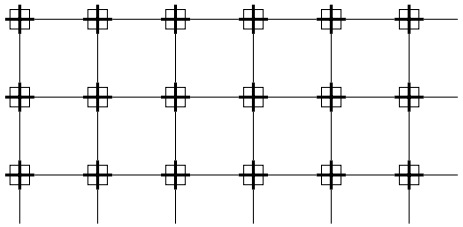


  - Programs must communicate via messages
  - Need I/O processors
  - Need a communication medium
  - It may be a bus
- Direct connections possible



Distributed Memory (b)

- The AP1000 again



  - Most pairs are *indirectly* connected!
  - Each cell has I/O processor
  - Messages go right then down
  - Concurrent messages are possible.
  - Toroidal structure
  - Scalable!

Memory Organization (4)
Virtual Shared Memory (a)

- The simulation of a single address space on a Distributed Memory machine.
- The mechanism
  - Each address can be split, say

6	26

  - Top 6 bits could give the cell number
  - or they could be a page number (where page could be anywhere).

Virtual Shared Memory (b)
Issues

- Hardware support
  - Must have specialised communications circuits
- Local versus non-local (NUMA)
  - Local references are faster
  - Code in local memory
- Locality of Programs/Data
  - Make variables local!
- Programming languages
  - Consequences in design & especially in implementation

Memory Organization (5)
Memory Caches (a)

- CPUs faster than main memories
  - even faster than shared main memories
  - much faster than nonlocal memories
- Examples
  - Contrast

    m = n∗n;

    on a 1-address machine
  - with

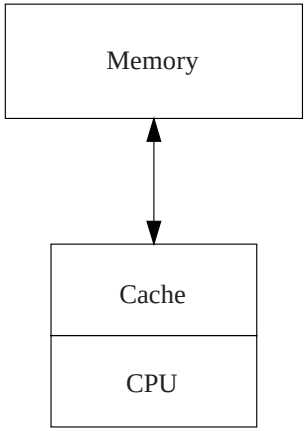
    x = a + b; w = x + 1;

    y = a∗x∗w + b∗x + w;

    on a register machine

Memory Caches (b)

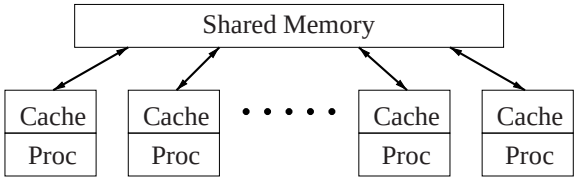
- Single processor cache
  - Generalize from register machine



  - cache is associative memory

Memory Caches (c)

- Multi-processor case



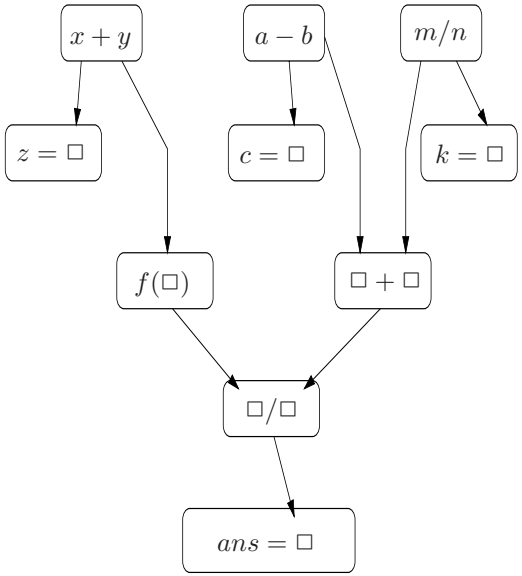
  - Cache coherence problem

Data-Flow Architectures

- Sequencing constrains computation unnecessarily

```
z = x+y;
c = a-b;
k = m/n;
ans = f(z)/(c+k);
```

- Can use dependencies to control computation



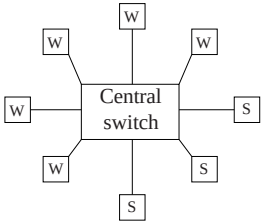
Functional Language Machines

- Miranda, ML, Lisp
- No side effects

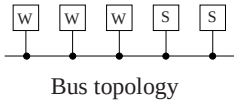
```
f( g(e1, e2), h(e3, e4))
```

  - maybe 7 computations here to do in parallel

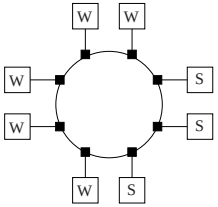
- Eager v. Lazy evaluation
  - Each can save time
  - Each can waste time



Star topology



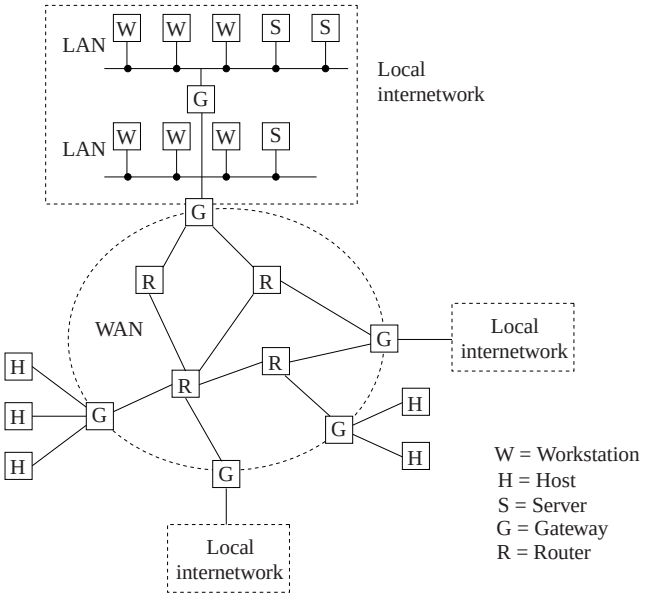
Bus topology



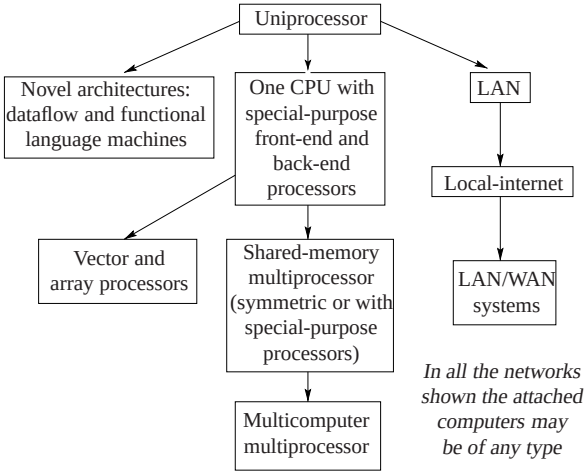
Ring topology

W = Workstation
S = Server
■ = Repeater

Star, Bus and ring LAN topologies



LANs connected by WANs



Summary of concurrent system topologies

Arrows take you in a direction of increasing concurrency

Fundamental Features of Concurrent Systems

- **Support for Tasks**
  - Real-time – obvious
  - OS – users, devices
  - Database systems – transactions
  - Supercomputers
    - processors have processes
- **Support for Task Management**
  - Create, run, kill, suspend, nice
- **Support for Co-operation**
  - Sharing of resources
  - Messages
- **Support for Protection**
  - Support virtual machines
- **Support for Deadlines**
  - Alarms
  - Simulation

References

(Overview of Concurrent Systems)

- Required Reading
  - Bacon, Chapter 1
- Supplementary Reading
  - Chapter 1 of any OS text

Introduction to Processes (1)

- Definition – Per Brinch Hansen
  A **process** is a computation in which the operations are carried out strictly one at a time; concurrent processes overlap in time. They are **disjoint** if each refers to private data only and **interacting** if they refer to common data.
  (A **computation** is a finite sequence of operations applied to a finite collection of data.)

Introduction to Processes (2)

- process $\neq$ program
  The **process** relates to the execution of operations. It's about what happens on some occasion.
  The **program** is the recipe for a computation. It is text that is followed in a process.

Introduction to Processes (3)

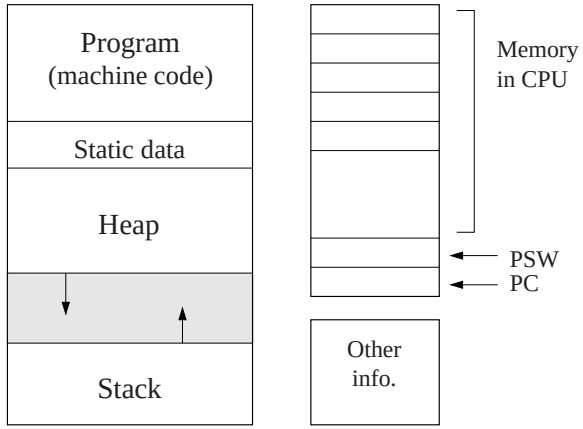
- Examples
  - Programs and processes in a Unix system.
  - Tasks on an assembly line.
  - Musicians – Distinguish the score and each performance.
  - The shared family book. (Jean Bacon)

UNIX

- Origins - The Multics Project
  - MIT and GE
  - Ken Thompson, Dennis Ritchie
  - The C language
  - PDP 7, then PDP 11
- Features
  - Hierarchical file system
  - Compatible I/O (file, device & inter-process)
  - Process creation
  - Command language choice
  - Portability
- Descendants
  - Version 7, BSD 4.2, 4.3, 4.4
  - System V, Solaris
  - XINU, Minix
  - Linux
  - Mach (e.g. Darwin, Mac OS X)

UNIX Processes (1)
What are they?

- The Unix Virtual Machine?
  - The state of a uni-process virtual machine
  - Private memory, code, registers, program counter



UNIX Processes (2)
Reproduction

- In UNIX, process creation is by cloning!
  In the standard Unix C library there is a function *fork*.
- Typical usage: pid = fork();
- *fork* spawns a duplicate
  - same memory, regs, PC
- *fork* returns:
  - 0 to child;
  - and to parent:
    PID (process id) if it works
    -1 if it fails
    never 0

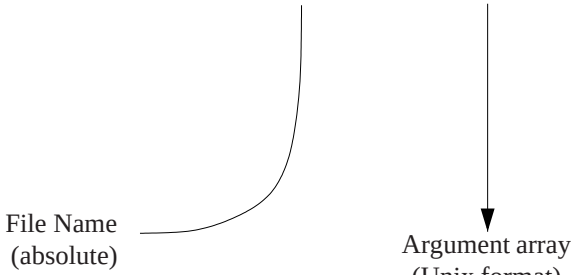
UNIX Processes (3)
A Change of Mind

- Maybe another program is more suitable?
  - Passing responsibility
  - Many computations have quite different phases

- A child process may need to do a very different job

- MODULE UxProcesses - again

public native int Exec1(String prog, String argt);



- Similar functions in C

UNIX Processes (4)
Semantics of *exec*

- Call of typical family member

```
Exec1(filnam, args);
```

- does the following:
  - Memory is replaced by the image given in the executable file
  - Registers initialized in the usual way for a program
  - Parameters are passed as if the new program was called from the shell
  - Only returns in case of a failure

UNIX Processes (5)
More about exec

- Typical Usage:

```
if (Fork() == 0) {
    Execp("/usr/bin/grep",
          Arg2("grep", keyword));
    Abort();
} else {
    <rest of PARENT program>
}
```

- Questions:
  - What is the child?
  - What is concurrent?
  - Why does grep appear twice?
  - Why might it abort?

UNIX Processes (6)
Requiem

- Co-ordination can be a problem
  - A child was spawned to do something in parallel
  - At some point the parent must be sure it is complete

- The Wait primitive solves it

```
public native int Wait();
     (* from Class  UxProcesses *)
```

  - Waits for termination of a child - any child
  - Returns PID of the child that has died

UNIX Processes (7)
Example of wait

- Pattern of Usual Usage

```
if (Fork() == 0) {
    <Child's task>
    UNIXexit(0);
} else {
    <Parent's task>
    Wait(Codes);
}
```

- Concurrency?

UNIX Processes (8)
Dozing

- Why wait for time to pass?
  - It may want to do something periodically
  - It can make sure other processes get a chance

- The Sleep primitive solves it

```
public native int Sleep(int secs);
  (* from Class UxProcesses *)
```

  - The supplied argument is the duration (in seconds)
  - May return early if woken up (by signal)
  - Result indicates sleep-time remaining
  - Use of sleep(0)

UNIX Shells (1)

- Job control language
  - Pre-Unix JCLs were proprietary, idiosyncratic

- Not *part* of *Operating System*
  - But must reflect its facilities
  - Can have a shell per user.

- There are many common ones
  - sh - Bourne shell
  - csh - C-shell
  - ksh - Korn shell
  - tcsh - Extended C-shell

– bash - Bourne-again shell

Though these are all similar!

UNIX Shells (2)

- Features of the shell
  - A virtual machine
  - Has a language
    - ∗ can invoke programs
    - ∗ types, variables, control
  - Can write programs
- Access to Unix's hierarchical file system supported by *cd*, *pwd*, *ls* etc.
- Uniform I/O is supported by re-direction of *stdin* & *stdout*
  - `ls -als >DIR`
  - `ls -als >/dev/mt0`
  - `ls -als | grep def`

UNIX Shells (3)

- Processes supported by shells
  - `ps`
  - `emacs George.java &`
  - `ps -axu | grep mcn`
- Can write executable programs in shell code
  - Usual terminology - *scripts*
- Interpreted, not compiled
- First line of file indicates the appropriate interpreter.

C-Shell Scripts An Example

- Example of shell script

```
#! /bin/csh
set filist = ‘cat $1‘
foreach nam ( $filist )
  if (-e $nam) then
    cp $nam $2/$nam.bak
  else
    echo $nam "doesn’t exist."
  endif
end
```

- What does it do?

  Suppose script is in file `~/bin/save`

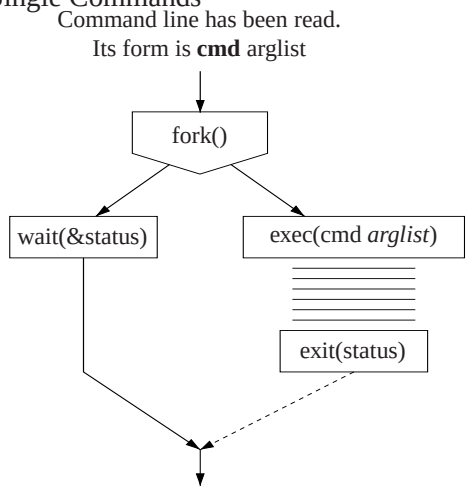
  Then `save workfiles ../backup` will copy each file mentioned in the file workfiles to the directory called backup with a .bak extension.
- Notes

  `$1` and `$2` refer to the arguments;

  $nam gives the value of variable nam

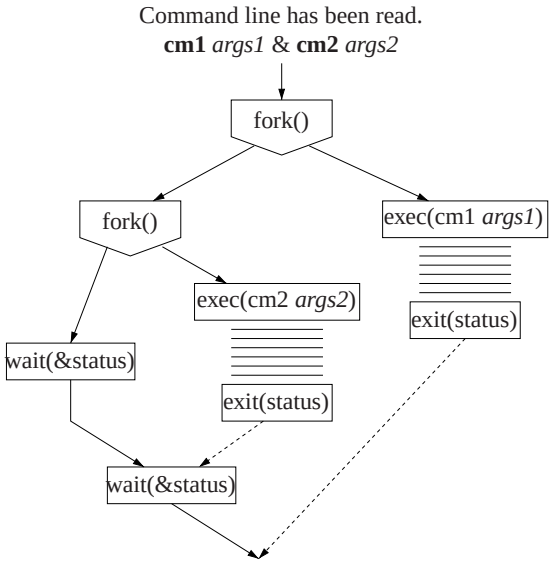
Command-Line Interpretation (1)

- Single Commands

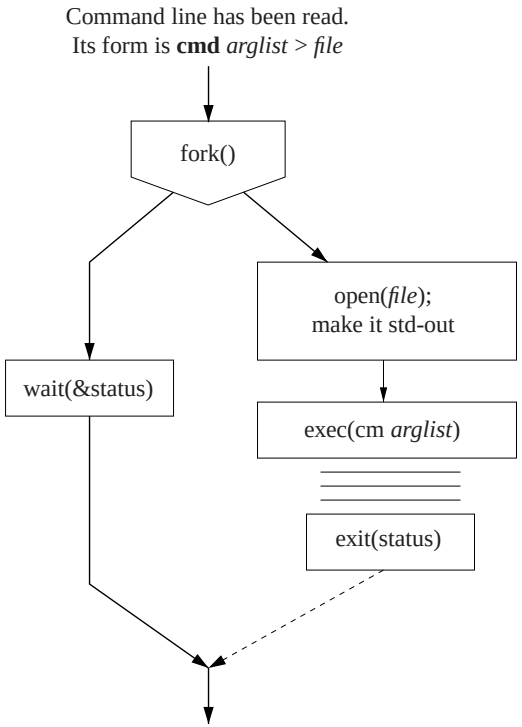


- This is like the top half of Fig 23.16 in Bacon.

Command-Line Interpretation (2)

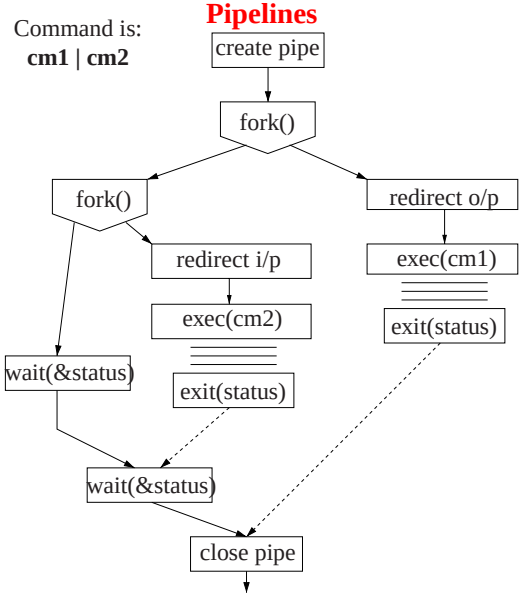


Cmd-Line Interpretation
I/O Redirection



- Justifies the fork primitive.

Cmd-Line Interpretation
Pipelines



References

- "man pages" on a Unix system
  Quite technical and long but authoritative.
  You need to look at
  - csh, sh.
  - fork, exec, exit, wait.
- Books on Unix
  Most books with Unix in title only address the shell & applications, not system prog.
  See books by Bourne & by Wang.
  Read about:
  - fork, exec, exit, wait.
  - pipe (covered soon)

The ADT, Semaphore

- **Information Content**
  The state of a semaphore can be described as:
  a counter and
  a queue of processes.
- **The big idea**
  Semaphores provide a way of suspending a process until it is safe for it to continue.
- **For example**
  A semaphore can be used to achieve mutual exclusion.

The Semaphore Operations

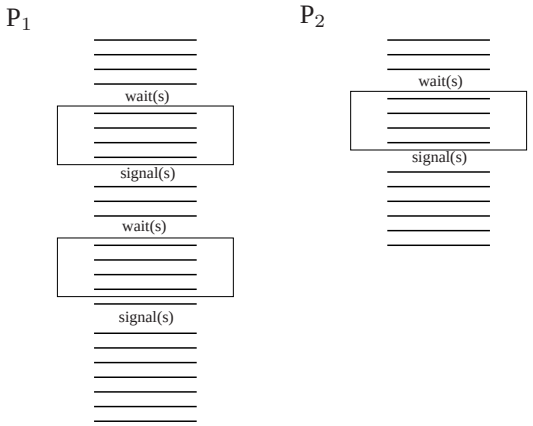
- **Initialisation**
  `init(S,J)`
  Initialises semaphore S;
  (queue empty, counter val = J).
- **Suspension**
  `wait(S)`
  After decrementing counter,
  if it is negative, suspend current process and add it to queue of S ;
- **Activation**
  `signal(S)`
  Activate 1st process in queue
  (if any); increment counter of S

Using Semaphores

- **Mutual Exclusion**
  Making sure two processes don't execute critical code (of the same class) at the same time.
- **Synchronization**
  Making sure some activity of one process is complete before some related activity of another process.
- **Resource control**
  Managing a homogeneous pool of resources of some type. (It must not matter to any process which of the items in the pool is allocated to it.)

Semaphores for Mutual Exclusion

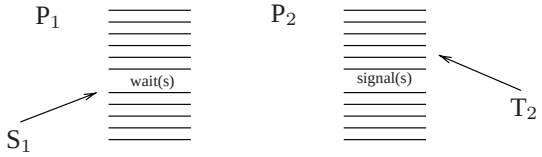
- Initialise Counter to 1 (Queue initially empty)
- Entry protocol is *wait* (s)
  Exit protocol is *signal* (s)



Using Semaphores for Synchronization

Suppose process P_1 cannot execute code S_1 until process P_2 has finished task T_2.

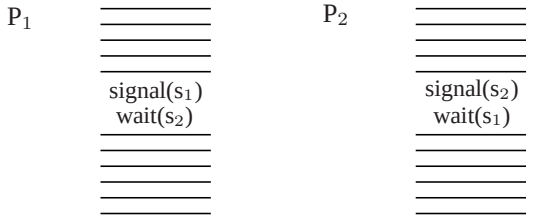
- Initialise semaphore counter to 0. (Queue initially empty)
- Before S_1 in P_1 place a `wait(s)`
  After T_2 in P_2 put a `signal(s)`



Using Semaphores for Rendezvous

Suppose neither process P_1 nor process P_2 can proceed until each has reached a certain point in their computations.

- Use two semaphores, S_1 & S_2
- Initialise semaphore counters to 0. (Queues initially empty)



Control of Resources

Where there are multiple instances of a resource, processes should be able to claim some,
. . . but only as many as exist.

- **Examples**
  - Mag tape units
  - Communications channels
  - Processors

  - Network access to software
  - Space
- **The Solution**
  - Use a semaphore.
  - If there are *n* units, initialise semaphore to *n*
  - Use *wait* to allocate, and *signal* to deallocate.
  - Mutual exclusion is the case where n=1.

Locks v. Semaphores

- **Both can delimit critical code**
- **Operating System Dependencies**
  - Semaphores implemented using OS process manager.
  - Locks are independent of OS. Locks involve busy-waiting and special instructions.
- **Efficiency Issues**
  - OS call is constant overhead (semaphores)
  - Busy-waiting is variable cost (locks).

Implementation of Semaphores (1)

- **Considerations**
  - It is implemented in OS
  - Primitive available for the unconditional suspension of processes - *Block*
  - Can inhibit interrupts
  - Must consider multiple processors
- **Declarations**

```
public class Semaphore
{
   private int counter;
   private ProcessQueue waiters;
   private boolean mutex;

   public Semaphore() {
     <initialization (Queue etc)>
   }
```

Implementation (2)

- **Code for *wait***

```
   public void wait() {
     private Process p;

     InhibitInterrupts();
     lock(mutex);
     counter = counter - 1;

     if (counter < 0) {
       p = <process doing wait>;
       enter(p, waiters);
       block(p); // block,
             // enable interrupts
     }

     unlock(mutex);
     EnableInterrupts();
   }
```

Implementation (3)

- **Code for *signal***

```
   public void signal() {
     <code for signal>
   }
```

... is similar!

- **Overkill?**
  Why protect the critical code in TWO distinct ways?

Why Inhibit Interrupts?

- **Achieve Mutual Exclusion**
  - It locks out other processes on the same processor.
  - There is no time-slicing when interrupts disabled.
- **Is there a problem?**
  - It only excludes processes on the same processor.
  - Another processor doing a signal or a wait will execute this critical code.

Why Have a Lock?

- **Achieve Mutual Exclusion**
  - It locks out other processes on other processors.
- **Is there a problem?**
  - It may not exclude other processes on the same processor.
  - Another process may start (or resume) in response to an interrupt. This 2nd process will sit and 'spin' *for ever*.

Producer-Consumer Problem (1)

- **A Problem?**

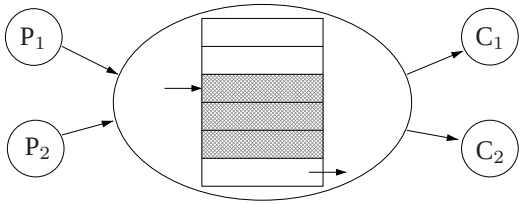
  Not really; more a solution.
  even an ADT.
  Many systems have this pattern.

Must know.

- **Also Called Bounded Buffer Problem**
  It applies when there is a need to smooth out bursts of activity.
  Not about delivering fixed size blocks of characters to devices.

Producer-Consumer (2)

- **Template Description**
  Some processes produce items of some given size. (Order does not matter.)
  Some process consumes these items asynchronously.

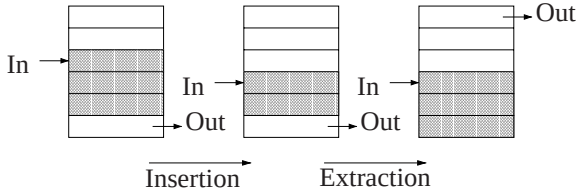


Cyclic Buffers

- **Data Structures**

  BUF[0..N-1] - The buffer
  Count - Buffer occupancy
  In - The index for insertion
  Out - The index for extraction

- **The movie**



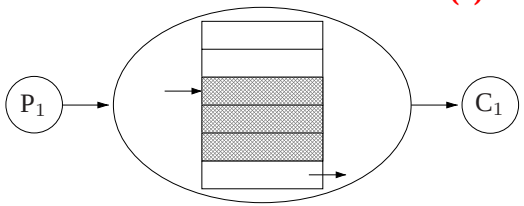
  - **Insertion operation** (no concurrency)

```
BUF[In] = Item;
In = (In+1) % N;
Count = Count + 1;
```

  - **The extraction operation** (also atomic)

```
Item2 = BUF[Out];
Out = (Out+1) % N;
Count = Count - 1;
```

One Producer & One Consumer (1)



- **Data Structures**

  BUF[0..N-1], In, Out as before.
  Items - A semaphore initialised to 0.
  Spaces - A semaphore initialised to N.

One Producer & One Consumer (2)

The METHODS

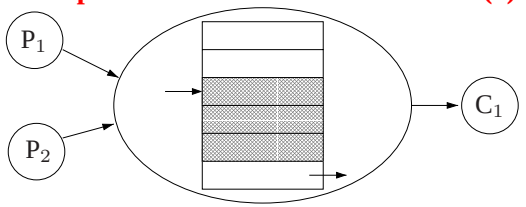
- **The Insertion Operation**

```
Spaces.wait();
BUF[In] = Item;
In = (In+1) % N;
Items.signal();
```

- **The Extraction Operation**

```
Items.wait();
Item2 = BUF[Out];
Out = (Out+1) % N;
Spaces.signal();
```

Multiple Producers & One Consumer (1)



- **Data Structures**

  BUF[0..N-1], In, Out as before.
  Items - A semaphore initialised to 0.
  Spaces - A semaphore initialised to N.
  Guard - A semaphore initialised to 1.

Multiple Producers & One Consumer (2)

The METHODS

- **The Insertion Operation**

```
Spaces.wait();
Guard.wait();
BUF[In] = Item;
In = (In+1) % N;
Guard.signal();
Items.signal();
```

- **The Extraction Operation**

```
Items.wait();
Item2 = BUF[Out];
Out = (Out+1) % N;
Spaces.signal();
```

Readers & Writers

- **Another 'situation template'**
  - Shared manipulable resource
  - Typically a data base

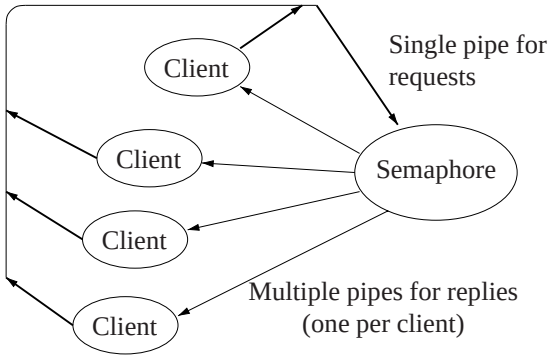
- If not ‘being updated’ can be shared (multiple readers)
- If being updated one process has exclusive access
- Writers have priority
- Think file update

Readers & Writers Implementation

- **Slide 10.11 in Bacon.**
- **Slides 10.8 - 10.10 in Bacon.**

Implementing Semaphores Using Messages (1)

- **Assume Unix processes, pipes**
  - Semaphore process used
  - Signal and wait done by messages
  - Several processes can share a pipe

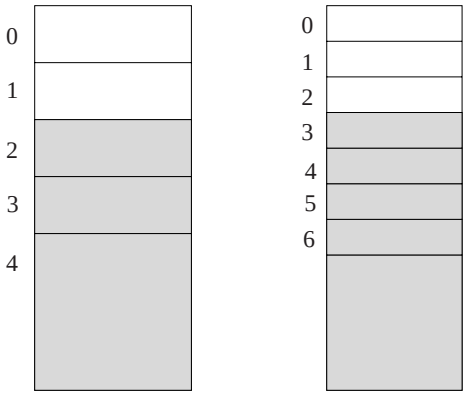


Implementing Semaphores Using Messages (2)

- **Protocols**
  - Wait - client sends W_α (where α is client id)
    Sem process sends reply which client reads.
  - Signal - client sends S no reply needed
  - Sem process maintains count; has list of processes awaiting reply.
  - Importance of pipe flushing.
- **Non Unix systems**
  - Messages are usually higher level
  - Replies may be mandatory

UNIX File Descriptors

- What are they?
  - Non-negative integers denoting open I/O channel
  - They index the current process's File Descriptor Table
  - Associated with each process is an FDT



  - The indexes (0, 1, 2, . . . etc.) are file descriptors.
  - The corresponding table entries mean something to the operating system.

File Descriptors (ctd)

- Three channels are standard

stdin	file descriptor 0
stdout	file descriptor 1
stderr	file descriptor 2

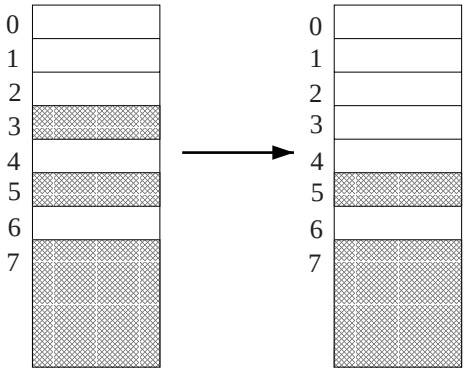
- Initially,

stdin	keyboard
stdout	current xterm window
stderr	current xterm window

- FDT preserved across *exec*
- FDT copied during *fork*

UNIX File Primitives

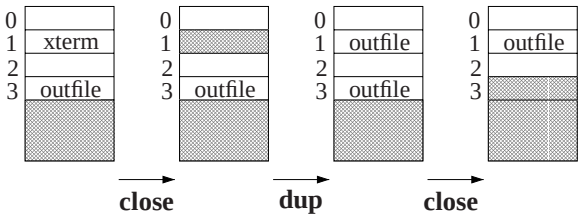
- Opening a file



  - First free entry in FDT chosen
- Closing a file
  - Entry in FDT is marked free
  - Opposite of open (above)

Redirecting I/O

- Redirecting output, say
  - User cannot change FDT directly
  - There is a primitive *dup* for this purpose

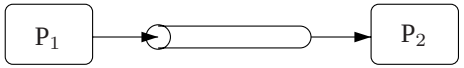


UNIX Pipes (1)
Basics

- What we already know:
  In the command

    ps -axu | grep 666

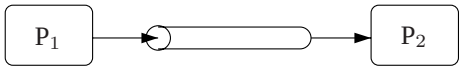
  - ps writes to stdout
  - grep reads from stdin
  - I/O has been redirected by the shell
  - Shell doesn't handle the characters
- Mental model



  - It's a buffer for characters
  - Handled by I/O sub-system of Unix
  - NB one-way, (but cf. Solaris (two-way))

UNIX Pipes (2)
Dependencies

- Synchronization



  - P_2 will wait for production
  - P_1 will not wait for consumption
  - There may also be buffering in P_1
  - The importance of timely flushing

Running a Shell as a Subprocess

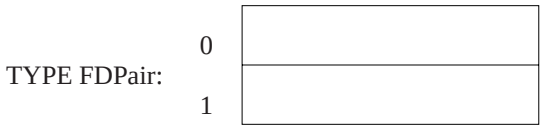
- Can have a shell to do a job
  - Unix allows easy creation of subprocesses
  - A subprocess can run a shell script

- An easy-to-code method
  inputStream Csh (cmd)
  - cmd is shell code
  - A process is created to run cmd
  - Result is an input stream on which output of cmd is delivered
- Task 2 of Lab 2 is like this

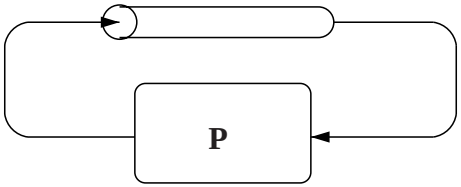
UNIX Pipes (3)
Creating pipes

- Opening a pipe

public void Pipe (FDPair pipeFDs)



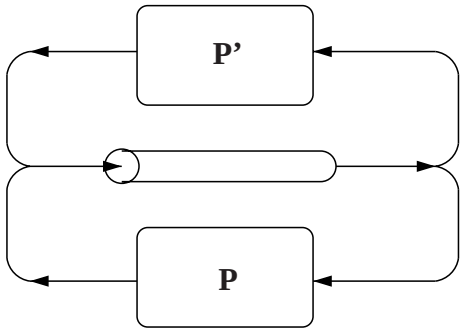
  - pipeFDs[0] set to file descriptor of input end of pipe
  - pipeFDs[1] set to file descriptor of output end of pipe
  - This is the result:



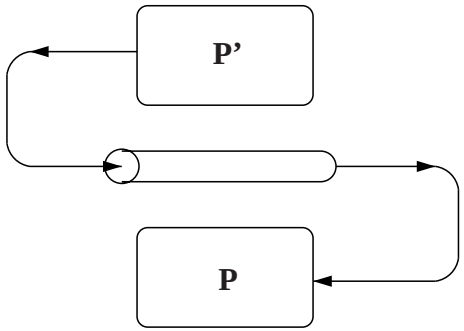
  - This does not connect two processes.

Connecting Two Processes by a Pipe

- Fork the process
  Do it AFTER opening a pipe

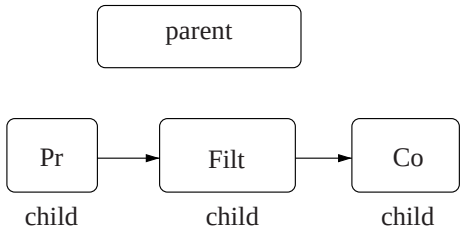


- Close unwanted descriptors



UNIX Pipes (4)
Subprocess Pipelines

- Desired state:



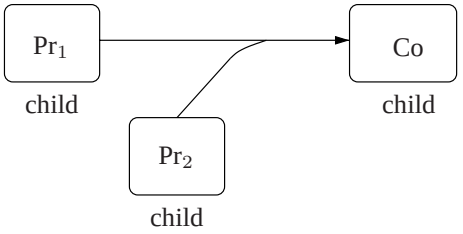
- Make pipes (before forking);
- Fork three times;
- Close pipes in parent

Each child closes descriptors that it won't use.

- Each child does an exec if appropriate

UNIX Pipes (5)
Sharing Pipes

- Pipes can be shared



- Integrity of messages
  - Pipe is just a stream of chars.
  - No interleaving BUT care required with buffering.
  - Messages less than 2K

References

- **man Pages**
  - man page for *pipe*
  - pages for *read*, *write*, *close*
  - page for *select*
- **Books**
  - S. R. Bourne,
    "The UNIX System"
  - Paul Wang,
    "An Introduction to Berkeley UNIX"

Processes v. Threads

- **Heavyweight Processes**
  - Each process has its own memory space
  - Co-operating processes can run on separate computers
  - Often separately compiled
  - Maybe in different languages
  - Typified by Unix processes
- **Threads**
  (a.k.a. Lightweight Processes)
  - They share (most of) their virtual memory
  - They have the *same* global variables and heap
  - But have separate stacks, different set of registers
  - Part of the 'same program'

Language Support for Co-operating Processes

- **What is supported?**
  - Mutual exclusion:
    Some languages have special syntax for critical sections
  - Synchronization:
    Primitives for signalling.
- **Advantages**
  - More reliable code
  - More readable code
  - Compiler checks on abuse of critical sections
- **Text Reference**
  - Bacon - Chapter 11
    Language primitives for shared memory

Special Syntax for Critical Regions

- **Shared variables identified**
  - Attribute *shared* allowed for any data item
  - `var A: shared array[1:n] of CHAR`
- **Critical Sections**
  - Each critical section is tagged with its *shared* variable
  - ```
    region S1
    begin
        S1.stk[S1.ptr] := e;
        S1.ptr := S1.ptr+1
    end;
    ```
- **The advantage**
  - The compiler can check that each occurrence of a *shared* variable appears inside such a region.

Limitations of Critical Region Syntax

- **The report card**
  - For simple mutual exclusion problems it is great.
  - . . . but in many cases it just doesn't help.
  - So it gets an F.
- **The problem area**
  - Sometimes a process should only do the critical section if some predicate holds . . .
  - and the computation of the predicate is critical code.
  - The process should wait for co-operation to make it true.
- **Example**
  - Bounded buffer situation.

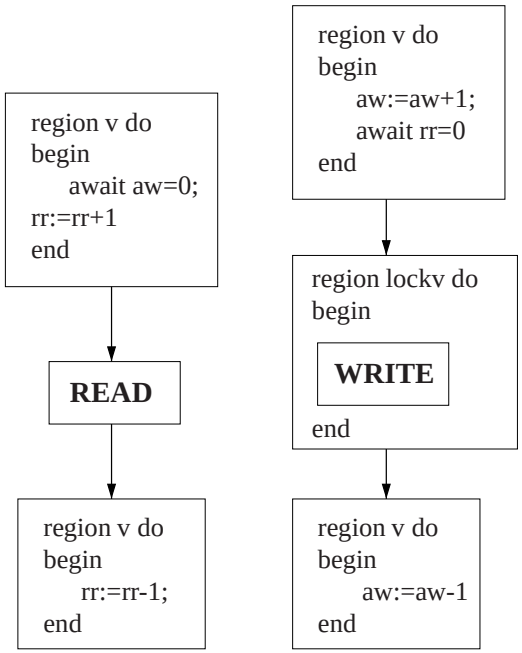
Conditional Critical Regions

- **Allow temporary escape** from a critical section
  - Sometimes critical code must wait until conditions are right;
  - e.g. wait for queue not empty or buffer not full.
  - Not always at start of section.
- **Brinch Hansen's *await* primitive**
  - The first instance of CCR (1973)
  - Syntax:
    `await (<bool exprn>)`
  - Suspends the current process if condition is false **and** allows another process into critical section.

Conditional Critical Regions Example

Readers and Writers:

This is Bacon Fig. 11.1
    ```

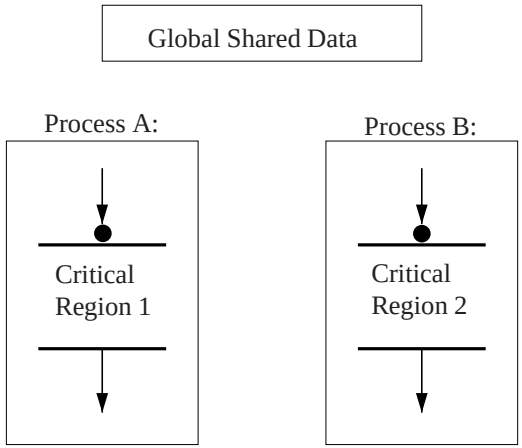


Conditional Critical Regions Conclusions

- **Compare** last slide with the solution using semaphores (9.28)
- **Very natural, very general**
- **Implementation problems**
  - Would have to check all conditions on every related assignment statement.
- **Condition variables**
  are less general but workable.

Monitors (1)
Motivation

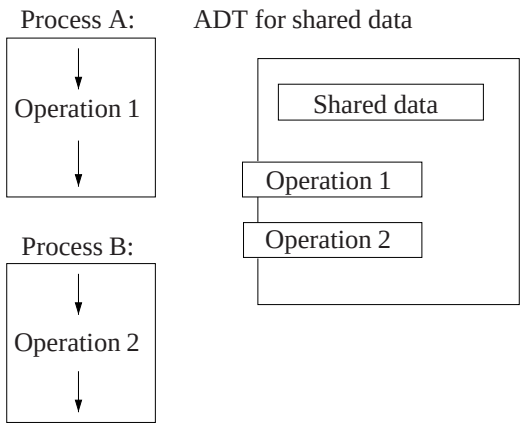
- **Shared variables and ADTs**
  - Each critical section is an operation on shared resource



  - Why not group them modularly?

Monitors (2)
Motivation (ctd)

– Let us group them modularly.



  - When together, regions can be implicit
  - This is the basic idea on which the concept of monitor rests.

Monitors (3)
Description

- **Definition**
  A *monitor* is an ADT with the property that no two processes can be *actively* computing in that ADT at the same time.
  - Each operation is a critical section; the controlled resource is the ADT.
- ***"actively* computing" ??**
  - While one process is actively computing, others may be queued waiting to start, . . .
  - or queued waiting for a signal, having temporarily escaped from critical code.

Monitors (4)
The Primitives

- **Condition variables**
  - The means of suspension is the *wait* primitive; the means of arousal is a *signal* from another process.
  - Syntax:
    `WAIT(<condition-var-name>)`
    `SIGNAL(<condition-var-name>)`
- **PAY ATTENTION TO THIS!**
  - DON'T confuse Monitor *signal* and *wait* with Semaphore *signal* and *wait*.

Monitors (5)

Pattern of Usage

- **Use of condition variables**
  Reference: This is Bacon Fig. 11.4

Monitor procedure

> • Ensure mutual exclusion;
>
> **if** not able to proceed **then**
>
> • WAIT(cond-var);
>
> Operate on shared data;
>
> Release mutual exclusion

- Point of potential delay
  - Note: the WAIT releases the exclusion on the monitor

Example 1
Single Resource Allocator

- Equivalent to a semaphore initialised to 1

```
private boolean busy = false;
private condition free;

public void reserve()
{
    if (busy) {
      WAIT(free);
    }
    busy = true;
}

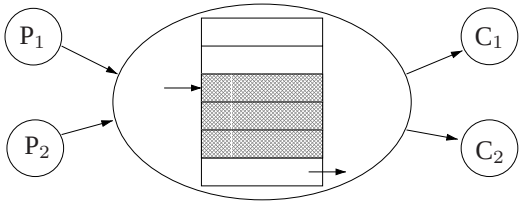
public void release()
{
    busy = false;
    SIGNAL(free);
}
```

```
}
```

- Exercise: do the same for multiple resources

Example 2
Bounded Buffer

- **Reminder**



- **The operations**
  - **Insert:** Put an object into the buffer, except if it is full.
  - **Remove:** Get the next object from the buffer, except if the buffer is empty

Example 2
Bounded Buffer (continued)

- **The Code for the Monitor**

```
public void Insert(item)
{
    if (<buffer full>) {
        notfull.wait();
    }
    <put item in buffer>;
    notempty.signal();
}

public void Remove(item)
{
    if (<buffer empty>) {
        notempty.wait();
    }
    <get item from buffer>;
    notfull.signal();
}
```

Example 3
Readers & Writers

- **Declarations**

```
private int rr,aw;
private boolean busy_writing;
private condition can_read, can_write;
```

- **Readers**

```
public void start_read()
{
   if (aw > 0) {
      can_read.wait();
   }
   rr = rr + 1;
   can_read.signal();
}

public void end_read()
{
   rr = rr - 1;
   if (rr == 0) {
      can_write.signal();
   }
}
```

Example 3
(ctd)

- **Writers**

```
public void start_write()
{
   aw = aw + 1;
   if (busy_writing || (rr > 0)) {
      can_write.wait();
```

```
    }
    busy_writing = true;
}

public void end_write()
{
    aw = aw - 1;
    busy_writing = false;
    if (aw > 0) {
        can_write.signal();
    } else {
        can_read.signal();
    }
}
```

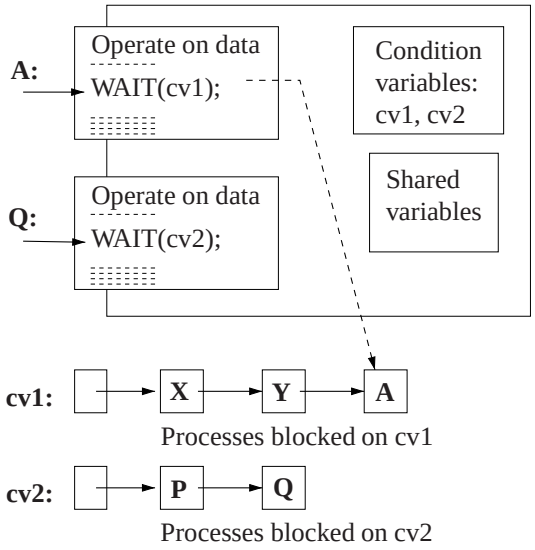
- **Reference**
  Bacon 11.2.3

Implementation of Monitor *wait* (1)

- **What happens in a *wait?***
  - The process is suspended unconditionally.
  - The process is put on a condition variable queue.
  - It is regarded as not being in the critical code. Another process waiting for entry is allowed access.

Implementation of Monitor *wait* (2)

- **Bacon Fig. 11.5**, in which we see process A do a WAIT.

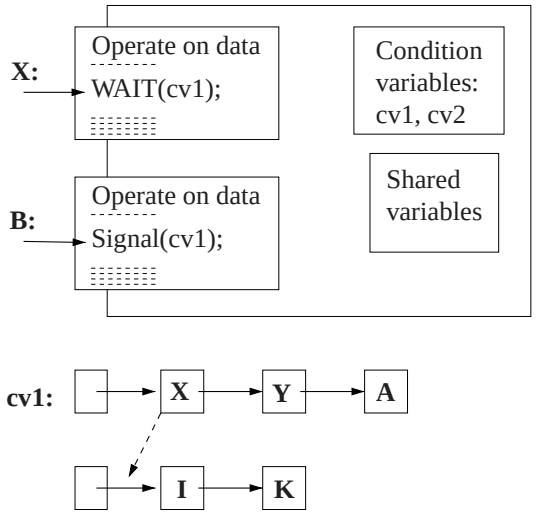


Implementation of Monitor *signal* (1)

- **What happens at a *signal?***
  - If the queue associated with the condition variable is empty then nothing!
  - Otherwise, the first process on the queue is made ready-to-run.

Implementation of Monitor *signal* (2)

- **Bacon Fig. 11.6**, in which **B** does a signal and activates **X**.



Monitors - Discussion

- **Mutual Exclusion**
  Effective but a blunt tool; every procedure provides mutual exclusion for ALL data in the module.
- **Mesa solution**
  In Mesa mutual exclusion done on a ‘per procedure basis’.
- **Synchronization**
  Some other constructs are more efficient.

Comparison of Sync. & Mutex. Primitives

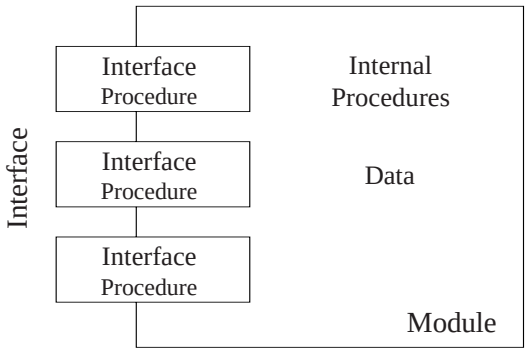
- **Three are Very Important**
  - Locks, Semaphores & Monitors

	Locks	Semaphores	Monitors
Level	Low	Low	High
Environ-ment	Machine level	Operating System	High Level Language
Cost	Busy-waiting	Constant Overhead	Constant Overhead

- **You may never see:**
  - Conditional Critical Regions
- **There are others!**
  - Event Counters

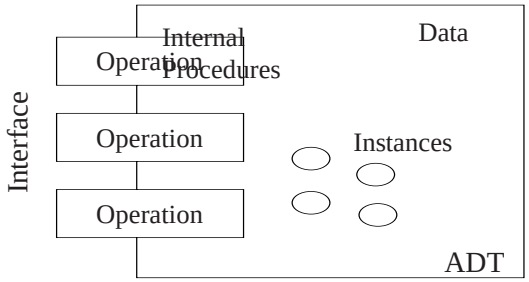
Modules

- **Essence**
  - Separate compilation
  - Information hiding
- **Advantages**
  - Complexity control
  - Reusability
  - Team programming
  - Maintainability
- **Bacon's Diagrams**



ADTs and Objects

- **Essence**
  - Data encapsulation.
  - Semantics captured in axioms.
  - Enable separation of the *Implementation* from the *Specification*.
  - ADTs exemplified in Modula-2.
  - Objects exemplified in Eiffel
- **Bacon's Diagrams**



JAVA
Thumbnail Sketch

- **What is Java?**
  A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded and dynamic language.

Several Examples

- **Taken from Deitel & Deitel,**
  Java - How to Program
  - Focus is on concurrency.
  - Comments are about all aspects of Java, but ...
  - Not a comprehensive intro.
  - See the list of books on the COMP2310 web page. Any one will explain details.

D & D Example 1 (1)

- **3 Processes - each sleeps a while then prints.**

```
public class PrintTest {
 public static void main(String args[])
 {
   PrintThread thread1,thread2,thread3;
   thread1 = new PrintThread( "1" );
   thread2 = new PrintThread( "2" );
   thread3 = new PrintThread( "3" );
   thread1.start();
   thread2.start();
   thread3.start();
 }
```

- **Output:**

```
Name:    1;    sleep: 4297
Name:    2;    sleep: 672
Name:    3;    sleep: 27
Thread    3
Thread    2
Thread    1
```

Example 1 (2)

- **Notes on the main program:**
  - Every program has a main method, called `main`
  - Invariably the main method has params: `String args[]`
  - This program has 2 classes: `PrintTest` and `PrintThread`
  - Object `thread1` is *declared* in `PrintThread thread1, ...`
  - `thread1` is *created* by doing `thread1=new PrintThread("1")`
  - `thread1` is *executed* by `thread1.start();`
  - `start` is a inherited by `PrintThread` from `Thread`

Example 1 (3)

- **The PrintThread Class:**

- Declarations & constructor

```
class PrintThread extends Thread {
 int sleepTime;

 public PrintThread( String id ) {
  super( id );

  // sleep between 0 and 5 seconds
  sleepTime = (int) (Math.random()*5000);

  System.out.println( "Name:"+getName()
              + "; sleep: "+sleepTime );
```

- **Notes:**
  - `PrintThread` is a subclass of `Thread` - it inherits.
  - Each instance of `PrintThread` has its own `sleepTime`
  - The param, `id`, to constructor `PrintThread` is its name
  - super(id) assigns name to thread by calling constructor, `Thread(id)`, in superclass

Example 1 (4)

- **PrintThread Class (ctd):**

- The run method

```
public void run()
{
  // put thread to sleep for random interval
  try {
    sleep( sleepTime );
  }
  catch ( InterruptedException exception ) {
    System.err.println( "Exception: " +
                        exception.toString() );
  }
  // print thread name
    System.out.println("Thread " + getName());
  }
```

- **Notes:**
  - `run()` is a invoked by `start()`
  - Every thread needs a `run()`
  - The construction: `try {`*S*`} catch (`*E*`) {`*H*`}` allows for exception *E* in code *S* to be handled by code *H*.

D&D Example 2 (1)

- **1 Producer and 1 Consumer sharing a cell.**

```
// Shows multiple threads modifying a
// shared object.

public class SharedCell {
 public static void main( String args[] )  {
   HoldInteger h = new HoldInteger();
   ProduceInteger p = new ProduceInteger(h);
   ConsumeInteger c = new ConsumeInteger(h);

   p.start();
   c.start();
 }
}
```

- **The main program**
  - Probably in its own file
  - Depends on classes HoldInteger, ProduceInteger & ConsumeInteger
  - Declaration & creation of objects done together here.
  - `HoldInteger h = new HoldInteger();` is equivalent to

```
        HoldInteger h; h = new
        HoldInteger();
```

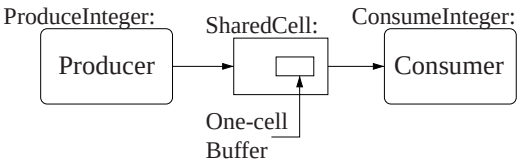
Example 2 (2)

```
class HoldInteger {
  private int sharedInt;

  public void setSharedInt(int val)  {
      sharedInt = val;
    }

  public int getSharedInt()  {
      return sharedInt;
    }
}
```

- **Context:**



- **Note:**
  - Use of `public` and `private`

Example 2 (3)

```
class ProduceInteger extends Thread {
  private HoldInteger pHold;

 public ProduceInteger( HoldInteger h )
 {
  pHold = h;
 }

 public void run()
 {
  for ( int count = 0; count < 10; count++ ) {
```

```
    pHold.setSharedInt( count );
    System.out.println(
     "Producer set sharedInt to " + count );

    // sleep for a random interval
    try {
      sleep( (int) ( Math.random() * 3000 ) );
    }
    catch( InterruptedException e ) {
      System.err.println( "Exception "
                          + e.toString() );
    }
   }
  }
}
```

Example 2 (4)

```
class ConsumeInteger extends Thread {
  private HoldInteger cHold;

  public ConsumeInteger( HoldInteger h )
  { cHold = h; }

  public void run()
  {
    int val;
    val = cHold.getSharedInt();
    System.out.println(
          "Consumer retrieved " + val );

    while ( val != 9 ) {
      // sleep for a random interval
      try {
        sleep( (int) (Math.random() * 3000 ));
      }
      catch( InterruptedException e ) {
        System.err.println(
              "Exception " + e.toString() );
      }
      val = cHold.getSharedInt();
      System.out.println(
            "Consumer retrieved " + val );
    }
  } }
```

Example 2 (5)
The Output

```
Producer set sharedInt to 0
Consumer retrieved 0
Producer set sharedInt to 1
Consumer retrieved 1
Producer set sharedInt to 2
Producer set sharedInt to 3
Consumer retrieved 3
Producer set sharedInt to 4
Producer set sharedInt to 5
Consumer retrieved 5
Consumer retrieved 5
Producer set sharedInt to 6
Consumer retrieved 6
Consumer retrieved 6
Producer set sharedInt to 7
Producer set sharedInt to 8
Producer set sharedInt to 9
Consumer retrieved 9
```

- The **problem** is that actions **setSharedInt** & **GetSharedInt** are not forced to take turns.

D & D Example 3 (1)

```
// Fig. 13.5: SharedCell.java
// Show multiple threads modifying shared
// object.  Use synchronization to ensure
//  that both threads access the shared
//  cell properly.

public class SharedCell {
 public static void main( String args[] )
 {
   HoldInteger h = new HoldInteger();
   ProduceInteger p = new ProduceInteger(h);
   ConsumeInteger c = new ConsumeInteger(h);

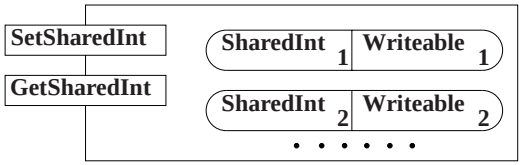
   p.start();
   c.start();
 }
}
```

- **Class ProduceInteger as above:**
- **Class ConsumeInteger as above:**

Example 3 (2)

```
class HoldInteger {
  private int sharedInt;
  private boolean writeable = true;

public synchronized
              void setSharedInt(int val) {
    while ( !writeable ) {
      try { wait(); }
      catch ( InterruptedException e ) {
         System.err.println(
            "Exception: " + e.toString());
      }
    }
    sharedInt = val;    writeable = false;
    notify();
  }
```



Class **HoldInteger** is a monitor

Example 3 (3)

```
public synchronized int getSharedInt() {
  while ( writeable ) {
     try {
        wait();
     }
     catch ( InterruptedException e ) {
        System.err.println(
           "Exception: " + e.toString() );
     }
  }

  writeable = true;   notify();
  return sharedInt;
}

}
```

Heavyweight Processes in Java

- **Relevant classes are**
  java.lang.Process
  java.lang.Runtime
- **Process Creation:**
  - Done by one of four forms of **exec** from **Runtime**
  - **exec** takes a **String** param
    - the executable of the program to be executed
    - and its parameters.

    `public Process exec(String cmd) throws IOException`
  - Another version takes an array of **String** parameters.

Use of exec - Example

- **Three step recipe**
  - Get an instance of the runtime class object:

    `rt = Runtime.getRuntime();`
  - Compute the command to be executed as a string:

    `cmd = "ls -als Java-1.1.5";`
  - Execute the process (which produces a listing of the Java 1.1.5 directory):

    `proc = rt.exec(cmd);`
  - This process is an object, with methods.
- **Not UNIX-specific!**

Normal Process Termination

- **Awaiting process completion**
  - This is done using `int waitFor()` from **Process**
- **Exit value of a process**
  - Normal completion gives exit value of 0.
  - Process can choose to give a nonzero exit value.

    `System.exit(status)`
  - Can be discovered using `int exitValue()` from **Process**
- **Preparation**
  - Should stop all threads first.

Termination with Prejudice

- **Murder!**
  - This is done using `public abstract void destroy()` from class **Process**
  - `destroy()` does nothing if the process has finished.
- **Don't rely on GC**
  - Garbage collection does not kill processes
- All above methods apply to **Process** instances.

Pipes

- **Process I/O assumptions**
  - It is expected a process that comes from an **exec** has a **stdin**, **stdout** & **stderr**
- **Getting at these channels**
  - Write to the standard input of the **exec**'ed process using **OutputStream** `getOutputStream()` from **Process**.
  - Read from standard output of the **exec**'ed process using **InputStream** `getInputStream()` from **Process**.
  - Ditto standard error.

The Class Runtime

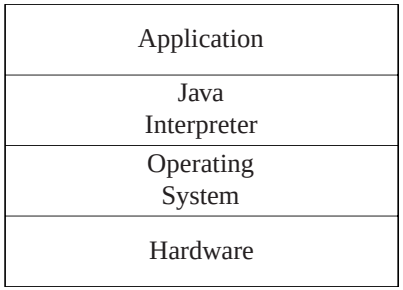
- **We have seen exec**
- **Memory**
  - Can discover how much is free and what the total amount is.
- **Explicit prompt to gc**
- **Loading libraries**
  - Loads dynamic library whose name is given by default or as a parameter.
  - Often the dynamic libraries contain native methods.
- **Tracing**
  - Both instructions & method calls can be traced.

Summary

- **Java Allows Concurrency**
  - Both as threads and heavyweight processes
- **Threads**
  - Allows priority scheduling.
  - Has monitors.
- **Processes**
  - Similar to Unix processes. (. . . but does not clone)
  - Provides pipes to stdin and stdout of spawned process.

Operating Systems
Virtual Machinery

- **Reference**
  - Bacon, Chapter 2, esp. 2.3–2.6
- **OS as an Abstract Machine**



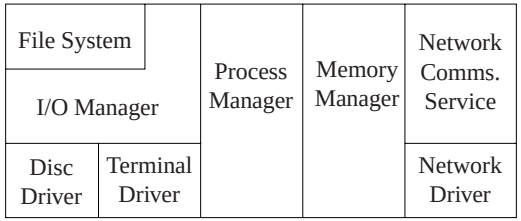
- **Operating System Functions**
  - To manage resources
    - protection
  - To provide services
    - to users
    - applications
    - internally

Virtual Computers
Bacon Slides

- **Figure 2.12**
  Contrasts user view of machine with reality in architecture.
- **Figure 2.13**
  Bacon's slide of typical closed operating system.

Operating Systems
Structure (1)

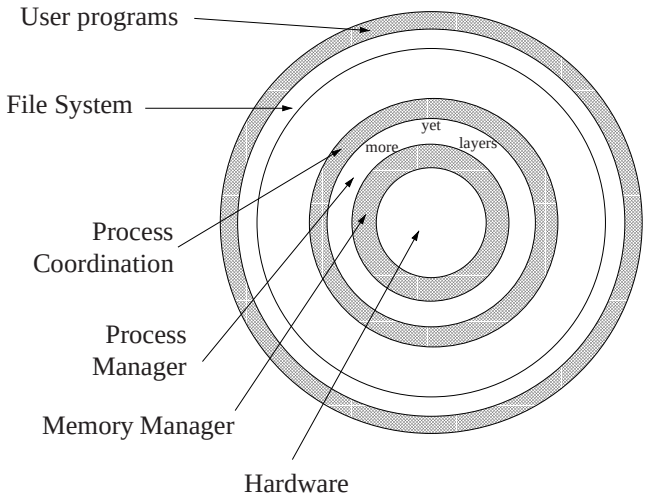
- **The Bacon view**
  - Compare with figure 2.13



- **A problem**
  Bacon is stuck with being general so cannot commit to showing as much layering as is possible/desirable.

Operating Systems
Structure (2)

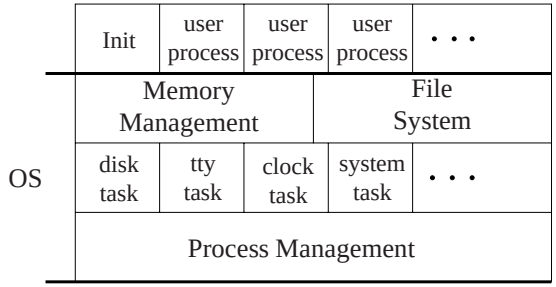
- **The Comer View - XINU**
  - Structure as on PDP11



- Structure on Intel architecture is a little different.

OS Structure (3)

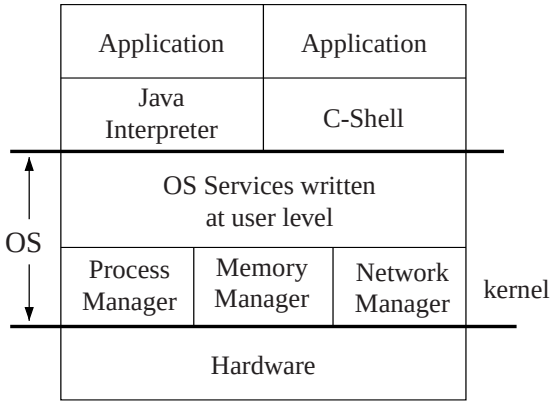
- **The Tanenbaum View - MINIX**



  - Distribution possible!
    File system can be on a separate machine
  - User can only talk to memory manager or file system.

OS Structure (4)
Micro-kernels

- **The aim**
  - Make 'privileged OS' as small as possible
  - As much functionality as possible at 'user level'
  - Compromise between efficiency and flexibility.

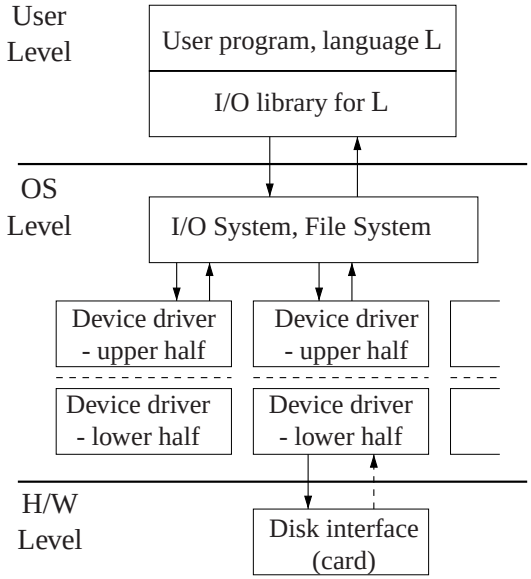


OS Structure (5)
Micro-kernels (ctd)

- **The advantages**
  - Small kernel easier to build and maintain
  - Services above kernel easier to build, maintain and change
  - Kernel optimized for speed
  - User level of OS can be optimized for space
  - OS policy is at user level and thus flexible
  - System configuration can be tailored

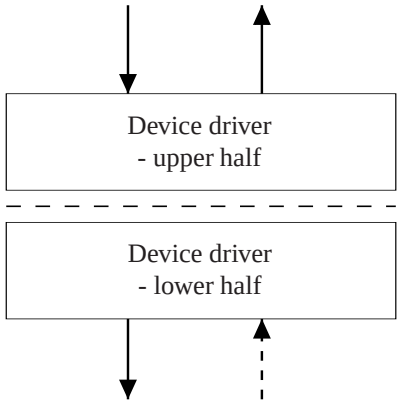
Operating Systems
Device Management (1)

- **Bacon, Chapter 3**
- **The cast in the I/O story**



Device Management (2)
Driver Structure

- **The two-halves structure**



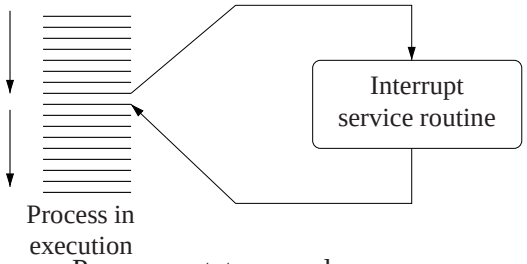
- Upper half is synchronous
  Lower half is interrupt driven
- Each half communicates with the other by shared memory and by semaphores.
- Each half may need to *wait* on the other; each can *signal* the other

Device Management (3)
Driver Examples

- **Terminal Input:**
  - Upper half:
    getchar returns when char is there.
    (use of semaphore)
  - Lower half:
    activated by external events; handles cooked mode.
- **Terminal Output:**
  - Upper half:
    putchar returns if space is there.
    (use of semaphore)
  - Lower half:
    activated by device-ready interrupt;
- **Disk I/O:**
  - Upper half:
    Block supplied to be read/written
  - Lower half:
    disk head scheduling

Interrupts

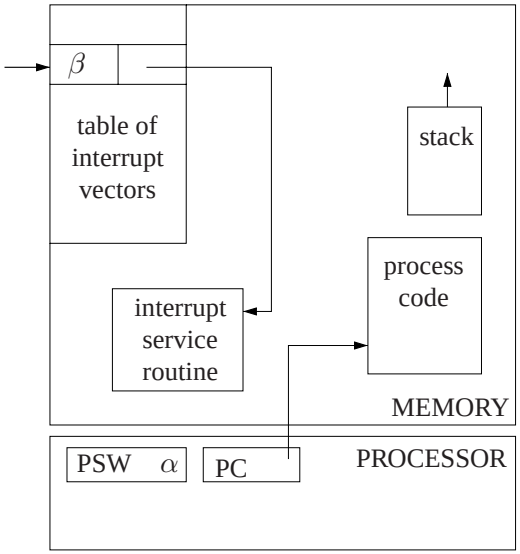
- **See section 3.2 of Bacon**
- **What happens**



- Processor status saved (PC, PSW, few regs)
- PC set to address of the interrupt service routine appropriate to interrupt
- PSW set appropriately
  Privileged mode on?
  Interrupts disabled?

Interrupt Processing

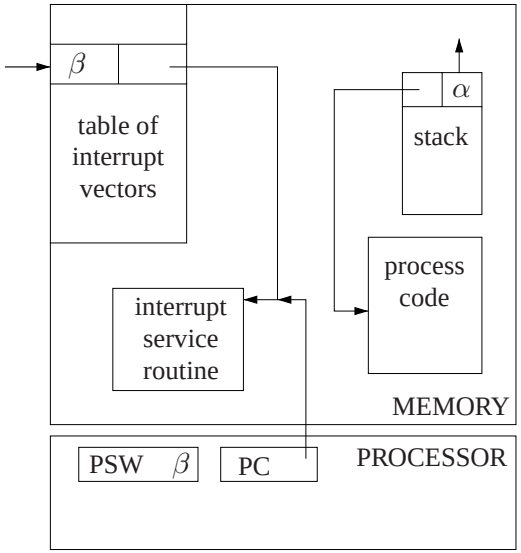
- **Corresponds to Bacon fig. 3.5a**



Interrupt about to occur

Interrupt Processing

- Corresponds to Bacon fig. 3.5b



Interrupt has just occurred

Interrupts and Device Management (Bacon Slides)

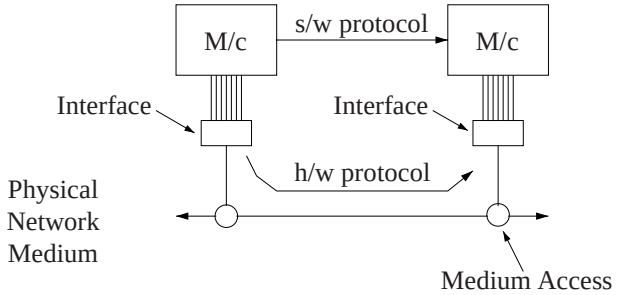
- **Figure 3.9 of Bacon**
  - MC68000 Interrupt vectors
- **Figure 3.3a of Bacon**
  - Polled interface
- **Figure 3.3b of Bacon**
  - Interrupt driven interface

Communications Management

Virtual vs Real

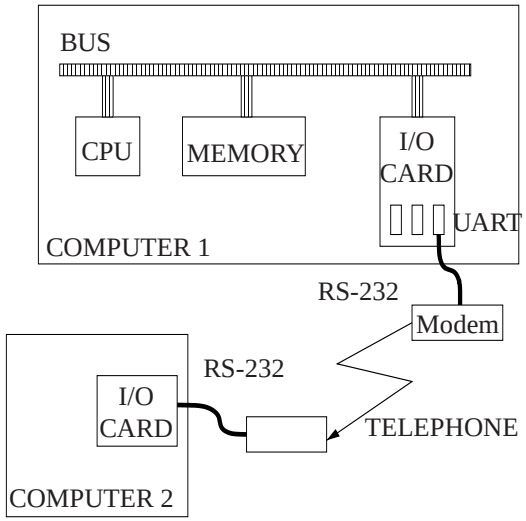
- **See sections 3.6, 3.7 of Bacon**
  - High level protocol for easy communications at application level. (m/c = machine)

  - Low level protocol is needed for implementation on actual communication medium.



Media (1a) RS-232C

- **Computer-computer connection**

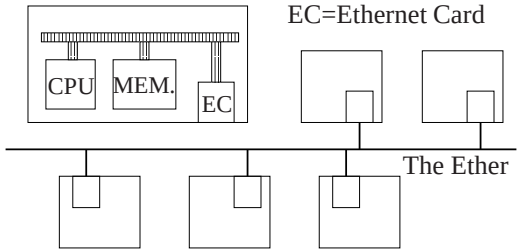


Media (1b)
RS-232C

- Bus is bit parallel, dozens of wires
- RS-232 is bit-serial, several wires
- Telephone is modulated bit serial
- UART = Universal Asynchronous Receiver-Transmitter
- Reference: Tanenbaum

Media (2)
Ethernet

- **Network connection**



- **The ether is coaxial cable**
  - Frequency is of order of 10 MHz.
  - Cable is a transmission line.
- **Packets 64–1518 bytes**
  - Header, sumchecks give minimum packet size.

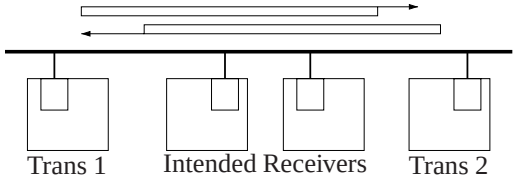
Media (3)
CSMA/CD

- **Carrier Sense, Multiple Access** with **Collision Detection**
  - **C**arrier **S**ense means that a transmitter checks that 'the ether is quiet' before sending
  - **M**ultiple **A**ccess means any station can talk to any other
  - **C**ollision **D**etection means transmitter receives and checks its own signal.
  - Repeat collision unlikely because of random, exponentially-increasing backoff before retransmission.

Media (4)

Ethernet Length

- **Cable has limited length**
  - Transmission speed is approx. speed-of-light

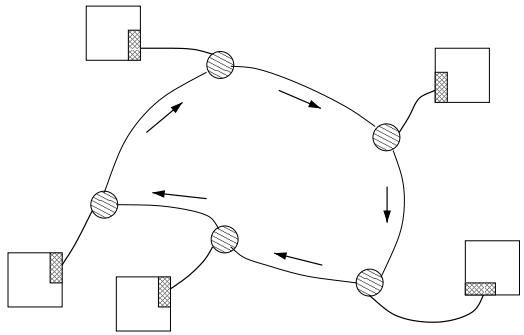


- **Limiting Case**
  - Max length is a few km

$$\frac{250\,\text{bit} * 200\,000\,\text{km.s}^{-1}}{2 * 10\,000\,000\,\text{bit.s}^{-1}}$$

Media (5)
Token Rings

- **FDDI**
  - Fibre Distributed Data Interface
  - 100 Mb.s^{-1}
- **Cambridge Ring etc.**



Media (6)

Token Rings (ctd)

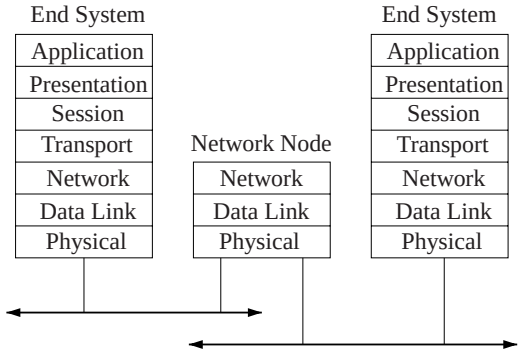
- **Typical setup**
  - Source awaits empty car;
    Loads car and raises flag;
    - car is now occupied.
  - Destination station reads car contents;
    Marks message as received.
  - Source station sees that car and marks it empty;
    Does not re-use it.
- Upper level duties
  - Small blocks from large.

Media (7)
LAN Comparisons

Ethernet	10 $Mb.s^{-1}$
IEEE 802.4 token bus	10 $Mb.s^{-1}$
FDDI token ring	100 $Mb.s^{-1}$
Cambridge Ring	10 $Mb.s^{-1}$
Cambridge Fast Ring	100 $Mb.s^{-1}$
Cambridge Backbone Ring	1000 $Mb.s^{-1}$

ISO/OSI Reference Model (1)

- Reference: Bacon, section 3.8
- Open Systems Interconnection
  - framework for discussion and comparison.



ISO/OSI Reference Model (2)

- Physical layer / Data link layer
  - bits over medium between 2 computers, error free
- Network layer / Transport layer
  - packets over a route, standard service
- Session layer
  - client level access
- Presentation layer
  - provides independence of representation

Memory Management

- See Chapter 6 of Bacon
- Memory Hierarchy
  - Registers, cache, main memory, disk storage
- Address Space
  - MMUs and dynamic relocation
  - Virtual memory
- Protection
- Segmentation
  - Sharing
- Paging
  - Page tables
  - Page replacement policies & hardware

File Management

- See Chapter 7 of Bacon
- Functions of File Manager
  - Secure information storage (Protection against malice and misadventure)
  - Directory service (Organisation!)
- Sharing
- Networked file systems
- Memory mapped files
- Persistence

Process Management (1)
Processes & Design

- Reference: Bacon, Chapter 4
- Where *should* processes be used?
  - With multiple processors
  - With asynchronous components
  - For independent tasks
- Processes and virtual machines
  - Each process a separate machine?
    - Can view it that way
  - Processes can be hidden

Process Management (2)
Requirements

- OS must support several fictions
  - *Separation* where the hardware is shared

- *Sharing* where the hardware is separate

- Process should be an ADT with the following operations :
  - Create, Kill
  - Suspend, Run
  - Communication operations.

Process Management (3)
Process Descriptors

- Main use is for state-saving

Process Descriptor

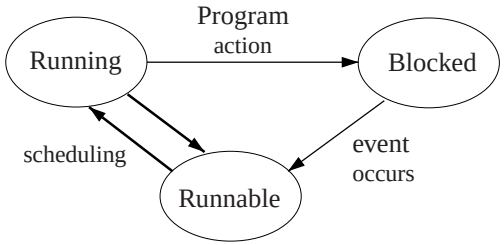
Process ID
Process State
Saved PSW
Saved SP
Saved Registers
Cause of Wait
Swap Address
Various times
Queues Link
File Desc. Table
Other stuff

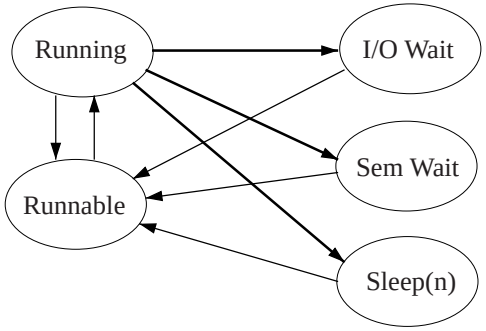
- What is process state?
  - Enumerated type (running, ready, blocked)
  - ... or something more elaborate

Process Management (4)

State Transition Diagrams

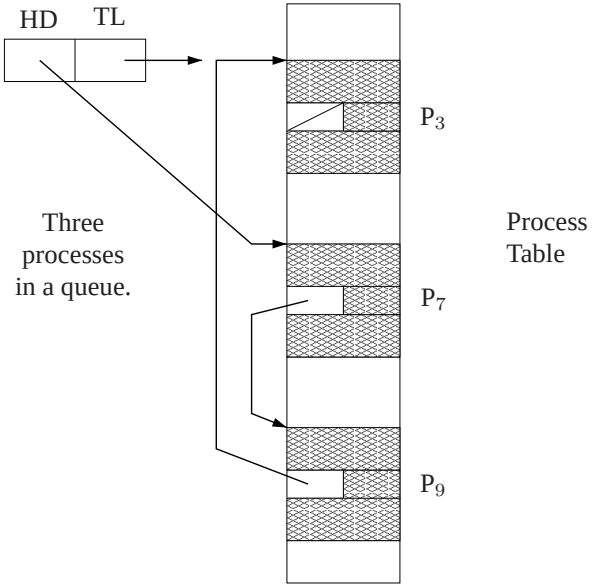
- They show which process state transitions are sensible.





Process Management (5)
Queue of Process

- Processes in a queue are linked through the process table. Only 1 linkage cell is needed for various queues.



CSP
Communicating Sequential Processes

- **Just a Language Fragment?**
  - Assignments, sequencing
  - Conditionals and loops,
    - Dijkstra style
  - Processes and families of processes
  - Uniform Input/Output
- **Influence of CSP**
  - Straight implementations
  - Occam
  - Ada

CSP
Introduction

- **Lasting Concepts**
  - Unification of process communication and I/O
  - Nondeterministic choice among possible inputs
  - Distributed termination
- **Supplied Reference**
  - C.A.R. (Tony) Hoare, *Communicating Sequential Processes*, - Communications of A.C.M., August, 1978.

Conventional Conditionals

- **Conventional Conditionals.**

```
if (x==y) {
   flag = true;
}
```

```
if (a>b) {
   max = a;
} else {
   max = b;
}

switch (col) {
  case red:
     stop();
     break;
  case orange:
     accelerate();
     break;
  case green:
     go();
}
```

Dijkstra-Style Conditionals

- **Guarded Commands**

```
[ x=y → flag := true]

[
a > b → max := a []
a ≤ b → max := b ]

[
col=red ⟶ stop() []
col=amber ⟶ accelerate() []
col=green ⟶ go() ]
```

Syntax Of Guarded Alternatives

- **Possible CSP Statement is:**

```
[
g1 → S1 []
g2 → S2 []
....
gn → Sn ]
```

  - Each g_j is a boolean expression. Each S_j is a statement.
- **Terminology**
  - $g_i \rightarrow S_i$ is called a *Guarded command*
  - *Guard* (g_i) is a condition that must hold before command (S_i) executed

Semantics of Guarded Alternatives

```
[ g1 → S1 []
  g2 → S2 []
  ....
  gn → Sn ]
```

- If no guard is true the alternative statement *fails*
- Otherwise, one of the guarded commands with true guard is chosen and executed
- Yes, there is *Nondeterminism!*
- e.g.

```
[
a ≥ b → max := a []
a ≤ b → max := b ]
```

Guarded Command Repetition (1)

- **Conventional Constructs**

```
while (i <= n) {
  sum = sum + f(i);
```

```
  i = i+1;
}
```

- **Dijkstra-Style Repetition**

```
*[
i ≤ n → sum := sum + f(i);
i := i+1 ]
```

- **Syntax**

```
*[ g1 → S1 []
   ....
   gn → Sn ]
```

Guarded Command Repetition (2)

- **Semantics**
  - On each iteration, one of the guarded commands with true guard is executed.
  - The choice is made non-deterministically
  - Repetition continues until all guards false

More Examples (1)

- **Factorial**

```
i := 2; fac := 1;
*[ i <= n →
fac := fac * i;
i := i+1 ]
```

  - Computes the factorial of *n* in variable *fac*
  - Suppose n = 4; when *i* is 2, 3 & 4 *fac* is updated and *i* is incremented.

More Examples (2)

- **Greatest Common Divisor**

```
x := a; y := b;
*[
x < y → y := y-x []
y < x → x := x-y ];
gcd := x;
```

  - Computes the gcd of non-negative integers, *a* & *b*, in variable *gcd*
  - Exit condition x=y
  - This is Euclid's Algorithm

Input and Output Commands (1)

- **Syntax**
  - input command: *process_name* ? *target_variable*
  - output command: *process_name* ! *expression*
- **Examples**

cardreader ? cardimage

lineprinter ! lineimage

File(i) ? c

Q ? x

P ! (y+2)

Input and Output Commands (2)

- **Rendezvous (handshake)**
  - If Q ? x is a command in P and P ! (y+2) is a command in Q then P and Q can rendezvous & exchange data.
  - If it happens, then x is set to the value of y+2
  - No buffering

Input Commands as Guards

- **Example:**

```
i ≤ n; j:  integer; X?j
                  → k:integer;
X?k; Z!(j+k)
```

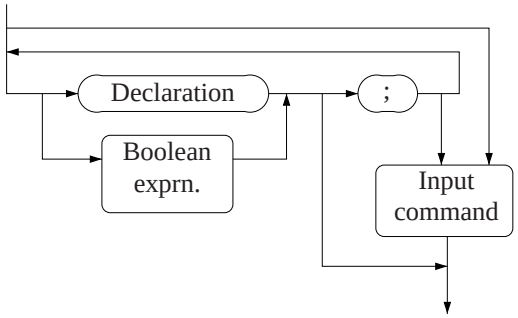
  - The guard consists of a boolean expression, a declaration and an input command.
  - So, what is the general form for a guard?

Complete Syntax of Guards

- **Full Syntax of Guards**

```
<guard> ::= <guard list>   |
            <guard list> <input cmd>  |
            <input cmd>
<guard list> ::= <guard elt> {; <guard elt>}
<guard elt>  ::= <bool expr> | <declaration>
```

Guard



Inputs in Guards: The Semantics

- **Evaluation of Guards**
  - The semicolon acts as *and*
  - Declarations evaluate true
  - Boolean expressions evaluate normally
  - Supposing an input command is P?v. It will be deemed:
    - true, if P is waiting to make a rendezvous
    - false, if P has terminated already
    - pending, otherwise
- **That Example, again:**

```
i ≤ n; j:  integer; X?j
                  → k:integer;
X?k; Z!(j+k)
```

  - If i≤n then this guard can be selected *when* X tries to send an integer to this process. Then X?j is executed.

CSP Example 1

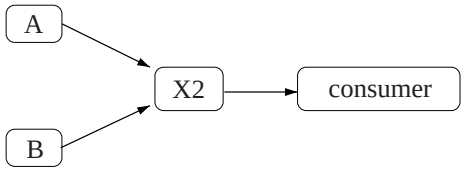
'Copy'

```
X::  *[character; west?c
→ east!c ];
```

- X terminates when west terminates
- This is called distributed termination
- X fails if east terminates before west
- X acts as a buffer

Example 2 'Collect'

```
X2::  c:  character;
*[
A?c → consumer!c []
B?c → consumer!c ] ;
```



- X2 terminates when A and B both terminate
- communication with A and B can interleave arbitrarily.
- This is where the non-deterministic choice wins.

Example 3

'Squash'

- A program which copies its input, replacing each pair of asterisks by an up arrow.

```
X::
*[
c:  character; west?c →
[ c ≠ asterisk → east!c []
  c = asterisk → west?c;
  [ c ≠ asterisk
        → east!asterisk; east!c []
    c = asterisk → east!uparrow
  ] ] ]
```


```
x**2 + c*y**2 ⇒ x↑2 + c*y↑2
```

- We assume the input does not end with an asterisk.

Example 4 'Disassemble'

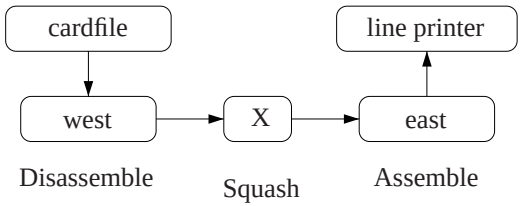
- A program which extracts the stream of characters contained on a file of cards.

```
west::
  *[
cardimage:  (1..80) char;
cardfile?cardimage →
      i:  int; i:= 1;
      *[ i ≤ 80 →
         X!cardimage(i); i := i+1];
      X!space
]
```

- **Assemble**
  - Stream of characters to lineprinter of width 125.

Example 5
Conway's Problem

```
[west::  Disassemble ||
       X::  Squash ||
 east::  Assemble ]
```

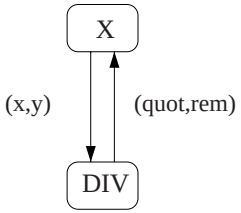


- Hard to do elegantly in other languages

Example 6
Division

```
DIV::

*[x,y:  integer; X?(x,y) →
  quot:  integer; quot := 0;
  rem:  integer; rem := x;
  *[rem ≤ y → rem := rem - y;
              quot := quot+1];
  X!(quot,rem) ];
```

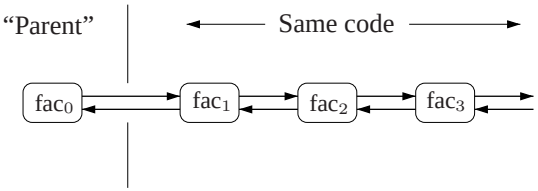


  - Gives both quotient and remainder as results.
  - Closest thing to a procedure in CSP

Example 7
'Factorial' (a)

```
fac(i:1..limit) ::
    *[
n:  integer; fac(i-1)?n →
[ n=0 → fac(i-1)!1 []
  n>0 → fac(i+1)!(n-1);
        r:integer; fac(i+1)?r;
        fac(i-1)!(n*r) ]];
```

  - This is code for a family fac(1), fac(2), ... fac(limit)



Factorial (b)

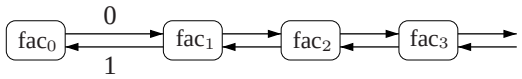
- **One of the family**

```
fac(4) ::
    *[
n:integer; fac(3)?n →
[ n=0 → fac(3)!1 []
  n>0 → fac(5)!(n-1);
        r:integer; fac(5)?r;
        fac(3)!(n*r) ]];
```

- **The nature of process arrays**
  - Note that in fac(i) the subscript is not a variable. It only has a role at compile time.

Factorial in Action (a)

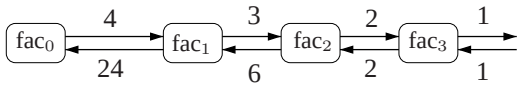
- **Factorial of 0**
  - fac(0) sends 0 to fac(1)




```
fac(1) ::
    *[
n:  integer; fac(0)?n →
[ n=0 → fac(0)!1 []
  n>0 → fac(2)!(n-1);
        r:integer; fac(2)?r;
        fac(0)!(n*r) ]];
```

Factorial in Action (b)

- **Factorial of 4**




```
fac(1) ::
    *[
n:  integer; fac(0)?n →
[n=0 → fac(0)!1 []
 n>0 → fac(2)!(n-1);
       r:integer; fac(2)?r;
       fac(0)!(n*r)]];
```

  - When fac(1) sends 3 to fac(2) it ultimately gets back 6 (the factorial of 3).

– It then can return 4*6 to fac(0).

Types in CSP

- **Primitive Types**
  - integer 666
  - character 'A'
  - real 3.14159
- **Structured Types**
  - Arrays (0..99) integer
  - Pairs (4,y)
  - Constructor applied to arguments.

    Cons(true,y)
- **Structured Values**
  - An expression of structured type that contains no vars.

    List(4,nil)

    List(2,List(4,nil))
- **Structured Targets**
  - An expression of structured type that contains no vars.

    (x,y)

    List(a,List(b,nil))

Assignments

- Can have a structured value assigned to a structured target.

```
(x,y) := (2,3)
```

Same as x:= 2; y:=3

```
(x,y) := (y,x)
```

Swaps values of x, y

```
has(x,y) := has(2,3)
```

Same as x:= 2; y:=3

```
C(x,y) := C(y,x)
```

Swaps values of x, y

```
List(a,List(b,nil)) :=
                    List(2,List(4,nil))
```

- Can have a structured value assigned to a less structured target.

  `List(a,List(b,nil)) := List(2,c)`
- Assignment compatibility!

Multiple Channels

- Can have a structured value sent from an output.
- Can have a structured target mentioned in an input command.

  `P?has(x,y)` in Q can rendezvous with `Q!has(2,3)` in P.

  `P?has(x,y)` in Q can't rendezvous with `Q!(2,3)` in P.
- This allows multiple channels between two processes.

Example 8
Table Lookup

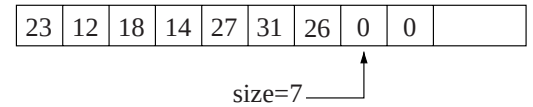
```
S ::
   content:  (0..99) int, size:  int;
   size:=0;

*[n:  int; X?has(n) → SEARCH;X!(i<size) []
  n:  int; X?insert(n) → SEARCH;
       [i<size → skip []
        i=size; size<100 →
             content(size) := n;
             size := size+1;
] ]
```

SEARCH is an abbreviation for

```
      i:  int; i:=0;
      *[ i_size; content(i) ≠ n → i :=
i+1 ]
```

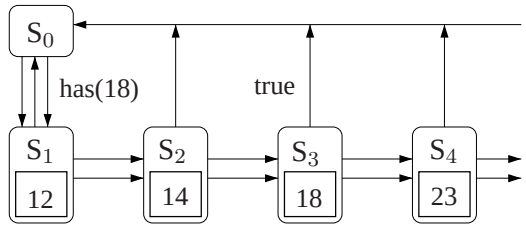
content:



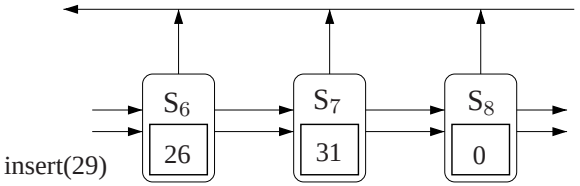
Example 9
Parallel Sets

- **Exercise:**

  Read and understand the section 4.5 in the CSP paper.
- **Consider membership tests:**



- **.. and insertions:**



Eventcount, An abstract Data Type

- Reed and Kanodza, CACM 22(2), February 1979, 115–123
- **Information Content**
  The state of an eventcount is captured as:
  - A counter
  - a queue of processes
- **The Operations**

advance(E)	Increment counter of the eventcount E and return the new value of E.
await(E,n)	Suspend the current process until the counter of eventcount E is greater than or equal to n.
read(E)	Return value of counter of E.

No need for initialisation;
Counter always starts at 0.

Eventcounts Uses

- **Mutual Exclusion**
  - In conjuction with sequencers (see below)
- **Synchronisation**
  - Standard scheme follows
- **Resource Control**
  - Such as in bounded buffer scheme
- **Similar scope to semaphores**

Using Eventcounts for Synchronization

- **Consider:**

```
while (true) {
    produce();
    advance(eventA);
}
```

```
for (i=1;i<=infinity;i++) {
    await(eventA,i);
    consume();
}
```

- **This pattern ensures**
  - produce_2 happens before consume_2
  - produce_j happens before consume_j

Using Eventcounts for Rendezvous

- **Consider:**

```
for (j=1;j<=infinity;j++) {
  head_of_cycle();

  advance(eventB);
  await(eventA,j);

  tail_of_cycle();
}
```

```
for (i=1;i<=infinity;i++) {
  head_of_cycle();

  advance(eventA);
  await(eventB,i);

  tail_of_cycle();
}
```

The Abstract Data Type, Sequencer

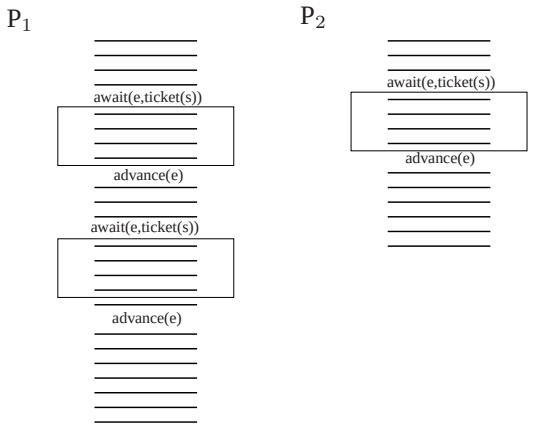
- **Information Content**
  The state of a sequencer is captured as: a counter.
- **The Operation**

ticket(S)	Return the value of the counter of sequencer S and increment it.

- **The Idea**
  - The operation *ticket* is just like taking a number when queueing for service (in a deli or shoe store etc.)
  - Sequencer counters are initialised to 0.
  - Invariably used in conjunction with eventcounts.
  - A sequencer plus an eventcount is the deli system.

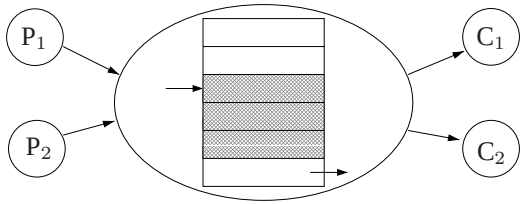
Using Eventcounts for Mutual Exclusion

- Use an event count and a sequencer, e & s, both initialised to 0.
- Entry protocol is
  await (e, ticket(s))
- Exit protocol is
  advance(e)

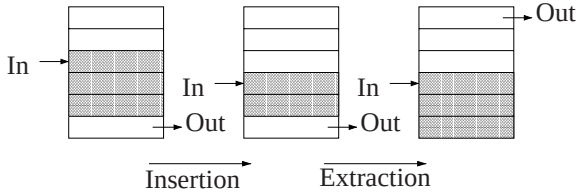


The Producer-Consumer Problem

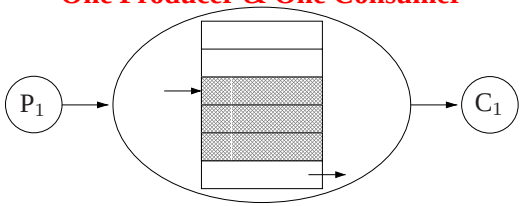
- Recap



- The movie



One Producer & One Consumer



- **Data Structures**

`BUF[0..N-1]`	the buffer, as before
`IN, OUT`	Eventcounts initialised to 0.
`P, C`	Sequencers for production & consumption.

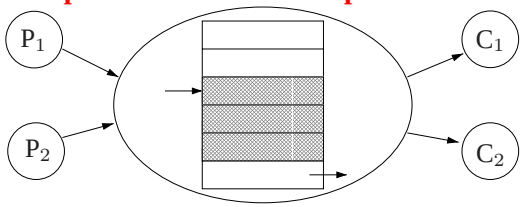
- **Operations**
  - **Insertion operation**

```
j = ticket(P);
await(OUT, j-N+1);
   BUF[j % N] = Item;
advance(IN);
```

  - **Extraction operation**

```
k = ticket(C);
await(IN, k+1);
   Item2 = BUF[k % N];
advance(OUT);
```

Multiple Producers & Multiple Consumers



- **Data Structures**
  As above.
- **Insertion operation**

```
j = ticket(P);
await(IN, j);
await(OUT, j-N+1);
   BUF[j % N] = Item;
advance(IN);
```

- **Extraction operation**

```
k = ticket(C);
await(OUT, k);
await(IN, k+1);
   Item2 = BUF[k % N];
advance(OUT);
```

What is a distributed system?

- Many types of jobs may be broken up into processes that get the job done.
- Each process may be worked on by a separate processor.
- Multiple processors may co-operate on a single job.
- If processors communicate results to each other, then we have a distributed system.
- Examples: a multi-computer, a computer network, or a network of people.

Why have distributed systems?

Resource sharing: access to (possibly remote) hardware devices and data, no matter where you are.

Openness: key interfaces are published, so you can construct a system with heterogeneous hardware and software.

Scalability: the system has to grow in an ordered way.

Fault tolerance: coping with hardware or software failure.

Transparency: hide the separation of components, so the system can be perceived as a whole.

What this section is about

How to understand, design, and build distributed systems.

- concepts and abstractions
- what makes a good distributed system
- how they can fail
- design choices
- language support for distributed processing (C/UNIX and Java)
- tools to help you to build distributed systems
- how to diagnose problems and fix them

Some basic issues

- How to declare concurrency.
- How to synchronize processors.
- How to communicate 'results'.
- How to prevent processing or communication errors.
- How to recover from them.
- How best to distribute the workload.

Systems we can study

- Message Passing Interface (MPI)
- Sockets and XTI
- Ada, occam, Linda
- RPC, RMI, CORBA, DCOM, ...
- Anything else?

Inter-Process-Communication

- References for today: Bacon, chapters 12–13; Ben-Ari, chapters 7–10
- Ada: *Programming in Ada* & *Programming in Ada 95*, J. G. P. Barnes, Addison-Wesley
- occam and Linda: see Bacon & Ben-Ari for other references

Where have we been?

- Processes:
  - heavyweight (UNIX): `fork`, `exec`, `wait`
  - lightweight (Java threads): `new Thread`, `start`, `wait`, `notify`
  - CSP (rendezvous)
  - interrupts
- Resources: CPUs, files, printers, memory, etc.:
  - producers/consumers; buffers (reader/writer)
  - changing/borrowing/sharing
- Mutual exclusion: critical sections
- Semaphores, monitors, locks (Java `synchronized` methods)

Evolution of interprocess communication primitives (1)

(see Bacon, Figure 12.1)

Given: **Low-level primitives:** Synchronization through event signals or semaphores

Where do we go from here?

Two approaches:

(a) Shared memory

Hide the low-level primitives and enforce structuring:

- Critical regions
- Monitors
- Rendezvous

Evolution of interprocess communication primitives (2)

(b) No shared memory

Make the primitives more powerful—combine data transfer and synchronization:

- Message passing
- Pipes
- Sockets

- Streams
- Remote procedure call

To share, or not to share?

- Systems that share memory can simulate systems that don't.
- Systems that don't share memory can simulate systems that do.
- Shared memory systems and non-shared memory systems are *equally powerful* . . . but not always *equally desirable* to a software engineer!

When to share

(Bacon, §12.5)

- Systems where memory is not well protected, e.g. PCs with 80286 and earlier. All processes – and the OS – run in a single shared address space (or use 'real' addresses)
- The language's runtime system functions as a simple operating system, e.g. embedded and real-time systems.
- A particular program might run on one processor or on a shared-memory multiprocessor, and may have different behaviour/performance characteristics: be careful!

When not to share

(Bacon, §12.6)

- Protected systems, e.g. multi-user systems, where each process runs in its own separate address space.
- Processes run on different computers, e.g. across a local network.
- Systems where you want the freedom to change where processes start executing.
- Systems which migrate processes to achieve load balancing.

Choosing primitives: synchronous or asynchronous?

- Synchronous: participation of sender and receiver; blocking; rendezvous
- Asynchronous: non-blocking, check for message, or interrupt
- The difference: buffering.
- Compare the telephone system and the postal system.
- Unbuffered synchronous: the lower-level concept: occam and Ada.
- Buffered asynchronous: higher level but less efficient: Linda.

Choosing primitives: process identification

- Direct connections, a switching system (telephone exchange), or a bulletin board?

occam Dedicated channels connecting pairs of processes; the most efficient.

Ada A process calls another process by name without divulging its own identity; good for server processes.

Linda Broadcast messages (that need not be signed with a process identifier); greatest flexibility: add or remove processes dynamically.

Choosing primitives: data flow

- One-way or two-way? Telegrams or telephone calls.
- Asynchronous systems (e.g. Linda) use one-way data flow.
- Synchronous systems use channels; then decide one one-way (occam) or two-way (Ada).
- A trade-off between expressiveness and efficiency. Are you mostly sending messages that don't need a reply? Or, do you expect most messages to need a reply?

Choosing primitives: Process creation

- Do all process exist at program startup time, or can processes be created dynamically?

Why would you want dynamic creation?

Flexibility Design the program without knowing how many processes will be needed.

Dynamic use of resources Match number of processes to requirements at runtime: save memory and improve efficiency.

Load balancing Add new processes as the workload increases.

Static or dynamic?

- Static creation works best in embedded systems (air traffic and medical monitoring) where configuration is fixed and predictability is important!
- Dynamic creation for large transaction processing systems (airline reservation). Computing requirements change during

the day; you can't halt the system to change configuration!

Message passing (refresher)

(Bacon, chapter 13)

- Used for systems that can't share memory (or don't want to)
- Used for synchronization between processes.
- Used for data transfer between processes.
- Used over networks.
- Messages may be sent to processes on different computers, on different CPUs on the same computer, or on the same CPU.

A typical message

(Bacon, Figure 13.1)

- Two parts to a message:
  - Message header (used by the transport mechanism):
    - Destination
    - Source
    - Type of message
  - Message body (used by the communicating processes)
- The source information is likely to be either inserted or checked by the OS!
- Messages may be sent via pipes, sockets, or other network methods.
- Transmission is usually arranged by the OS.

Basic message passing (1)

(Bacon, Figure 13.2)

- Process **A** wants to send a message to process **B**.
- Assumptions:
  - each process knows the identity of the other
  - it is appropriate for the sender to specify a single recipient, and for the recipient to specify a single sender
  - there is an agreement about the size of the message and its content (an application protocol).
  - asynchronous, buffered message passing

Basic message passing (2)

- **A** builds up the message and executes the SEND primitive. It doesn't block.
- **B** reaches the point where it needs to synchronize with and receive data from **A**, so it executes WAIT.
- If **B** isn't WAITing when the message arrives, (a copy of) the message is stored in **B**'s message buffer for the time being.
- If **B** calls WAIT before the message arrives, its execution is blocked.

Message passing options

(Bacon, §13.5)

- Receive from 'anyone'
- Send the same message to more than one process
- Request & reply primitives (useful for client/server systems; wait for a reply to acknowledge receipt)
- Wait only a certain time for a reply; automatic resending
- Wait for a particular message from one process, and not just the oldest (matching message types, ports)
- Expire messages

Broadcast and multicast

(Bacon, §13.5.6)

- You may want to send to many processes at once.
- Can use a special destination code in the header.
- The implementation can then send the one message to multiple processes.
- This is more efficient than executing the same SEND message many times.
- RFC1983 defines the following terms:

  unicast An address which only one host will recognize.

  multicast A packet with a special destination address which multiple nodes on the network may be willing to receive.

  broadcast A special type of multicast packet which all nodes on the network are always willing to receive.

Message timeout

Bacon (§13.5.8)

- You may want to stop waiting after a while, if you don't receive a message.
- Specify a timeout period: stop waiting after n ms, even if I haven't received my

message.

- Useful in case the source process has crashed, or if a message has been lost on the network.
- Very useful in real-time systems, where things must happen in a certain time, or else!

Message expiration

- You may want to replace out-of-date messages before the receiver process has looked at them.
- For example, if you are tracking an object (a satellite or a part of a large industrial process), you only care about its current position.
- Discarding out-of-date messages saves on processing time for the receiver.
- Can tell the implementation: just keep the most recent message of this type (set the buffer size to 1).

Case Studies: Ada, Occam & Linda

Case study: Ada

(Ben-Ari, chapter 8)

- Developed for DoD as their standard language for critical systems. Now Ada 95.
- Communication in Ada is synchronous and unbuffered.
- Two tasks must meet in a rendezvous in order to communicate. The first one to arrive must wait for the arrival of the second.
- Remember: both tasks are executing concurrently, and they only synchronize at the rendezvous.

The rendezvous

- The location of the rendezvous belongs to one of the tasks, called the accepting task. The other task, the calling task, must know the identity of the accepting task and the name of the location of the rendezvous.
- However, the accepting task does *not* know the identity of the calling task.
- Great for programming servers!
- A task is divided into two sections, the specification and the body. The specification may only contain declarations of entries.

A buffer

Task specification:

```
task Buffer is
  entry Append(I: in  Integer);
  entry Take  (I: out Integer);
end Buffer;
```

A sample call:

```
Buffer.Append(I);
```

Task body:

```
task body Buffer is
begin
  ...
  accept Append(I: in Integer) do
    ... statements
  end Append;
  ...
end Buffer;
```

What happens?

- The accepting task is an ordinary sequential process. The `accept` statement is executed in sequence when the instruction pointer reaches it, except that its definition requires synchronization with a calling task.
- When both tasks meet:
  1. The calling task passes its `in` parameters to the accepting task and blocks.
  2. The accepting task executes the statements in the accept *body*.
  3. The `out` parameters are passed back to the calling task.
  4. The rendezvous ends; both tasks are no longer suspended.

Degenerate bounded buffer

```
task body Buffer is
  B: array(0..N-1) of Integer;
  In_Ptr, Out_Ptr: Integer := 0;
  Count: Integer := 0;

begin
  loop
    accept Append(I: in Integer) do
      B(In_Ptr) := I;
    end Append;
    Count := Count + 1;
    In_Ptr := (In_Ptr + 1) mod N;
```

```
    accept Take(I: out Integer) do
      I := B(Out_Ptr);
    end Take;
    Count := Count - 1;
    Out_Ptr := (Out_Ptr + 1) mod N;
  end loop;
end Buffer;
```

Much better ...

```
task body Buffer is
  B: array(0..N-1) of Integer;
  In_Ptr, Out_Ptr: Integer := 0;
  Count: Integer := 0;

begin
  loop
    select -- non-deterministically!
      when Count < N =>
        accept Append(I: in Integer) do
          B(In_Ptr) := I;
        end Append;
      Count := Count + 1;
      In_Ptr := (In_Ptr + 1) mod N;
    or
      when Count > 0 =>
        accept Take(I: out Integer) do
          I := B(Out_Ptr);
        end Take;
      Count := Count - 1;
      Out_Ptr := (Out_Ptr + 1) mod N;
    end select;
  end loop;
end Buffer;
```

More Ada

The `select` statement:

- Evaluate the guards. If there are calling tasks waiting on entry queues for open alternatives, a rendezvous is commenced with the first task on one of those queues.
- Alternatives: `select` with `else`, `delay`, or `terminate` alternatives.

More bits and pieces:

- dynamic task creation
- task priorities

Occam

Summary:

- What's in a name?
- CSP
- Unusual syntax
- Every statement is a process
- 'Assembler' for Transputer networks
- 'Minimalist' support for distributed programming ...
- ... based on rendezvous

Occam versus Ada

Ada	occam
rendezvous at `accept` or `select` *belonging* to a task	channels declared in scope visible to both processes
calling task must know name of accepting task, name of entry and parameter signature	channel protocol defines signature of data that can flow through channel
many calling tasks can rendezvous with the one entry	every channel connects exactly two processes: one only outputs, the other only inputs
dynamic process creation	static process creation
can have 'global' variables	no 'global' variables: can not load-store to common memory
non-deterministic choice of open entries in a `select`	choose first open alternative

Occam peculiarities

- There are assignments, input statements, and output statements (plus the usual conditional and looping constructs, but no recursion!)
- *Every* statement is considered to be a process. It is up to the programmer to indicate *explicitly* whether statements will be combined in sequence or in parallel.

```
SEQ
  statement_1
  statement_2
  statement_3
```

or

```
PAR
  statement_1
  statement_2
  statement_3
```

- Indentation is used to group statements.

Occam Sieve (1)

```
VAL INT n IS 50:       -- # of primes to generate
VAL INT limit is 1000: -- range to check
[n-2] CHAN of INT link:  -- links between filters
[n-1] CHAN of INT prime: --channels to Print process
CHAN OF INT display: -- output display to device 1
PLACE display AT 1:

PROC Starter(CHAN OF INT out,print)
INT i:
  SEQ
    print ! 2
    i := 3
    WHILE i < limit
      SEQ
        out ! i
        i := i + 2: -- generate odd numbers

PROC Sieve(CHAN OF INT in,out,print)
INT p,next:          -- filter out one prime
  SEQ
    in ? p     -- p is prime
    print ! p
    WHILE TRUE
      SEQ
        in ? next
        IF (next\p) <> 0 -- evaluated first
          out ! next
          TRUE    -- IF needs a true guard
        SKIP:
```

Occam Sieve (2)

```
PROC Ender(CHAN OF INT in,print)
INT p:        -- consume rest of numbers
  SEQ
    in ? p
    print ! p
    WHILE TRUE
      in ? p:

PROC Printer([] CHAN OF INT value)
INT p:     -- print each prime, in order
  SEQ i = 0 FOR SIZE value
    SEQ
      value[i] ? p
      display ! p:

PAR              -- main program
  Starter(link[0],prime[0])
  PAR i = 1 FOR n-2
    Sieve(link[i-1],link[i],prime[i])
    Ender(link[n-1],prime[n-1])
    Printer(prime)
```

Linda

- Linda was developed by David Gelernter in the 1980s. It is a small set of parallelism primitives which can be added to any language. So there is C-Linda, Ada-Linda, etc.
- The Linda system provides a tuple space. This is a region of 'shared' memory, accessed only by the Linda primitives. The tuple space is used to store typed tuples of objects.
- The Linda primitives use pattern matching on the tuple signatures. None of these match:
- `(1,'A')` (integer,character)
- `(1,2)` (integer,integer)
- `('A',1)` (character,integer)

Linda primitives

The primitives (where `T` is a tuple):

- `Output(T)` Add `T` to the tuple space.
- `Input(T)` Remove a matching tuple from the tuple space. If there isn't one right now, suspend until there is.
- `Read(T)` Just like `Input(T)`, but don't remove the tuple.
- `Eval(T)` Each element is evaluated, and the results are packaged into a tuple and stored in the tuple space. This call is non-blocking: the evaluations happen in 'parallel'.

More Linda primitives

- `Try_Input(T)` Non-blocking `Input(T)`. Continue if there isn't a matching tuple (return a status value).
- `Try_Read(T)` Non-blocking `Read(T)`.
- Alternative names: `out`, `in`, `read`, `inp`, `readp`, `exec`.

More Linda

- 'Parameter' (message) passing:

```
Output(1, 'A');
  -- one process creates arguments
```

```
Input(I: Integer; C: Character);
  -- one process receives them
```

- Job allocation to processes:

```
Input("job", J: Job_ID, 24);
  -- process 24 waits to serve
Output("job", 17, 24);
  -- send request to process 24
Output("job", 17, P: Process_ID);
  -- send request to any process
```

Remote procedure call

```
-- code for caller
Output("Proc", 24, 65, 'X');
Input(24, B: Boolean);

-- code for accepting process
Input("Proc", Caller: Process_ID,
      I: Integer; C: Character);
Proc(I, C, B);
Output(Caller, B);
```

Futures

A programming construct specifying that the value of some computation will be needed at some later point, but allowing the system to schedule that computation to run at any arbitrary time. Futures are one way to present *lazy evaluation* in shared-variable programming. (Greg Wilson)

```
-- caller; Compute_job1 is a function
Eval("job1",Compute_job1);
-- do some computation
-- -- -- -- --
-- now we need the answer
Input("job1", Answer: Integer);
```

Linda comments

- Atomicity: each operation on the TS must appear to be indivisible.
- TS is unprotected: *any* worker can gain access to *any* information. Unsuitable for e.g. multiple clients/servers
- Tags are linear character sequences: no hierarchy and no scoping. This is bad for abstractions, and hinders the development of large programs.
- TS operations are slower than 'normal' shared memory or message passing operations (due to unique copies and exclusive access)
- Even more expensive on distributed architectures: hardware broadcasting helps.

Sockets, streams, XTI

- W. Richard Stevens, *UNIX Network Programming* and *Advanced Programming in the UNIX Environment*.
- Two main ways of doing **interprocess communication** (IPC) in UNIX: sockets and XTI.
- XTI and its predecessor TLI are implemented using streams.

OSI network model

- Details of message transmission vary from the voltage and current used (low level), up to the format of the message (high level).
- ISO has produced the Open Systems Interconnection Reference Model (OSI)

The layers:

1. Physical (hardware, OS)
2. Data link (OS)
3. Network (IPv4, IPv6)
4. Transport (TCP, UDP, or bypass)
5. Session (Apps)
6. Presentation (Apps)
7. Application (Apps)

- Sockets and XTI bridge levels 4 and 5.

Sockets

- Berkeley UNIX (BSD) provides **sockets**. (They're also in SVR4.)
- Compare file I/O: `open`, `creat`, `close`, `read`, `write`, `lseek`. One argument is a **file descriptor**.
- If would be nice (transparent) if this could be transferred to network I/O.
- Sockets let you do this!
- Given a socket file descriptor, you can call `fdopen` and get a nice, buffered stream that you can call `fprintf` etc. on.
- **Extremely important:** Don't confuse the two uses of the word 'stream'!

Protocols and names

- Sockets use a **client-server protocol**. This is not symmetrical: initiating a network connection requires that the program know which rôle (client or server) it is to play.
- A network connection can be connection-oriented (e.g. TCP) or connectionless (e.g.

UDP). The former is more like file I/O.

- Names are more important in networking:
  - You can do file operations with only a file descriptor: you don't need to know the original name of the file.
  - A networking application may need to know the name of its peer process to verify that the process has authority to request the services.

Connection-oriented or connectionless

- Connection-oriented protocol ⇔ concurrent server
- Connectionless protocol ⇔ iterative server
- For a connection-oriented use: first start the server: `socket`, `bind`, `listen`, `accept`. Then clients can start: `socket`, `connect`.
- Once connected, server and client communicate using `read` and `write`.
- For a connectionless use: use `socket` and `bind`, but then use `recvfrom` and `sendto` to communicate without establishing a 'connection'.

inetd

- One of the most important UNIX processes!
- Handles requests: smtp, ftp, telnet, daytime, etc.
- Uses configuration files to specify protocols and ports, user ids, filenames and parameters; sets up a socket for each service
- Uses `select` call to listen for requests
- Maybe do a `fork`/`exec`
- Maybe not, e.g. daytime is handled iteratively

Streams (1)

- Streams are used in System V to implement the **Transport Layer Interface** (TLI) and the **X/Open Transport Interface** (XTI) (the System V equivalents of sockets).
- Streams provide a full-duplex connection between a user process and a device driver. (It could be a pseudo-device driver.) There's no need for this to be associated with a hardware device.
- Data transferred up and down a stream consists of **messages**.
- Each message contains a **control** part and a **data** part (their content is defined by the application).

Streams (2)

- There are two system calls to read and write these messages: `getmsg` and `putmsg`. (There's also a `poll` call to support multiplexing (see below).
- A process can add modules between the stream head (the system call interface) and the device driver.
- Any number of modules can be 'pushed' onto a stream. Each new module gets inserted just below the stream head (a LIFO stack).
- An example of a processing module is one to implement a terminal line discipline (protocol). The driver just inputs and outputs characters and it is the terminal line discipline that implements features such as special character handling, forming complete lines of input, etc.

Streams (3)

- Streams provide a nice layered implementation of the networking system. A streams module that accepts data from multiple sources is called a **multiplexor**.
- For example, the TCP/IP daemon builds the multiplexor when it is started (typically at system initialisation).
- Before streams, the link between a process and most device drivers (other than disks) was the UNIX character I/O system.
- Without streams, if you didn't have the source code for UNIX and wanted to add a new feature to the kernel, you wrote a character device driver to do what you wanted. You accessed the driver through the `open`, `close`, `read`, `write`, and `ioctl` system calls.

Streams (4)

- (This is how most third-party networking software is implemented on systems without streams.)
- The problem with this is that this facility just isn't adequate or efficient enough for implementing network protocols. And also, the modules required to implement protocols at layers above the device driver don't belong in the device driver.

- Using streams, a site without source code can add new stream modules to their system in a fashion similar to adding a new device driver.
- Oops, what about Linux?

XTI

- In 1986, AT&T released TLI with SVR3, which contained only streams and TLI building blocks, i.e. no TCP/IP, which was provided by third parties.
- SVR4 (1990) provided TCP/IP as part of the basic OS.
- XTI defined and released by the X/Open Group in 1988: a superset of TLI.
- Posix.1g standard started with XTI.
- Programming techniques the same as with sockets.
- What's different: function names, function arguments, and some of the 'nitty-gritty details'.

Further references

- Reading: DS chapter 2.
- More on sockets: DS §4.5, Bacon §23.16.2
- More on System V streams: Bacon §23.17.5

Challenges

Distributed systems have problems that shared memory systems don't:

- Individual computers can crash.
- Message transmission between processes may be unreliable.
- Message transmission may take significant time.
- The computers may not all agree on what time of day it is.

All of these can lead to inconsistencies in the state of the system.

Crashes

- References: DS 13 & 15, Bacon 14
- A distributed system may involve many computers.
- What happens if one crashes?
- Suppose that a process has side-effects before it crashes, e.g., partially writes to a file, sends instructions to other processes ...
- More generally, suppose that an operation has an externally visible effect but is interrupted ...
- How do you keep the system consistent?

Modelling crashes

- A simple model for a server crash is the fail-stop model.
- It assumes that the crash happens instantly, that the system stops immediately, and doesn't go wild.
- Hardware crash: all registers, cache, MMU, volatile main memory are gone!
- Changes to persistent state are assumed to be correct, but may be incomplete.
- Software crash: clients can determine that there is a failure in the server.
- Many crashes don't behave this way!

Crash resilience

(DS §15.3, Bacon §14.3)

- If a computer fails to complete a job, we may want the system to appear as though that job had never started.
- That way, we could run the whole job again later, without having to undo the work done by the failed computer.
- Crash resilience is the extent to which an application has support to recover from system crashes ...
- ...it's also called crash or failure transparency, or fault tolerance.
- 'A fault-tolerant system can detect a fault and either fail predictably or mask the fault from its users.' (DS p. 462)

Idempotent operations

(DS §4.3, Bacon §14.4)

- An operation is idempotent when running it once gives exactly the same effect as running it multiple times.
- If the expected acknowledgement of a request fails to arrive, you can just send the request again.

Examples: Which of the following are idempotent?

- Delete a file.
- Eat your lunch.
- Add $100 to your bank balance.
- Set your bank balance to $101.
- Send an e-mail message to your friend.

Atomic operations

(DS §12.1 & 12.4, Bacon §14.5)

- An operation on some object is atomic iff:
  - when it terminates successfully, all effects are made permanent.
  - else, there is no effect at all.
  - it doesn't interfere with any other operations on the same object.
- Atomic operations either succeed entirely, or else they fail without harm.
- Atomic operations need not be idempotent.

Examples: Which of the following are atomic operations?

- `x = x + y*y/4;`
- `wait(semaphore);`
- `if (x == 3) stop = TRUE;`

Transactions

- Transaction processing systems (TP systems) allow the building of atomic operations called transactions.
- Define the start and end of a given transaction, then either:

  commit: operation completes successfully and the effects are guaranteed to be permanent

  abort: failure to complete one of the atomic operations (so rollback)
- Important point: if the system says that an atomic operation has been 'done', then the changes must be recorded in permanent store.

Stable storage

DS §15.4

- Writing a file to disk doesn't guarantee that the file will persist. A disk error might cause a failure.
- We can create stable storage, by writing the same data to two disks simultaneously.
- That way, even if one write fails, the other will (almost certainly) succeed.
- (It's a bit more complicated than this! See DS for details.)

Implementing atomic operations: logging

(DS §15.2, Bacon §14.6.1)

- If we keep a log of every operation, then we can roll back to a consistent state when an operation fails.
- This gives the same effect as an atomic operation.

Write-ahead log:

- Keep a permanent log of what you change before you change it.
- Record the old value, then the new value, then change the value over.
- That way, you can always revert the system to its former state, even in a crash (i.e. reverting must be idempotent).

Example

Suppose x = 4, y = 0

Action	Log
begin	Start
x := x + 1	x = 4/5
y := x * 2	y = 0/10
x := x + y	x = 5/15
end	End

Implementing atomic operations: shadowing

(DS §15.2, Bacon §14.6.2)

- Make a copy of all of the data you want to work with.
- Change the data in the copy.
- Swap the copy for the original (in a single operation, i.e. atomically).
- That way, either the operation worked, or else it didn't (it isn't 'half-done').

Non-volatile RAM (NVRAM)

(DS p. 233, Bacon §14.7)

- Main memory (Random Access Memory) whose contents survive computer shutdown.
- For caching data in a file storage device: change the contents of memory first, then write out to disk.
- To hold the data structures for a file service: write out the NVRAM only when the changes are consistent.
- To buffer requests for disk access to use the disk device efficiently.
- To implement shadowing.

NVRAM risks

- If a computer crash is not fail-stop, it may risk corruption of permanent storage (e.g. disks).
- To write out to disk, you must follow a special protocol: set various registers and call a particular system call or interrupt.
- When writing to NVRAM, you don't follow a protocol: it's just like modifying 'normal' memory.
- NVRAM has more risk of corruption if the crash is not fail-stop.

So, what's the point?

- NVRAM is persistent, as disks are.
- Disks have intricate arrangements in their blocks.
- These arrangements help protect against/detect catastrophic failures.
- Computer memory has no such protections. A system can write anywhere in memory that it chooses.

What has this got to do with distributed systems?

- The more computers you rely on, the greater the probability of partial failure.
- The more computers you rely on, the more resilient your system must be to partial failure.
- Yet, the more computers you rely on, the more you depend on 'short cuts' to make the processing faster.
- Yet another classic trade-off!

Distributed IPC

- References: DS 4, 5, 10, Bacon 15
- IPC = inter-process communication
- This includes the low-level details (e.g. message passing) ...
- ... and the high-level details (e.g. how the system is organized and programmed)
- Today: the high-level stuff

Special characteristics of distributed systems

(DS pp. 46–49, Bacon §5.5)

Independent failure modes: The components of the system and the network may fail independently of each other.

No global time: Each system component must keep track of its own time.

Inconsistent state: The effects of one component take time to propagate to other components; in the mean time, the system may be left in an inconsistent state.

How to agree on time?

- A distributed system may run on many computers, each connected by a *relatively slow* network.
- Each computer must keep track of the time.
- How *often* should they synchronize?
- How *reliably* can they synchronize?

How computer clocks work (1)

- Some computer clocks are built from quartz crystal (like a watch).
- When a current is passed through the crystal, it vibrates at a set rate.
- This rate creates clock ticks that can be used to tell time.
- There may be thousands of clock ticks each second.
- The tick rate may vary, depending on temperature, pressure, current and manufacturer's standards.
- Because computer operations are fast, computer clock precision is more important than for most wristwatches.

How computer clocks work (2)

- Some computer clocks work directly off the main current.
- AC current resembles a sine wave, with a fixed frequency.
- This frequency is generated by the power supply, and by transformers.
- The frequency may fluctuate over time, or be interrupted.
- Thus, the clock speed may change over time.
- In either case, we say that clocks are subject to **clock drift**.

How to create a single, logical time

(DS 10.3, Bacon §5.6)

- Send the time with each message.
- If the time of an incoming message is later than my current time, then advance my current time to be after the time of the incoming message.
- This is Lamport's Algorithm.
- See Figures DS 10.5, 10.6, Bacon 5.3, 5.4.

Comments on Lamport's Algorithm

- Creates a logical time for all transmitted events on a distributed system.
- The logical time may bear no resemblance to real time.
- Only transmitted events appear on the logical timeline.
- Local events appear in order on the local computer, but we can't tell whether they happen before or after local events on another computer unless we transmit a message.

Alternative methods

- Have each node broadcast its local time to all the others. Put the local clock forward to the value of the fastest clock.
- Periodically connect to a special time server, running e.g. the Network Time Protocol (NTP). Adjust the local clock by counting more or fewer ticks per second over a period of time.

Client/server review

- When a client wants something done, it sends a request message to a server.
- The server replies when the job is done.
- The programmer must insert send/receive messages into the code to make this happen.
- The programmer may have to deal with message or server failure in each distributed program.
- Is there a more convenient way to program a distributed system?

Remote procedure calls

(DS 5, Bacon §§15.6–15.7)

Motivation

- Client/server communication is based on I/O.
- A distributed system should look and feel like just one CPU.
- → remote procedure calls!

Idea

- A process is called as a procedure on another machine.
- The calling process is suspended until the result is returned.
- Data is transmitted through variables, just like a normal procedure call.

Two approaches

- Integrate RPC mechanism into a programming language that has a notation for defining interfaces. (Java RMI)
- Add a special-purpose interface definition language (IDL) for describing the interfaces between clients and servers. (Sun RPC, CORBA, DCOM)
- In both cases, aim for **transparency**: make remote procedure calls as much like local procedure calls as possible; no distinction in syntax (Birrell and Nelson).

RPC: how it works

1. Client procedure calls the client stub.
2. Client stub builds a message and traps to the kernel.
3. Kernel sends the message to the remote kernel.
4. Remote kernel gives the message to the server stub.
5. Server stub unpacks the parameters and calls the server.
6. Server does the work and returns the result to the stub.
7. Server stub packs it in a message and traps to the kernel.
8. Remote kernel sends the message to the client's kernel.
9. Client's kernel gives the message to the client stub.
10. Client stub unpacks the result and returns to the client.

RPC: parameter passing (1)

Parameter marshalling

- How do you pack parameters into a message?
- Different machines have different character representations, e.g. ASCII/EBCDIC
- Different machines have different byte numbers for integer representations, e.g. 12345 (base 10) = (0,0,30,39) in PC memory, (39,30,0,0) in SPARC memory
- One's complement/two's complement inconsistencies.
- Many different floating point number representations
- How to pass pointers to local memory?

Solutions

- Client and server each know the types of the parameters.
- Can arrange a canonical form for message transmission.
- Stubs can handle conversion to and from the canonical form.
- Not very efficient if the machines are the same!
- Stub procedures may be generated automatically, by the compiler.

RPC: parameter passing (2)

Canonical forms

- A network standard representation used for all RPC parameters transmissions.
- E.g., ASCII character set, two's complement integers, 0/1 for Boolean values, IEEE floating point values, all other data little-endian.
- Client/server stubs must convert their parameters into this form before transmission across the network.

Pointer mechanisms

- Pointers to structures of known length can be handled by passing the structures
- Pointers to arbitrary data structures (e.g. graphs) may be forbidden.
- Alternatively, send pointers, and the server may dereference each pointer by passing messages back to the client: very inefficient!

RPC (Part 2): dynamic binding

References: DS 5, Tanenbaum §2.4.

- In an RPC context, dynamic binding is used to match up clients and servers.
- Each server declares its specification to a stub generator.
- The stub generator produces a server stub and a client stub.
- Clients linked with the appropriate client stubs at compile-time.
- Servers linked with the appropriate server stubs at compile-time.
- (Doesn't have to be this way, e.g. CORBA and DCOM.)

Diversion: Sun RPC (1)

- Reference: `http://pandonia.canberra.edu.au/ClientServer/rpc/rpc.html`
- We want these remote procedures:
  ```
  long bin_date(void);
  char *str_date(long);
  ```
- The program with these specified as remote procedures for a remote machine would define the two functions `bin_date` and `str_date` in file `rdate.x`:
  ```
  program RDATE_PROG {
    version RDATE_VERS {
      long   BIN_DATE(void) = 1;
      string STR_DATE(long) = 2;
    } = 1;
  } = 1234567;
  ```

Sun RPC (2)

- Run this file through `rpcgen`. You get:

`rdate.h` a header file for both client and server sides.

`rdate_svc.c` a set of stub functions for use on the server side. This also defines a full main function that will allow the server side to run as a server program i.e. it can run and handle requests across the network.

`rdate_clnt.c` a set of stub functions for use on the client side that handles the remote call.

Sun RPC (3)

Functions are generated from the specification as follows:

- The function name is all lower-case, with `_1` appended.
- On the client side the function generated has two parameters, on the server side it also has two; the extra parameter differs between sides.
- The client side function has either the one parameter of the spec, or a dummy `void *` pointer (use `NULL`) as first parameter.
- On the client side, the second parameter is a handle created by the C function `clnt_create()`.
- On both sides, the function return value is replaced by a pointer to that function value.

Sun RPC (4)

In this example, `rdate_clnt.c` defines and implements

```
long *bin_date_1(void *, CLIENT *);
char **str_date_1(long *, CLIENT *);
```

On the server side, `rdate_svc.c` refers to (and you must implement):

```
long *bin_date_1(void *, struct svc_req *);
char **str_date_1(long *, struct svc_req *);
```

- Now you must write client-side wrappers and implement those '`_1`' functions at the server end.
- The client's main program must call `clnt_create()` before calling any remote procedure, and pass this client object to the various client `_1` functions.

Registration with the binder

- Server exports its interface to the binder: it gives the binder its name, version number, a unique identifier (e.g. 32 bits), and a handle to locate it.
- The handle may be some sort of network address, etc.
- The server may register or deregister itself (`rpcb_set` and `rpcb_unset` on Sun).
- Clients issue lookup messages to the binder to locate and import servers.
- The binder ensures version compatibility.
- Disadvantages: (a) extra overhead of exporting and importing interfaces (b) centralized binder may create a bottleneck (c) multiple binders may need to be informed of a server's registration.

RPC failure semantics

Some common kinds of failures:

1. Client can not locate server
2. Request message from the client is lost
3. Reply message from the server is lost
4. Server crashes after receiving a request
5. Client crashes after sending a request

Can not locate server

- Server is down.
- Client wants an older version of the server.
- Use a particular return value, e.g. -1. However, a special return value may not be available in all functions.
- Raise an exception, which the client program may trap. However, not all languages have such facilities (e.g. Pascal). This also weakens RPC transparency.

Lost request messages

- These can be handled by the OS kernel.
- After sending a request, start a timer. If nothing comes back in a certain amount of time, resend the request.
- If there is no reply after n messages have been sent, then assume that the server can not be located (i.e. we're back to the previous condition).

Lost reply messages

- Difficult for the client to distinguish the condition from lost request or slow server.
- Many servers are stateless, so they don't care whether the client gets the message.
- For idempotent requests, a client can repeat the request after a timeout.
- Some requests are not idempotent, e.g. credit transfer.
- Client may assign each request a sequence number. The kernel can keep track of each client's current sequence number, and resend a reply to a request with the same sequence number, if it has already done the work.
- Client request headers may distinguish repeat requests from new requests.

Server crashes

- If a server crashes before receiving a request, then treat it as case 1.
- Otherwise, a server crash may occur (a) after receiving but before processing a request, or (b) after processing the request, but before replying.

At least once semantics: wait until the server reboots, then retry. But what if the work already completed successfully?

At most once semantics: give up, and report failure. But maybe the work was actually done?

Exactly once semantics: desirable, but very hard to achieve!

- This problem makes RPC behave very differently from single-processor systems.

Client crashes (1)

- May produce useless server computations, called orphans. These tie up the CPU and other resources, such as files that may only be accessed by one process. Also, after rebooting, may resend RPC requests that are already active, and get confused when the reply from the orphan comes in!

Extermination: clients keep logs of what they are doing. After a reboot, then inspect the logs and send messages to the servers to exterminate orphan computations.
This uses a lot of disk space, and servers themselves may be clients. Thus, orphans and grandorphans, etc. may exist.

Client crashes (2)

Reincarnation: each client stores an epoch number. On reboot, it broadcasts a new epoch. Old epoch computations are killed, and old

epoch replies can be easily detected.

Gentle reincarnation: after receiving an epoch broadcast, a server tries to locate the owner process of the current computation. If it can not find the owner, then the computation is killed.

Expiration: each RPC has a quantum T to finish its work. If a server needs more time, it must request that time from the client (this is a nuisance!). If after a crash, a client waits for time T before restarting, then all orphans will die.

What is a reasonable T? It depends on the kind of RPC.

Killing orphans can be dangerous when they are performing file locking or making requests of other services.

RPC limitations

- Transparency is limited
- Remote access to global variables (e.g. UNIX `errno`) is hard to implement.
- Weakly-typed languages (such as C) use null-terminated strings of arbitrary size, etc., which makes it hard to marshal parameters correctly.
- Does not work well with pointer-based data structures.
- Can not always deduce the number and types of parameters from a formal specification of the code, e.g. `printf`.

Java

- Java in the news: Sun versus Microsoft
- What is Java 'good' at?
- Security policies; sandboxes
- Internationalization (I18N) and localization (L10N); Unicode, etc.
- Database connectivity (JDBC)
- 'Migration' of classes and objects: RMI, applets, servlets, aglets, ...
- 'Write once, run anywhere' (or is it 'crash everywhere'?)

Problems

- Is it possible to pin down the Java language?
- Standardization process is underway
- Sun's JDK changes a lot between releases; 1.2 has major differences
- Example:

```
System.out.println("Hello world!");
```

This is no longer (1.1.4) recommended. Instead, use a `PrintWriter`:

```
PrintWriter out =
  new PrintWriter(System.out,true);
out.println("Hello world!");
```

More problems ... for us

- The Java class library is huge (in the finest tradition of Smalltalk and Modula-3).
- Books can't keep up with the software.
- So ... we need to use the online hypertext documentation ...
- ... but even that isn't complete, so some experimentation is called for
- There are defects in the JDK (I found one in 1.1.4)

Problems it solves

- Adding active/interactive content to web pages (not just cute animations)
- That content runs in a secure environment, and can be multi-threaded
- Examples: dynamic front-ends to information services, communications programs (net chat), dissect a frog

What you get

- Apart from `java.lang` and `java.util`, etc., you get:

`java.applet`	implement applets
`java.awt`	GUI stuff (→ JFC)
`java.beans`	reusable components
`java.io`	I18N I/O classes
`java.lang.reflect`	reflection
`java.net`	sockets, etc.
`java.rmi`	RMI (Java RPC)
`java.security`	security policies
`java.sql`	database access
`java.text`	dates, money, etc.

And now:

`javax.servlet`	servlets

Java and distributed systems

- You can use the nice socket interface to do client/server-type stuff.
- Use the web to give you an interface; use Java (applet etc.) to do the 'work'.
- Servlets: a replacement for CGI.
- Aglets (IBM): mobile agents (relies on *serializability*); used in a similar way to applets

- RMI (Remote Method Invocation): a clean implementation of RPC.

RMI

- Define a remote interface
- Write an implementation class
- Write a client (could be an applet and a web page)
- Compile Java source files
- Generate stubs and skeletons
- (Move HTML file to deployment directory)
- Set paths for runtime (CLASSPATH)
- Start remote object registry
- Start server
- (Start applet)

What happens

- The client accesses the registry (binder) and gets back an object (proxy) that implements the remote interface.
- Now you can access fields and invoke methods on the object.
- The methods are caught in the stub, which communicates with the skeleton on the remote machine.
- You can pass objects as parameters and get objects back.
- BUT parameter objects are passed by **value** only.

What's new in 1.2

- Write your own SocketFactory (for encryption, etc.)
- Object activation
- An activation daemon, rmid
- A 'pure Java' ORB

The Getting Started example

- From the online documentation
- A distributed 'Hello World'
- Get a remote 'Hello World' object
- Ask it for its string
- Display it in an applet
- (Alternatively, write a stand-alone client)

Define a remote interface

```
package examples.hello;

public interface Hello
  extends java.rmi.Remote
{
  String sayHello()
    throws java.rmi.RemoteException;
}
```

- public
- extends Remote
- method throws RemoteException

Implementation class (1)

```
package examples.hello;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl
  extends UnicastRemoteObject
  implements Hello
{
  private String name;

  public HelloImpl(String s)
    throws RemoteException {
    super();
    name = s;
  }

  public String sayHello()
    throws RemoteException {
    return  "Hello World!";
  }
```

Implementation class (2)

```
public static void main(String args[])
{ // Create, install a security manager
  System.setSecurityManager(new
    RMISecurityManager());
 try {
   HelloImpl obj = new
     HelloImpl("HelloServer");
   Naming.rebind(
     "//myhost/HelloServer",obj);
   System.out.println(
     "HelloServer bound in registry");
  } catch (Exception e) {
    System.out.println(
      "HelloImpl err: "+e.getMessage());
    e.printStackTrace();
  }
}
}
```

Use remote service (1)

```
package examples.hello;

import java.awt.*;
import java.rmi.*;
```

```
public class HelloApplet
  extends java.applet.Applet
{
  String message = "";
  public void init() {
    try {
      Hello obj = (Hello)Naming.lookup(
        "//" + getCodeBase().getHost() +
        "/HelloServer");
      message = obj.sayHello();
    }
```

Use remote service (2)

```
    catch (Exception e) {
      System.out.println(
        "HelloApplet exception: " +
        e.getMessage());
      e.printStackTrace();
    }
  }

  public void paint(Graphics g) {
    g.drawString(message, 25, 50);
  }
}
```

Write web page to contain applet

```
<HTML>
<title>Hello World</title>
<center> <h1>Hello World</h1>
</center>
The message from the HelloServer is:
<p>
<applet codebase="../.."
    code="examples.hello.HelloApplet"
    width=500 height=120>
</applet>
</HTML>
```

Compile everything

- Hello.java → Hello.class
- HelloImpl.java → HelloImpl.class
- HelloApplet.java → HelloApplet.class
- use rmic:
  `rmic examples.hello.HelloImpl`
- generates `HelloImpl_Stub.class` and `HelloImpl_Skel.class`

Ready to go

- Deploy the files
  - Move everything into an area that can be served by a web server
- Start registry
  - Windows: `start rmiregistry`
  - Unix: `rmiregistry &`
- Start remote server
  `java -Djava.rmi.codebase=http://somewhere/codebase/ examples.hello.HelloImpl`
- Run applet
  - Load into a browser or use appletviewer

The result

You don't need an applet

- You can write a stand-alone client
- The server code stays the same
- In the client:
  - you don't need to refer to the codebase; you can e.g. hard-code the address of the server
  - don't extend Applet (init, paint, etc.)
  - write a main() that e.g. just prints out what you get back from the server
- This doesn't work! You get an exception: no security manager installed.

- So in the client, install one, just as in the server

Wait a minute!

- rmic is a Java program that reads a .class (i.e. binary) file and spits out two .class files. How?
- Java programs can load .class files (i.e. classes) and interrogate them. This feature is called 'reflection'.
- You can get the class object corresponding to any object: every object recognizes `getClass()`. Or use `Class.forName("some_class")`
- Then get access to the fields and methods of that class
- The compiler can be called from the virtual machine: it can generate .class files.

An example

```
import java.io.*; import java.lang.reflect.*;
public class PrStringMethods {
  public static void main(String[] args)
    throws java.lang.ClassNotFoundException  {
    int i;
    PrintWriter out = new PrintWriter (System.out,true);
    Method[] ml = Class.forName (
      "java.lang.String").getMethods();
    for (i=0; i<ml.length; i++)
      out.println(ml[i].toString());
  }
}
```

The output

```
public static java.lang.String
  java.lang.String.copyValueOf(char[])
public static java.lang.String
  java.lang.String.copyValueOf(
  char[],int, int)
public static java.lang.String
  java.lang.String.valueOf(char)
public static java.lang.String
  java.lang.String.valueOf(double)
. . .
public int java.lang.String.length()
. . .
public java.lang.String
  java.lang.String.trim()
```

So what?

- The Java compiler, rmic, and rmiregistry are written in Java, and run in a normal Java virtual machine
- You don't need to have any of the .class files on the client side when you start execution; they get transferred when needed
- You can do reflection calls on any remote object
- For example, find out about 'hidden' fields or methods
- This is like having an interface repository!

Object activation

- New to 1.2
- In the previous model, all remote objects run as threads in the server process
- What if you could create new virtual machines as needed?
- Activation: tell the activation daemon (rmid) that you're interested
- Server-side decision only: client code stays the same
- Closer to an implementation repository than the original model

So why bother?

- A 'pure Java' solution
- No compromises for language-independence
- Uses the Java object model
- What you're meant to do:
  - Use RMI within a Java-only group of servers
  - Use CORBA to talk to servers outside the group
- 1.2 contains a bare-bones ORB to get you started (naming service but no repositories)

Correctness

References: DS §§13.2 & 14.5, Bacon 17, Ben-Ari 2, Tanenbaum §3.5.

- There are two types of correctness properties:

Safety properties: The property must *always* be true.

Liveness properties: The property must *eventually* be true ('eventually' includes 'now').

Examples of safety properties

Mutual exclusion: two (or more) processes may not interleave certain sequences or subsequences of instructions – one process must complete its set before another process can start its own. The ordering of the *processes*

is not important. For example, access to system resources (disks, printers).

Absence of deadlock: a non-terminating system must always be able to proceed doing useful work. If a system gets into a state where it isn't doing anything useful *and* can't respond to any external signal or request, then we say that the system is *deadlocked.* The system is 'hung'. An *idle* system is not necessarily deadlocked.

Liveness and fairness (1)

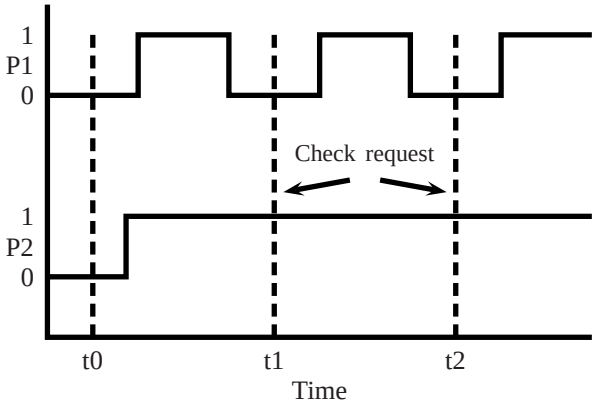
- A program which does nothing will have most safety properties. For a program to be correct, it must also have some liveness properties.

Absence of individual starvation: if a process makes a request for a resource, *eventually* it will be honoured. (This could be after a year!)

- Contention is a reality. We will want to specify how it is handled, via **fairness** properties.
- **Weak fairness**: a process which continually makes a request will eventually have its request granted.

Fairness (2)

- **Strong fairness**: a process which makes a request infinitely often will eventually have its request granted.
- **Linear waiting**: a process which makes a request will have its request granted before any other process has the same request granted more than once.
- **FIFO** (first-in, first-out): a process which makes a request will have it granted before another process, which made the same request but *later*, has its request granted.
- This system is *weakly* fair, but not *strongly* fair:



- Weak and strong fairness are not very practical measures: they don't tell you whether a system is really usable.
- Linear waiting is practical; it enables the calculation of a bound of the time required for a request to be granted.
- FIFO is the strongest type of fairness. It's easy to implement on 'single-box' computers, but for distributed systems, it may not be clear what 'earlier' and 'later' mean, so weaker definitions of fairness are important.

Deadlock

- A set of processes is deadlocked when each process in the set is waiting for an event which can only be caused by another process in the same set.

  'Deadlocks in distributed systems are similar to deadlocks in single-processor systems, only worse.' [Tanenbaum, p. 158]

- Example:

Process A	Process B
Holds the printer	Holds the tape drive
Wants the tape drive	Wants the printer

- Deadlock paths: Bacon figures 17.1 & 17.2.

Conditions for deadlock

Coffman et al., *System deadlocks*, Computing Surveys, June 1971.

A set of tasks is deadlocked if and only if all four of the following conditions hold simultaneously:

Mutual exclusion: Tasks claim exclusive control of the resources they require.

Wait for: Tasks hold resources already allocated to them while waiting for additional resources.

No preemption: Resources can not be forcibly removed from the tasks holding them until the resources are used to completion.

Circular wait: A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain.

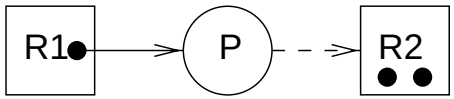
Strategies

Tanenbaum §3.5, DS §13.2

- The ostrich algorithm (ignore the problem).
- Detection (let deadlocks occur, detect them, and try to recover).
- Prevention (statically make deadlocks structurally impossible).
- Avoidance (avoid deadlocks by allocating resources carefully).

All four options are serious alternatives for a distributed system!

Object allocation graphs



- Processes in circles; resources as dots in squares. Here, P has been allocated R1 (solid arrow), and is requesting access to one of the two R2 resources (dashed arrow).
- The idea: avoid cycles!
- See Bacon §17.6 for an example.

Deadlock detection

DS §13.2, Bacon §17.7

- Allocation matrix A, where a_{ij} is the number of objects of type j allocated to process i.
- Request matrix B (same structure)
- Available objects vector V, working vector W
- Basic idea: find a process that could be given the necessary resources. Pretend that you make the allocation and that it completes and then releases all the resources (add them all to W).
- Keep going; any processes left at the end are deadlocked.

Centralized deadlock detection

DS §14.5, Tanenbaum §3.5.1

- Atomic transactions! So no killing off, just aborting. A transaction may well succeed the second time.
- Each machine maintains its own local object allocation graph ...
- ... but also maintain a centralized object allocation graph co-ordinator.
- If a process adds or deletes an edge, it sends a message to the co-ordinator. (Or, the co-ordinator can ask for information when it needs it.)
- Delays (messages arriving in the 'wrong' order) can lead to false deadlock and killing a process when you don't need to.
- A possible solution: use Lamport's Algorithm to provide global time; but then there are more overheads.

Distributed deadlock detection

DS §14.5, Tanenbaum §3.5.1

- Lots of algorithms out there.
- Chandy-Misra-Haas
- Processes are allowed to request multiple resources (e.g. locks) at once. A transaction can be speeded up considerably. But now a process may wait on two or more resources simultaneously.
- Send probe messages around. Each is a triple:
  - Process that just blocked
  - Process sending the message
  - Process to whom it is being sent
- Probes get updated as they go around. If a probe returns to its sender, it is deadlocked!
- A deadlocked process commits suicide.

Deadlock prevention

DS §13.2, Tanenbaum §3.5.2

- Allow processes to hold only one resource at a time.
- Require processes to request all their resources initially.
- Make processes release all resources when asking for a new one.
- All three are cumbersome!
- Impose an ordering on the resources, and require processes to acquire them in strictly increasing order. Then a process can never hold a high resource and ask for a low one: cycles are impossible.

Distributed deadlock prevention (1)

Tanenbaum, §3.5.2

- If you have global time and atomic transactions ...
- Assign each transaction a global timestamp when it starts. Crucial point: no two transactions are ever assigned exactly the same timestamp. (Use process numbers to break

ties.)

- When one process is about to block waiting for a resource that another process is using, check which has the larger timestamp (i.e. is younger).
- Allow the wait only if the waiting process has a lower/higher timestamp (is older/younger) than the process waited for.

Distributed deadlock prevention (2)

- It's *wiser* to give priority to older processes. The system has a larger investment in them, and they are more likely to hold more resources.
- A young process that is killed eventually ages until it is the oldest: so no starvation.
- This setup, where a processes that makes a request either gets it, waits, or is killed, is called wait-die.
- Alternatively: if the requesting processes is older, it can preempt; if younger, it can wait. This is called wound-wait.
- Wait-die can get into a cycle of repeatedly killing a young process.

Deadlock avoidance

Bacon, §17.8 (Banker's algorithm)

- If all processes can specify their total object requirements before they run ...
- To the deadlock detection algorithm, add: Maximum object requirements matrix C.
- The idea: given a request, can it be satisfied without leading to deadlock?
- If the allocation is made, and all the processes then request their maximum allocation of objects, would deadlock then exist? If not, then it is safe to grant the request.
- This is a worst-case analysis.
- Usually need a fallback, if no 'safe' allocation can be made.
- Get real!

Distributed mutual exclusion

References for today: DS §10.4, Bacon §9.3, Ben-Ari, Chapter 11, Tanenbaum, §3.2.

- Distributed processes need to read or update certain shared data structures: need mutual exclusion!
- Centralized algorithm
- A distributed algorithm (Ricart & Agrawala)
- Token passing in a virtual ring of processes

Centralized algorithm (1)

- Simulate how it's done on a single-processor system.
- One process is the co-ordinator.
- Whenever a process wants to enter a critical region, it sends a request message to the co-ordinator, stating which critical region it wants to enter, and asks for permission.
- When it's safe, the co-ordinator replies, and the requesting process enters the critical region.
- If it isn't yet 'safe', the co-ordinator can either refrain from replying for the time being, or send back a message 'permission denied'. The request is queued.

Centralized algorithm (2)

- Guarantees mutual exclusion
- Fair, since FIFO queue and no starvation
- Easy to implement
- Efficient: requires three messages per use (request, grant, release)
- BUT! Co-ordinator is a single point of failure. (And how do you distinguish a dead co-ordinator from 'permission denied'?)
- In a large system, a single co-ordinator can become a performance bottleneck.

Distributed algorithm (1)

- Ricart and Agrawala (1981)
- Need unique global timestamps
- Need reliable communication; a broadcast facility is a bonus
- The basic idea: when a process wants to enter a critical region, it:
  1. Builds a message containing the name of the critical region, its own process number, and the current time.
  2. Sends the message to all other processes (using broadcast if available!).
  3. Waits for 'OK' messages to come back from all other processes.

Distributed algorithm (2)

- When a request comes in:
  - If the receiver is not in that critical region and doesn't want to enter, send

back an 'OK'.
  - If the receiver is in that critical region, don't reply, but queue the request.
  - If the receiver wants to enter that critical region but hasn't done so, it compares the incoming timestamp with the timestamp it sent out; lowest one wins! If the incoming message has the lower timestamp, send back an 'OK'; otherwise queue it.
- When a process exits a critical region, it sends 'OK' messages to all processes on its queue and deletes them from the queue.

Distributed algorithm (3)

- Mutual exclusion guaranteed, without deadlock or starvation.
- $2(n-1)$ messages per critical region entry (n is the total number of processes). (Has been refined to n.)
- No single point of failure.
- Now there are n points of failure!
- Solve the problem with message acknowledgements and timeouts.
- All processes are involved in all decisions. Each process must do the 'same thing'.
- The group of processes may change ...
- Every process is a bottleneck.
- Slower, more complicated, more expensive, and less robust than the centralized algorithm.

Token ring algorithm

- Impose a logical ring on the processes.
- Pass a token around the network.
- If you're not interested in entering a critical region when you get the token, just pass it on.
- If you are, hold on to the token, do your critical stuff, and only pass on the token when you have finished.
- Oops, what happens if the token gets lost? How do you tell if it's really lost, or someone is just holding on to it for a long time?
- Crashes: require an acknowledgement after passing on the token. If a node has crashed, remove it from the logical ring and bypass it.

Election algorithms

- Rationale: many distributed algorithms require one process to be a co-ordinator (or special in some way). How does a system decide on – elect – a co-ordinator during execution?
- Need some way to distinguish processes, e.g. network address (maybe with process ID)
- In general, election algorithms elect the highest process number.
- Assume that each process knows the process number of every other process, but not necessarily which ones are up and which are down at the moment.
- The goal: after an election, all processes agree on who the co-ordinator is.

The Bully Algorithm (1)

- Garcia-Molina (1982)
- When a process P notices that the co-ordinator is down, it initiates an election:
  - P sends an ELECTION message to all processes with higher numbers.
  - If no one responds, P wins and becomes co-ordinator.
  - If a process responds, it takes over, and P's job is done.
- If a process receives an ELECTION message, it sends back an OK message, and then starts an election (unless it is already running one).

The Bully Algorithm (2)

- Eventually, all but one process gives up, and that is the new co-ordinator, and it announces victory to all other processes by sending them a COORDINATOR message.
- When a process that was down comes back up, it starts an election.
- If that process is the highest-numbered, it will win and take over.
- The biggest guy always wins, hence the name 'bully algorithm'.

Motivation: Parallel databases

References: chapter 8 of E. V. Krishnamurthy, *Parallel processing: principles and practice*, Addison-Wesley, 1989; DS 12, Bacon 18.

- A database (DB) is a collection of closely-related data stored in a computer system.

- A user accesses the contents of a DB using a (high-level) **query language**.
- There is a system which provides an interface between the user and the database: the **database management system** (DBMS).

DBMS tasks

- The DBMS is supposed to provide a 'user-friendly' way of specifying operations on the database. It is also supposed to provide:

Security: authorised access only

Integrity: maintain consistency constraints, e.g. not over-filling aircraft

Concurrency control: when there are several concurrent users

Crash protection and recovery: backups and restorations ('roll forward')

Actions

- Operations on the data in databases are called **actions**. The following are typical:

`Delete`**:** delete all records satisfying a query

`Insert`**:** insert a new record

`Update`**:** specify a new value to be taken by the object being modified

`Retrieve`**:** fetch relevant data satisfying a query

- A database can be in one of three states: `open` (ready for processing); `active` (processing); `closed` (finished).
- An action required by a user is called a **request**. A finite sequence of such requests is called a **transaction**.

Transactions (part 1)

Building transactions

- A transaction is a 'meaningful atomic operation' (Bacon), which may or may not be composite.
- Transactions may be allowed at many levels in a system. Some systems allow them to be nested.
- Successful termination is called commitment.
- Undoing the effects of a partially-executed transaction is called aborting.
- If a system allows a user process to abort a transaction, it's possible to use optimistic concurrency control: allow more concurrency than is strictly safe, and undo any problems that arise.

ACID properties

(Not to be confused with LSD properties)

A transaction (often called an atomic transaction) has the following properties:

Atomicity All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it's possible to roll back the system to the state before the transaction was invoked.

Consistency Transforms the system from one consistent state to another.

Isolation Results (including partial results) are not revealed unless and until the transaction commits. If the operation accesses a shared data object, invocation does not interfere with other operations on the same object.

Durability After a commit, results are guaranteed to persist, even after a subsequent system failure. This implies that the results are saved in permanent storage.

'The plan'

- We want several transactions to be executed in parallel.
- Implementing isolation is hard.
- We'll see three solutions – schedulers – later.
- What happens if you don't implement isolation? Loss of data integrity!

Data integrity

- A database is **consistent** if it satisfies a set of explicit logical conditions, called **integrity constraints** (boolean expressions).
- A database must proceed from one consistent state to another. If an action would violate this, it must be rejected, and the original data restored.
- Inconsistency happens via *side effects*. (If only retrieval operations occur, these can't happen.) There are three types:

`ru`**:** user U_i retrieves an object; user U_j updates the same object;

`ur`**:** user U_i updates an object; user U_j retrieves the same object;

`uu`**:** user U_i updates an object; user U_j updates the same object.

Inconsistencies (1)

- There are three different kinds of inconsistencies which arise from one or more of these types of side effects.
- **Lost action/update:** here, U_1 loses its update at time step 3. It is a combination of an `ru` and a `uu` side effect.

Time step	U_1	U_2
0	Retrieve A	...
1	...	Retrieve A
2	Update A	...
3	...	Update A

- To resolve this conflict, object A should be accessed by only one transaction at a time.

Inconsistencies (2)

- **Uncommitted dependency:** U_1's retrieve is no longer valid after the abort.
- This is realistic: U_2 could have crashed, and U_1 may have already terminated. This is a `ur` side effect.

Time step	U_1	U_2
0	...	...
1	...	Update A
2	Retrieve A	...
3	...	Abort

Inconsistencies (3)

- **Inconsistent analysis/retrievals:** e.g. U_1 computes F+C+H, where F=\$100, C=\$200, H=\$300; U_2 transfers \$100 from H to F.

Time step	U_1	U_2
0		
1	Retrieve F (total=\$100)	
2	Retrieve C (total=\$300)	
3		Retrieve H
4		Update H (H←H−\$100)
5		Retrieve F
6		Update F (F←F+\$100)
7		End transaction
8	Retrieve H (total=\$500)	

- Combination of an `ru` side effect on F, and a `ur` side effect on H.

Serializability and consistency (1)

- Ignoring crashes (for the moment), a consistent system state can be maintained by executing transactions serially.
- Any possible serial execution is deemed to be acceptable. If you want two transactions to be executed in a particular order, you must impose this at a higher level (put them together into a nested transaction).
- (If two operations can be performed in either order with the same effect, they are said to be commutative.)
- But within a single transaction, any two operations on the same object are carried out in the order specified in the transaction.

Serializing operations

See Bacon, Figures 18.1–18.3.

- Conflicting operations on one object must be done in order: serialized.
- The components of composite operations must be performed in order, but it's not necessary to serialize the whole lot.
- By delaying a part (or parts) of one or more composite operations (interleaving), it is possible to execute them concurrently, while keeping the ordering and achieving the necessary serialization.

Serializability and consistency (2)

- During concurrent execution, sub-operations are interleaved. What are the chances that two transactions will access the same data?
- If a specific interleaving of sub-operations can be shown to be 'equivalent' to some serial execution, then such a concurrent execution keeps the system in a consistent state. The execution sequence is said to be serializable.
- See examples in DS §12.4, Bacon §18.3.
- The challenge: achieve concurrent execution of transactions, while ensuring that no transaction ever sees an inconsistent system state. **Maintain the *appearance* of isolation!**

Schedules

- When there are several users of a DBMS, the transactions run concurrently, and their individual requests are interleaved. This interleaving is called a **schedule** (history,

log).

- A **serial schedule** is a schedule where there is no interleaving, i.e. the requests of transactions are kept together.
- If a schedule is equivalent to some serial schedule (it produces the same effects), we say it is **serializable**.
- *A schedule is correct if it is equivalent to a serial schedule.*
- (Unfortunately, proving correctness is an NP-complete problem, i.e. it probably takes exponential time).

Some observations

- When shuffled, two transactions do not necessarily produce the same effect that they would produce if they were performed serially.
- Retrieve and update operations should be controlled to avoid inconsistencies.
- There needs to be a scheduler which restricts possible sequences of retrieve-update operations by locking them in or out, yielding a consistent serializable schedule.
- If the scheduler fails, there needs to be a visible phenomenon so that corrective action can be taken.
- Next time: what are the different types of scheduler, and how do they work?

Transactions (part 2)

See DS 12, Bacon §8.6

- Identify abstract data object with set of operations, e.g. bank account: `create`, `delete`, `read-balance`, `check-balance`, `credit`, `debit`.
- Identify non-commutative (conflicting) pairs of operations, e.g.:
- Two `credit` operations commute.
- `credit` and `debit` are commutative.
- `read-balance` and `debit` are not.
- `check-balance` and `credit` are not.

Condition for serializability

- Objects are uniquely identified.
- (Sub-)operations are executed without interference (they are the finest granularity).
- A single clock is associated with each object, indicating the time at which operations take place (and thus their order).
- Each object records the time at which each operation invocation takes place (with the transaction identifier).
- *For serializability of two transactions it is necessary and sufficient for the order of their invocations of all conflicting pairs of operations to be the same for all the objects which are invoked by both transactions.*

Histories and serialization graphs

Bacon §18.7–8

- Histories: see Figures 18.7–9.
- Serializable history: a serializable execution of the transactions: all conflicting pairs of operations on each object are invoked in the same order as in the given history.
- An object is a witness to an order dependency between two transactions if they have invoked a conflicting pair of operations on that object.
- Serialization graph: a directed graph; vertices are transactions; edge $T_i \rightarrow T_j$ iff some object is a witness to that order dependency.
- A transaction history is serializable iff its serialization graph is acyclic!

Is a schedule serializable?

- We need to find a total ordering of the set of transactions that's consistent with the schedule:
  - Each object knows which pairs of operations conflict.
  - Each object knows which transactions have invoked conflicting operations: it's a witness to an order dependency between them.
- If the order dependencies are consistent, then you get an ordering for each pair of transactions. If not, then there's a cycle in the serialization graph.
- Finding a total ordering of the set of transactions: use a topological sort. You can do it iff the graph is acyclic.

At run time . . .

- Maintain a serialization graph of the transactions currently in progress.
- When a new transaction is submitted, cre-

ate a 'proposed schedule' by adding the operations of the new transaction to the existing serialization graph.
- Do it so that you don't end up with a cycle.

Transaction schedulers

DS 13, Bacon 19.

Three main types:

Locking: granularity: locking at the level of objects, predicates, or structures.

Timestamp: a serialisation order is selected before the transactions are processed. Transaction execution is forced to obey this order. Each transaction is assigned a timestamp, and when conflict arises, timestamp order is used.

Optimistic: assume conflict is unlikely, and patch things up if they go wrong.

Locking

- Every object in the database can be locked.
- Warning: once we have locks, we have the possibility of deadlock: we need a lock manager to to deadlock detection or avoidance.
- A transaction locks each object some time before it needs it, and unlocks it some time after it has finished with it.
- A possible approach: lock all the required objects at the beginning of a transaction, and unlock them all at commit or abort.
- Is it possible to achieve better concurrency?

Two-phase locking (2PL)

- Each transaction has a growing phase: locks can be acquired, but not released.
- Then, each transaction has a shrinking phase: locks can be released, but not acquired.
- Strict 2PL: release all locks on commit. (Avoids cascading aborts.)
- 2PL guarantees that conflicting pairs of operations of two transactions are scheduled in the same order: you get a serializable schedule.
- Deadlock is possible.

Semantic locking

DS §13.1–2, Krishnamurthy 8

- Distinguish two types of locks: S (read, shared) and X (write, exclusive).
- The idea: grab an S lock if you only need to read the object. Any number of transactions can get an S lock on an object.
- If you decide you want to update the object, try to 'upgrade' to an X lock.
- An X lock is granted only after all other transactions release any S or X locks on the object.

Dealing with deadlock

- In a scheduler which supports locks, we have (1) exclusive allocation (2) resource hold while waiting (3) no preemption. So we can get deadlock.
- Does the system already support aborting transactions? If so, deadlock detection followed by aborting the deadlocked transactions is a good design option.
- Alternatively, put timeouts on lock requests. Abort any transaction that has a timeout on one of its lock requests.

Time-stamp ordering (TSO)

- Put a time-stamp on a transaction when it starts.
- Recall that to get serializability we have to keep the ordering of conflicting operations consistent between two transactions.
- Use the time-stamp to determine which transaction should be the first to execute a conflicting operation.
- Suppose a transaction invokes an operation. Suppose a second transaction attempts to invoke a conflicting operation. Compare time-stamps.
- If the time-stamp of the second transaction is later than that of the first, then the operation can go ahead.
- Otherwise, the second transaction is 'too late' and must be aborted.

TSO properties

- As described, doesn't enforce isolation; may cause cascading aborts.
- Can be fixed: require the earlier (by timestamp of the transactions) of a pair of conflicting operations be committed before proceeding. See Bacon §19.5.1.
- Doesn't deadlock: there are no locks!
- Starvation? Unlikely in the sorts of scenar-

ios where time-stamp ordering is used.

Optimistic concurrency control

- The premise: conflict is unlikely, i.e. it's unlikely for two transactions to be operating on the same objects.
- Ensure a serializable execution, but achieve high availability of objects: minimize delay at transaction start.
- Don't change persistent memory: work on shadow copies of objects.
- Each shadow copy has a version: the ID of the last transaction to update the object.
- On commit, validate the history to see if it can be serialized with the transactions that have been accepted since it started.

Phases for each transaction

1. Execution (read): execute the transaction to completion on the shadow copies.
2. Validation: after commit, check the execution schedule to ensure serializability.
3. Update (write): update all changed objects in order, transaction by transaction.

- Each transaction should interact with a consistent set of shadow copies: apply updates atomically across all objects involved in the transaction.

Staying optimistic

- It's possible to use sophisticated protocols to keep everything consistent, e.g. two-phase commit in a distributed TPS (see DS §14.3, Bacon 21).
- Atomic commitment is bad for concurrency, and ...
- ... it misses the point: it's supposed to be optimistic!
- So take a shadow copy of an object only when needed. This is more likely to lead to rejection at validation time. So what?

The validator

- The validator knows all about the transactions whose validation or update phases overlap. It must ensure there has been no conflict.
- It checks: (1) the execution must be based on a consistent system state; and (2) the transactions must be serializable.
- If the validator says OK, the transaction goes into a queue, waiting to be written.
- Note: here, transaction order is determined at validation time: in theory, the validator may insert at any place in the queue! Cf. time-stamp ordering.

Properties of OCC

- No deadlock.
- Maximally parallel.
- Very efficient if the transactions are small and widely spaced.
- Becomes inefficient if transactions tend to overlap.
- Starvation if the premise (of minimal conflict) is violated.

More on Transactions

(This lecture material is by Vijay Boyapati.)

- References for today:
  - Transaction Processing, J. Gray & A. Reuter
  - Distributed Databases, S. Ceri & G. Pelegatti
  - DS 14

Transactions

- Flat
- Nested
- Distributed

Definition [Ceri & Pelegatti]:
A transaction is an atomic unit of database access, which is either completely executed or not executed at all.

Flat Transactions

- A flat transaction, **FT**, is an operation, performed on a database, which may consist of several simple actions.
- From the client's point of view the operation must be executed indivisibly (atomically).
- e.g. Withdrawing $20 dollars from your account:

```
CheckClientPassWordValid();
openClientAccount();
removeAmount(amount);
sendCashToATM(amount);
```

- Main disadvantage with **FT**s: If one action fails the whole transaction must abort.

Nested Transactions (1)

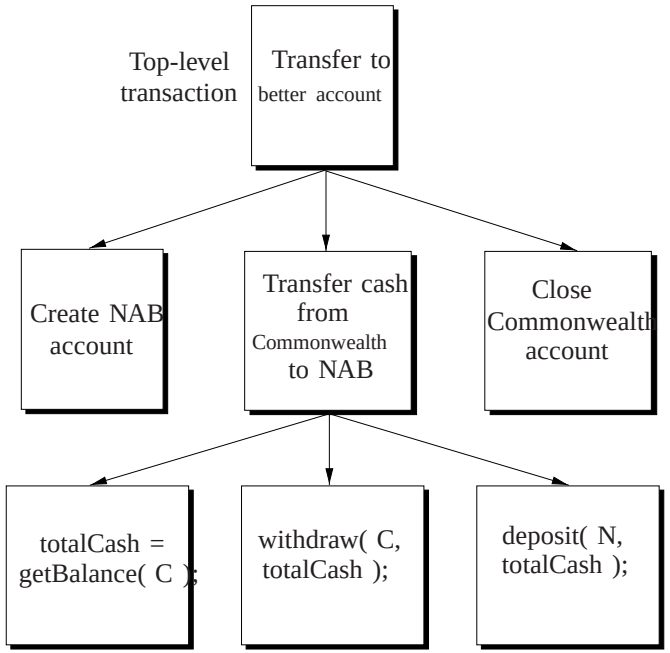
Used to overcome the disadvantages of flat transactions.

- A nested transaction, **NT**, is a tree of transactions.
- Transactions at the leaves are **FT**s.
- The root of the tree is the top level transaction.
- Other nodes are called subtransactions.
- The terms parent and child transaction are used in the obvious way.
- A subtransaction can commit (its job is done) or roll back (it may have crashed).

Nested Transactions (2)

e.g. Transferring to a better bank account:

- Create NAB account.
- Transfer money from Commonwealth account to NAB account.
- Close Commonwealth account.



Nested Transactions (3)

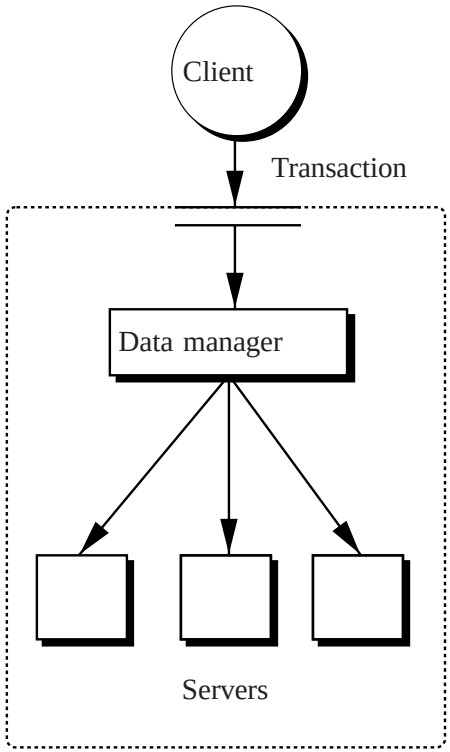
- A subtransaction may abort and roll back, which causes its descendents to abort and roll back – even if they have committed.
- Only the descendents are required to roll back!
  - This is the key difference between FTs and NTs.
  - With NTs, the subparts might not 'bring down' the whole transaction.

The first point is important (why?): Subtransactions of a nested transaction, while still having the A, C and I properties, should not have durability (D).

Distributed Transactions (1)

Definition [J. Gray & A. Reuter]:
A distributed transaction is a transaction that runs in a distributed environment and has to use several (>1) servers in the network, depending on where the data is.

- The distribution of a transaction should be transparent to the client that issued it.
- Distribution is the concern of the data manager.
- Don't confuse distributed and nested transactions:
  - 'Distributedness' relates to where the data is held (low level concern).
  - 'Nestedness' relates to how the client decomposes a transaction (high level concern).



ACID properties of Distributed Transactions

- Scenario: We have several servers working on one transaction (because data required in the transaction is distributed around a network).
- Problem: How do we make this transaction execute indivisibly?
- Solution: Use two-phase commit (2PC) protocol.

2PC protocol (1)

- Initialization
  - The first server contacted is deemed to be the coordinator.
  - Other servers used in the transaction, known as workers, are told the name of the coordinator.
  - Each server in the transaction tells the coordinator its name.
- Phase 1
  - Coordinator asks workers whether they can commit to their part of the transaction.
  - Workers send their responses to the coordinator.

  IMPORTANT: If a worker says it can commit then it must eventually be able to do that (even in the event of a crash).

2PC protocol (2)

- Phase 2
  - Coordinator checks responses. If everyone (including coordinator) can commit, coordinator sends a message to all workers telling them to commit. Otherwise coordinator tells everyone to abort and sends an abort message to the client (sorry, couldn't do the transaction).
  - If workers were asked to commit, they send a done message to the coordinator after completing their job.
  - Once the coordinator receives all the done messages it removes all transaction state information and sends a done message to client.

2PC performance

- In the event of a crash, 2PC can waste lots of processing time!
  At which stage?
- No bound possible on total time taken.

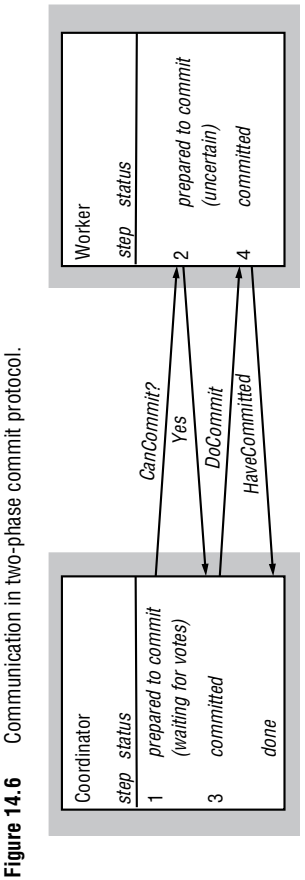


Figure 14.6 Communication in two-phase commit protocol.

Distributed Transactions (2)

- 2PC gives us atomicity.
- There are other concerns when dealing with distributed transactions:
  - Maximising concurrency (for performance)
  - Concurrency control (e.g. using locks)
  - Distributed deadlock detection
  - Data replication

Replication

- In large distributed systems most data objects will have replicas.
- Why replicate?
  - Performance (e.g. local cache)
  - Fault transparency
  - rm *
- Problem: maintaining consistency.
- We sometimes want (and often require) one-copy serializability: Transactions performed by various clients on replicated data items appear the same as transactions performed one at a time on single data items.
- Use replica managers.

Replica managers

- A replica manager (RM) is a process created to manage a replica of a data object.
- If the RM is told to perform an operation on its replica it must communicate with all the other replica managers, informing them of the operation.
- Communicating after the operation is complete would be preferable (it would reduce communication in the event of a failure).
- This is not possible (why?)
- One-copy serializability!

The Internet

- Functionally:
  - a big heterogeneous computer network;
  - the world's largest distributed system.
- Operationally: a community of users, sharing
  - bandwidth (network connections);
  - storage capacity;
  - computing power;
  - information;
  - services

How it developed (1)

1960s RAND Corp (Cold War think-tank): How could US authorities communicate after a nuclear war?

1964 RAND proposal: network with no central authority, capable of operating while in tatters. Network assumed to be *unreliable at all times*. All network nodes have equal status.

1968 National Physical Laboratory (UK) develops first such test network.

1969 ARPANET developed by US Defence Advanced Research Projects Agency (DARPA). First node installed at UCLA.

1972 37 nodes on ARPANET. Intended use was shared facilities, but main practical use was e-mail and news.

1973 First international nodes on ARPANET.

1977 TCP/IP in common use on ARPANET. Connections to ARPANET remain tightly controlled.

How it developed (2)

1983 Military break off, to become MILNET.

1984 National Science Foundation begins NSFnet. Constant bandwidth upgrades. Growing number of users.

1988 Internet Worm released, disabling most of the Internet.

1989 End of ARPANET. NSFnet takes up the slack.

1991 WAIS and gopher software released.

1992 World Wide Web (text only).

1993 Mosaic web browser produced: graphics for the WWW. Internet explosion! WWW proliferates at a rate of 341 634% of service traffic. Gopher grows at a rate of 997%. Development of benign 'Internet worms' to find web pages, and index them.

1995 NSFnet retires as a backbone service. Many other backbone services now

Who owns the Internet?

- Originally, DARPA provided guidelines for use.
- Later, NSFnet (because they owned the only backbone).
- Now, nobody owns it!
- Analogy with the English language:
  - Nobody owns it.
  - It evolves according to need.
  - It's up to you to learn how to use it properly.
  - No 'English, Inc.'. No board of directors.
  - Groups (governments and companies) may teach how to use it.

What people do on the Internet

- Electronic mail
- Discussion groups (Usenet, IRC, MUDs)
- Long-distance computing
- File transfers
- Banking
- Shopping
- Finding information (weather, stock exchange, anything!)

E-mail

- Faster than paper mail (snail mail)
- Cheaper than paper mail (often free)
- Send text, software, images, sounds, movies.
- Subscriptions to news, journals, interest groups.
- Personal, professional communications.
- Broadcast communications.

Discussion groups

- Newsgroups, IRC, MUDs, mailing lists
- Real-time responses or batch replies
- Discussion, gossip, commentary, conferencing, requests for information, advice, counselling, collaborations (fiction and non-fiction), finding a spouse
- Moderated or unmoderated
- Text, audio, visual (e.g. CU-SeeMe)

Long-distance computing

- Remote logins to high-performance computers
- Remote catalogue searches
- Co-ordinated multicomputing environments, e.g. PVM, MPI, autoson (local product!), etc.
- Factorization of numbers which are a product of large primes

File transfers

- WWW, FTP
- Archie, gopher, WAIS
- Remote publishing

- Text, audio, visual: still frames, or movies
- Browsing, searching (LEO)

Some risks

- Monitoring your Internet use: compiling information about you!
- Abuse of privilege: publication of private information.
- Misleading or incorrect information.
- Fraudulent representation.
- Attacks on your hardware/software/data: hacking, denial of service attacks, ...
- Morally objectionable content: offensive, inciteful, obscene, degrading, dangerous, ...
- System failure, misbehaviour, and other malfunctions

Dealing with the risks

Prevention

- Reducing the opportunities, e.g. limited site access, user authentication schemes, strong data encryption, tighter management, greater accountability

Deterrence

- Making it undesirable or difficult to place someone at risk, e.g. penalties (fines, denial of access, imprisonment), obstructions (data encryption, obscurity)

Detection

- Finding out what happened as soon as possible
- Limiting/correcting the damage

Insurance

- Getting compensation for loss, injury, etc.

Censorship

What may be censored? Content that is thought:

- seditious (inciting people against the government)
- subversive (overthrowing something established)
- to incite racial or religious hatred
- to infringe religious or cultural norms
- obscene
- defamatory
- contemptuous of court or parliament
- to breach copyright, the Official Secrets Act, etc.

Censorship in Australia

- Office of Film and Literature Classification classifies:
  - publications
  - films and videos
  - computer games
- Censorship classifications:
  - unrestricted (e.g. G-rated movies)
  - restricted (e.g. R ratings)
  - refused classification (i.e. banned)
- Censorship in Australia:
  - hard-core pornography (especially involving children, animals, violence, cruelty, exploitation)
  - incitement to or instruction in crime (especially violence or illegal drugs)
  - It's an offence to possess Refused Classification material in Australia, and an offence to import, sell, or hire such material.

Censorship and the Internet

- Is it 'importing' to download material from the Internet?
- Is is 'possessing' to have the material on your hard disk? In your account? In a public area? In a private area?
- What if the content is not known to you?
- What if it's encrypted, or stored invisibly in some bigger file?
- What if it's on a client machine? What if it's on a server?
- What if the computer stores it in (volatile) memory, prior to sending it somewhere else?
- What if the material is uploaded from a machine outside Australia?
- What if it's requested from a machine outside Australia?
- What if such content is placed on your screen without your consent?
- State versus Federal enforcement.

Self-censorship on the Internet

- Moderated newsgroups
- E-mail/news 'kill' files
- 'Net nannies': web filters that block certain sites, pages with keywords, etc.
- 'Trusted sites': server sites that uphold your personal values

- Delegated censorship: ask someone else to censor for you
- Personal censorship: do it yourself

Who is responsible?

- Infrastructure providers (e.g. Telstra)
- Service providers (e.g. Ozemail)
- Content providers (e.g. the person who owns the web page)
- Users (people who are browsing, downloading, reading)

Object-Oriented Middleware

What are we talking about?

- CORBA: Common Object Request Broker Architecture, by OMG: Object Management Group (`http://www.omg.org/`)
- DCOM: Distributed Component Object Model, by Microsoft
- RMI: Remote Method Invocation, by Sun (been there, done that)

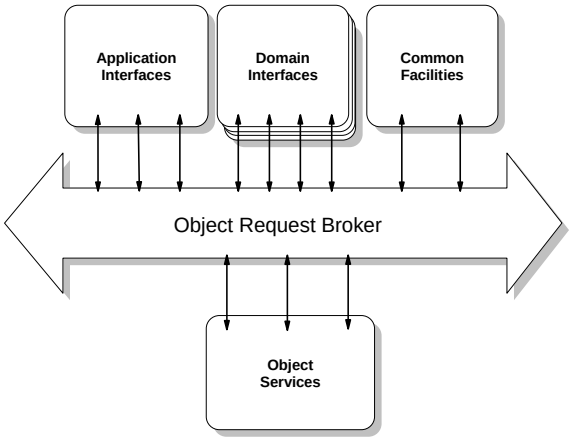
CORBA references: Steve Vinoski (`http://www.iona.com/hyplan/vinoski/`).

Object Management Architecture (OMA)

'The OMA is composed of an *Object Model* and a *Reference Model*. The Object Model defines how objects distributed across a heterogeneous environment can be described, while the Reference Model characterizes interactions between those objects.'

'In the OMA Object Model, an object is an encapsulated entity with a distinct immutable identity whose services can be accessed only through well-defined *interfaces*. *Clients* issue requests to objects to perform services on their behalf. The implementation and location of each object are hidden from the requesting client.'

OMA Reference Model Interface Categories



OMA components

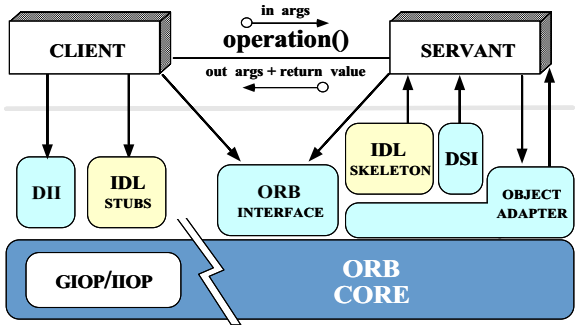
Object Request Broker (ORB): 'mainly responsible for facilitating communication between clients and objects.' It uses:

1. Object Services: domain-independent interfaces used by many distributed object programs, e.g. Naming Service and Trading Service.
2. Common Facilities: like above, but more end-user oriented, e.g. Distributed Document Component Facility (DDCF).
3. Domain Interfaces: like above, but application-domain oriented, e.g. telecommunications, medical, financial.
4. Application Interfaces: developed for a given application.

CORBA Components

- ORB Core
- OMG Interface Definition Language (IDL)
- Language Mappings
- Interface Repository
- Stubs and Skeletons
- Dynamic Invocation and Dispatch
- Object Adapters
- Inter-ORB Protocols

How CORBA fits together



ORB Core

Key feature of the ORB: transparency! The ORB hides object:

- location
- implementation
- execution state
- communication mechanisms

Clients get object references by using:

- object creation

- directory service (e.g. Naming and Trading)
- convert to string and back

The idea: 'keep the ORB as simple as possible, and push as much functionality as possible to other OMA components'.

OMG IDL

- Like Sun RPC IDL
- Looks like C/C++
- Built-in types like C, also string, sequence, fixed, typedef
- Only define types and functions, not data
- Multiple inheritance of interfaces

```
interface Fortune
{
  exception IndexOutOfBounds
  {
    short number;
    string fortune;
    string errorMsg;
  };
  string getFortune(in short number)
    raises (IndexOutOfBounds);
  oneway void setFortune(in short
    number, in string fortune);
};
```

Language mappings

- The current (2.2) spec gives mappings for C, C++, Smalltalk, COBOL, Ada, and Java.
- Others (e.g. Bourne shell, Perl, Eiffel, Modula-3) have been proposed (and are being used)
- The mappings for some languages are cleaner than others (guess which).
- Java seems to have been designed with CORBA in mind.

Interface repository (IR)

- 'Normally' you have to know about all the interfaces you're going to be using at *compile time*.
- The IR 'allows the OMG IDL type system to be accessed and written programmatically at runtime'.
- Basically, it's like reflection in Java.
- You can find out about the interfaces that are 'out there' – including which types and functions they define – and then invoke those functions (with DII).
- If you get hold of some CORBA object, you can track down its interface(s), invoke methods, etc.

Stubs and skeletons

- 'A stub is a mechanism that effectively creates and issues requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation.'
- 'The stub essentially is a stand-in within the local process for the actual (possibly remote) target object.'
- 'The stub works directly with the client ORB to *marshal* the request.'
- 'The server ORB and the skeleton cooperate to *unmarshal* the request.'
- Etc. etc., as for RMI

Dynamic invocation and dispatch

- Dynamic Invocation Interface (DII): 'supports dynamic client request invocation'
- Dynamic Skeleton Interface (DSI): 'provides dynamic dispatch to objects'

View them 'as a *generic stub* and *generic skeleton*, respectively'.

DII supports Synchronous (client blocks), Deferred Synchronous (client blocks when answer needed), and Oneway ('fire and forget') Invocation.

Object adapters

- 'The glue between CORBA object implementations and the ORB itself.'
- Object registration: turn programming language entities into CORBA objects
- Object reference generation
- Server process activation
- Object activation
- Request demultiplexing: handle multiple connections
- Object upcalls: dispatch requests to registered objects

Inter-ORB protocols

- Multiple ORBs need to talk to each other
- General Inter-ORB Protocol (GIOP): 'specifies transfer syntax and a standard set of message formats for ORB interoperation over any connection-oriented transport. GIOP is designed to be simple and easy to implement while still

allowing for reasonable scalability and performance.'
- Internet Inter-ORB Protocol (IIOP): 'specifies how GIOP is built over TCP/IP transports.'
- Support for GIOP and IIOP is mandatory; an implementation may provide others.

DCOM

- CORBA is a *specification*; DCOM is an *implementation* (not of CORBA) (by Microsoft).
- Similar aims and objectives.
- IDL based on Open System Foundation's Distributed Computing Environment (DCE) IDL (see Tanenbaum): very complicated!
- You guessed it: supports C++, C, VB, VJ++.
- Built into NT; available for 95/98; Solaris version.

So what?

- Another layer in your software: somewhere in the *middle*.
- Lots of good implementations of CORBA out there, and everyone is talking about it.
- Has anyone built anything useful with it? Maybe.
- CORBA and RMI: Sun says use RMI within a group of Java servers; use CORBA to talk to 'the outside world'.
- CORBA spec talks about CORBA/COM interoperability: recognition of the inevitable.

Where we've been (1)

- Evolution of IPC primitives (via shared or distributed memory)
- Choosing primitives:
  - synchronous or asynchronous
  - process identification
  - data flow
  - process creation
- Message passing:
  - technical details
  - options
  - broadcast & multicast

Where we've been (2)

- Case studies: Ada (rendezvous), occam (channels), Linda (tuple space)
- Futures
- BSD Sockets and System V streams
- Connection-oriented and connectionless protocols
- Iterative and concurrent servers
- Modelling crashes
- Crash resilience
- Idempotent and atomic operations

Where we've been (3)

- Stable and persistent storage; logging and shadowing
- NVRAM versus disks
- Distributed IPC
- Global logical time (Lamport's algorithm)
- Client/server
- RPC
- Handling lost messages

Where we've been (4)

- Correctness: safety and liveness properties
- Fairness
- Deadlock and livelock
- Deadlock detection, prevention, and avoidance
- Distributed mutual exclusion: centralized and distributed algorithms
- Election algorithms

Where we've been (5)

- Transaction processing systems
- Schedules; serializability and consistency
- ACID properties
- Parallel databases
- Actions and effects; inconsistencies
- Histories and serialization graphs
- Transaction schedulers

Where we've been (6)

- Types of locking: 2PL, semantic locking
- Time-stamp ordering
- OCC
- Types of transactions: flat, nested, distributed; replication of data
- Java
- RMI
- The Internet and censorship
- O-O middleware: CORBA, DCOM, RMI

- In the labs: MPI, semaphores, dining philosophers, worker farming, bully algorithm, Java