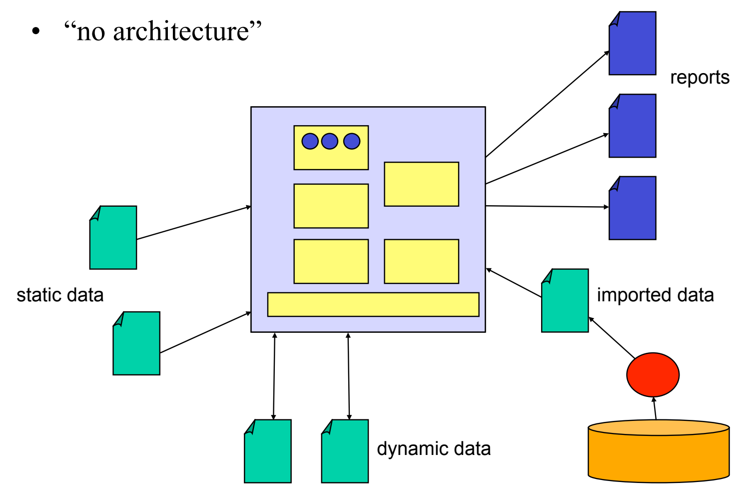
# Systems Architecture

## 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
  - many games
- Large, corporate batch systems
  - payroll
  - reports
  - astounding number of very large mainframe COBOL programs

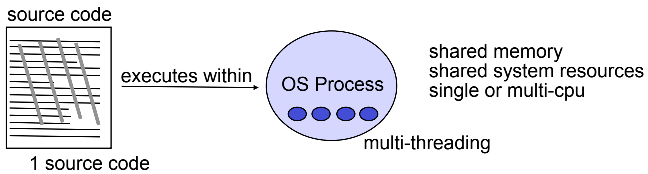


# Characteristics

- 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

- multiple threads share address space.
- Java includes built in threads.
- Windows NT supports threads well.
- “modern” programming model.

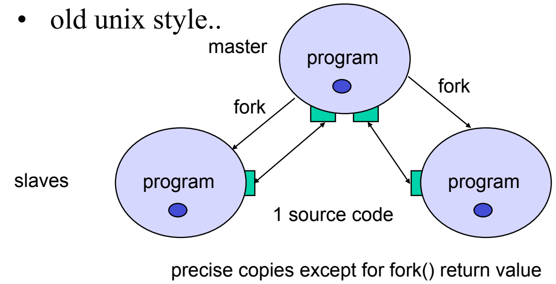



Threads share address space hence sharing data is fast but requires sophisticated synchronization.



# Concurrency

- symmetric multi-processing
- newly forked processes get copy of parents address space
- old unix style..

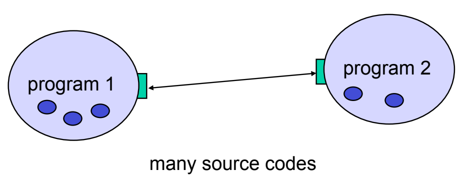



forked processes start out with a copy of parent’s address space. Sharing harder and more coarse grained hence fewer synchronization issues.



# Concurrency

- distributed processing
- different programs cooperating.






# Concurrency

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

## Concurrency

- Why symmetric multi-processing?
  - you need parallelism
    - throughput
    - interactivity..
  - a program is not written to be multi-threaded
    - many unix systems lacked good thread implementations until recently
  - modern fork implementations good, hence cost may be inexpensive relative to amount of work to be done by slaves.
  - Course grained parallelism.
- Many (unix) system mechanisms support communication between processes:
  - signals, pipes, named pipes, shared memory regions, message queues, etc.
  - Mostly outside programming language purview.



## Monolithic Architecture

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



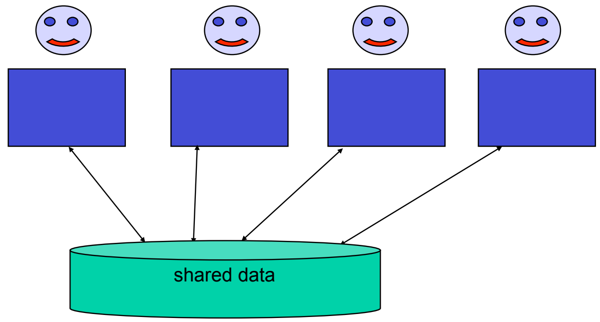
## Data

- 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 or other heavyweight infrastructure.
- More on this later.



## Advantages of Monolithic Systems

- Performance
  - Reading and writing of data can be optimized for performance without regard to issues such as multi-user data sharing.
  - read data directly from the disk via file system
  - read data less directly from the disk via layers of intervening software (e.g., RDBMS, OODBMS, distributed data server).
  - modifying data needn’t worry about writers in other address spaces.
    - in-memory is massively quicker
    - caching would present many subtle issues for shared data systems
  - No IPC overhead
- Simplicity
  - less code to write
  - fewer issues to deal with
    - locking, transactions, integrity, performance, geographic distribution



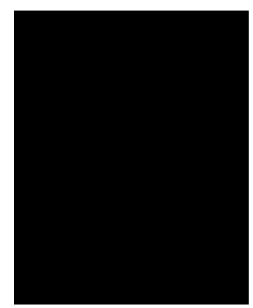
## Disadvantages of monolithic systems

- Lack of support for shared access
  - forces one-at-a-time access
  - mitigate:
    - allowing datasets that merge multiple files
    - hybrid approaches
      - 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



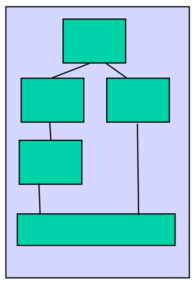
## Red Herring

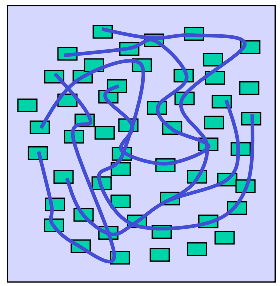
- Monolithic systems are “less modular” ??
- monolithic exterior obscures potential modularity of isolated layers or other software structure.



# Red Herring

- The code for distributed systems will need to share common objects.
  - The fact that the system has been sliced into distributed programs doesn't mean that modules are nicely decoupled.







# Red Herring (sort of)

- Distributed systems *require* architects to define and maintain interfaces between components
  - stub generator need to know the distributed interface.
  - overmuch coupling shows up as performance problem.
  - 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.,*

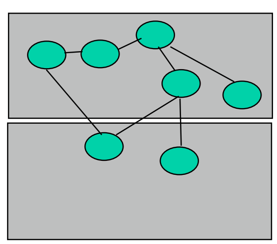
```
extern void a_routine_I_should_not_call(int a, int b);
```

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



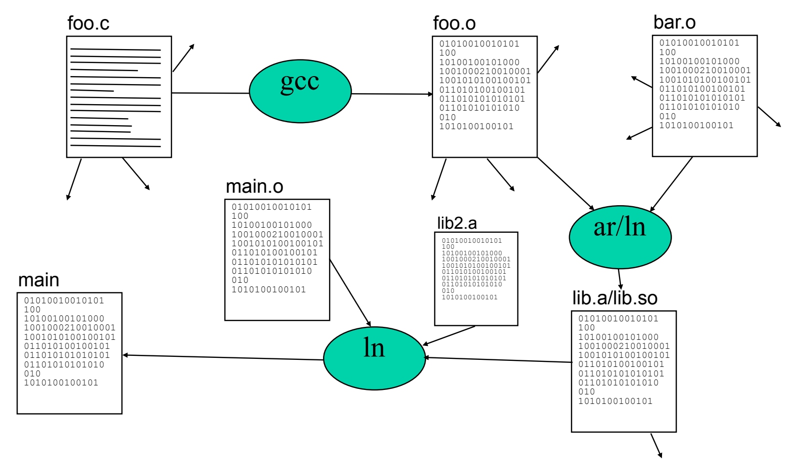
# Module Structure

- 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 (unix) cl/lib/link for windows



# 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 or layer
    - not necessary to do so
      - I've seen libs organized by alphabetic split of objects.
    - 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
- Hopefully libraries are reusable.