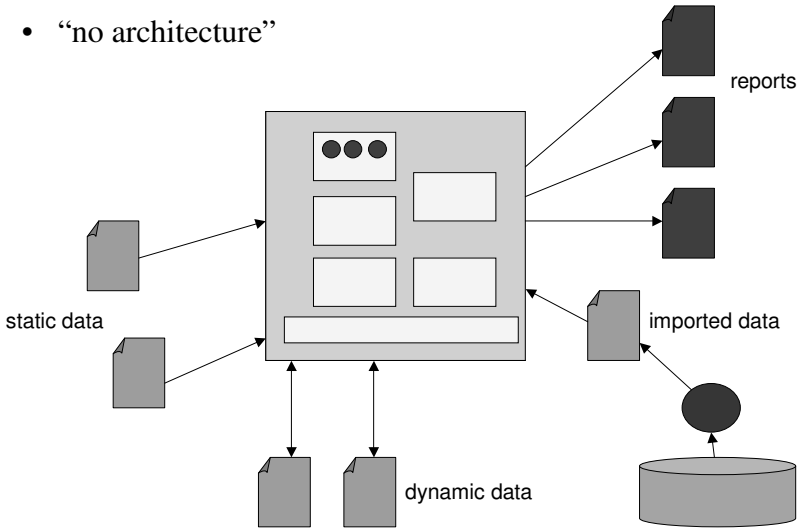
Systems Architecture

Monolithic Systems

Monolithic Systems

- "no architecture"



Examples

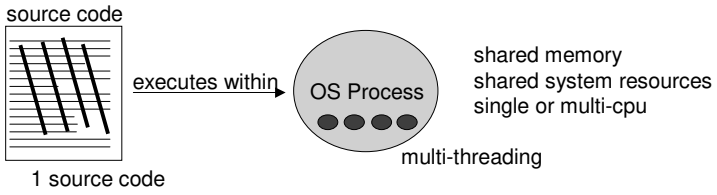
- Most programs you deal with day-to-day
  - word processing
  - spreadsheets
  - powerpoint
  - e-mail (?)
  - inexpensive accounting packages
  - development environments
  - compilers
  - most games
    - (not *Combat Flight Simulator*)
- Large, corporate batch systems
  - payroll
  - reports
  - Descartes route planning

Characteristics

- Usually written in a single programming language.
- Everything compiled and linked into a single (monolithic) application
  - as large as 1 MLOC C++
  - as large as 100M loaded executable
  - as large as 2G virtual memory
- May operate in both
  - batch mode
  - GUI mode
- Data
  - load into memory
  - write *all* back on explicit save
  - ➢ No simultaneous data sharing
- May have concurrency
  - multi-threading
  - multi-processing (but only one executable)

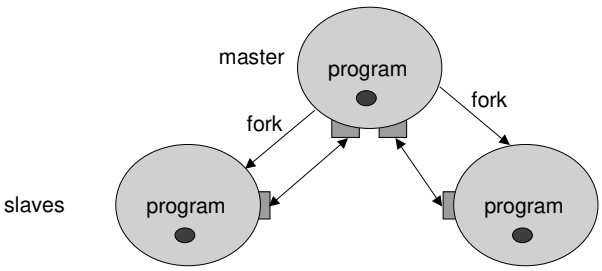
Concurrency

- multi-threading



Concurrency

- symmetric multi-processing

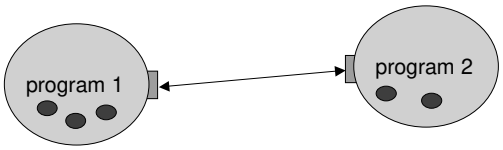


precise copies except for fork() return value

1 source code

Concurrency

- distributed processing



many source codes

Concurrency

- Why multi-threading?
  - performance (when you have access to multiple CPUs)
  - A design philosophy for dealing with asynchronous events
    - interrupts
    - GUI events
    - communications events
  - Maintain liveness
    - can continue to interact with user despite time-consuming operations
      - e.g., SMIT "running man"
  - performance
    - pre-load, network initializations
  - multi-tasking (lets the user do many tasks at once)
    - e.g., downloads from the net
- You WILL have to multi-thread your program
  - start early in the design process

Concurrency

- Why symmetric multi-processing?
  - you need parallelism
    - performance
    - liveness
    - …
  - a program is not written to be multi-threaded
    - temporarily modifying shared data
  - fork cost is inexpensive relative to amount of work to be done by slaves
    - fork typically implemented with COW
- Tricks:
  - special allocators to group modifiable shared data to contiguous memory
- Using memory management hardware to switch volatile data based on "thread"

Monolithic Architecture

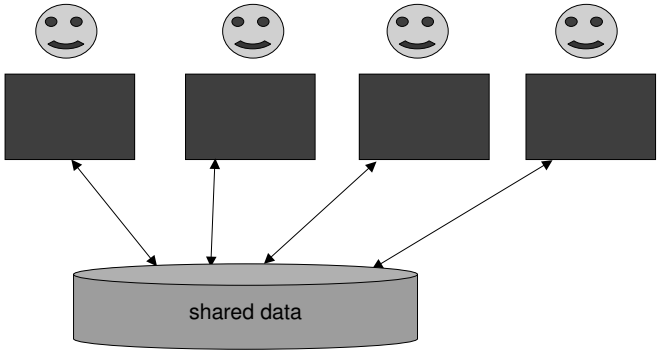
- A monolithic system is therefore characterized by
  - 1 source code
  - 1 program generated
  - but… may contain concurrency

Data

- In a monolithic architecture
  - data is read into application memory
  - data is manipulated
  - reports may be output
  - data may be saved back to the same source or different
- Multi-user access is not possible

Multi-User Access

- Can changes by one user be seen by another user?
  - not if each copy of the application reads the data into memory
  - only sequential access is possible



Multi-User Access

- Allowing multiple users to access and update volatile data simultaneously is difficult.
- Big extra cost
  - require relational database expertise
- More on this later.

Advantages

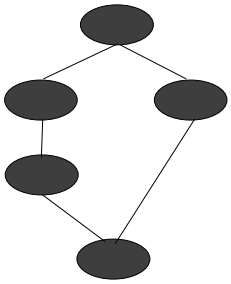
- performance
  - accessing all data
    - disk is disk!
    - either
      - read data more directly from the disk via file system
        - » highly optimized
        - » caching and pre-fetching built-in
      - read data less directly from the disk via layers of intervening software (e.g., RDBMS, OODBMS, distributed data server).
  - modifying data
    - in-memory is massively quicker
    - caching is not an option for shared data systems
      - delays while committing changes to a record
  - No IPC overhead
- simplicity
  - less code to write
  - fewer issues to deal with
    - locking, transactions, integrity, performance, geographic distribution

Disadvantages

- Lack of support for shared access
  - forces one-at-a-time access
  - mitigate:
    - allowing datasets that merge multiple files
    - hybrid approach
      - complex monolithic analysis software
      - simple data client/server update software
- Quantity of data
  - when quantity of data is too large to load into memory
    - too much time to load
    - too much virtual memory used
  - Depending on which is possible
    - sequential access (lock db or shadow db)
    - selective access

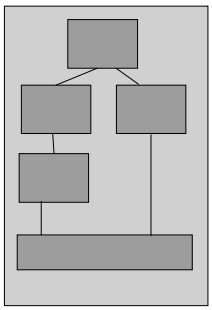
Red Herring

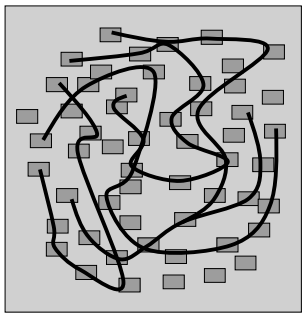
- Monolithic systems are “less modular”



Red Herring

- The code for distributed systems will need to share common objects.
  - This "module" organization could be terrible.





Red Herring (sort of)

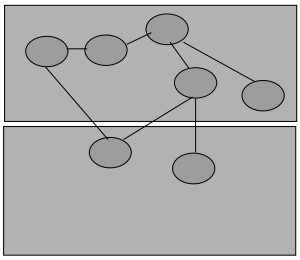
- Distributed systems *require* architects to define and maintain interfaces between components
  - cannot do anything without this
  - even for RDBMS systems
    - relational schema + stored procedures define an important interface
  - by default: nothing is visible
    - must work to expose interface
- For monolithic systems, this is "optional"
  - because there are no process boundaries, any tiny component can depend on (use, invoke, instantiate) any other in the entire monolithic system. *e.g.,*

```
extern void a_routine_I_should_not_call(int a, int b);
```

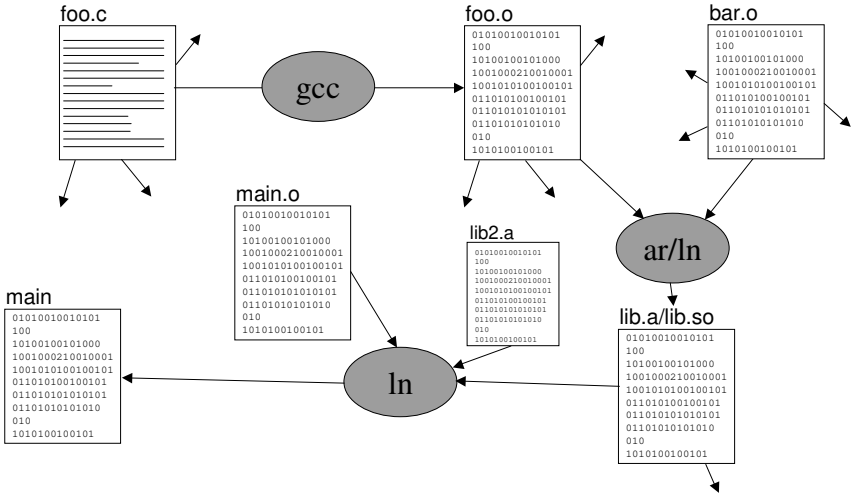
  - default: everything is visible
    - must work to hide non-interface

Module Structure

- To preserve the architectural integrity of a monolithic system, we must work to define and maintain (typically) extra-linguistic sub-system boundaries.
  - recall façade pattern



Library Structure



Library Structure in C/C++

- Decide
  - how many libraries to have
  - their names
  - which subsystems go into which libraries
    - wise to align library structure with a subsystem
    - not necessary to do so
      - e.g., could be a base level of utilities that rarely change whose TU's belong to unrelated subsystems (stretching it).
    - rationale
- Why?
  - reduce compilation dependencies
    - can be changing a bunch of .c's and .h's and others can keep using the library
    - but… don't change any.h's exported beyond the library
    - "poor man's" configuration management system
      - often most practical
  - Reduces link time (libraries often pre-linked)
  - Shipping libraries
    - Common library supports many apps