Shareable Executables

- Often multiple programs share some code
  - E.g., widely used libraries
- Do we need to load a different copy for each process?
  - Not if all they're doing is executing the code
- OS can load one copy and make it available to all processes that need it
  - Obviously not in a writeable domain

Some Caveats

- Code must be relocated to specific addresses
  - All processes must use shared code at the same address
- Only the code segments are sharable
  - Each process requires its own copy of writable data
    - Which may be associated with the shared code
  - Data must be loaded into each process at start time

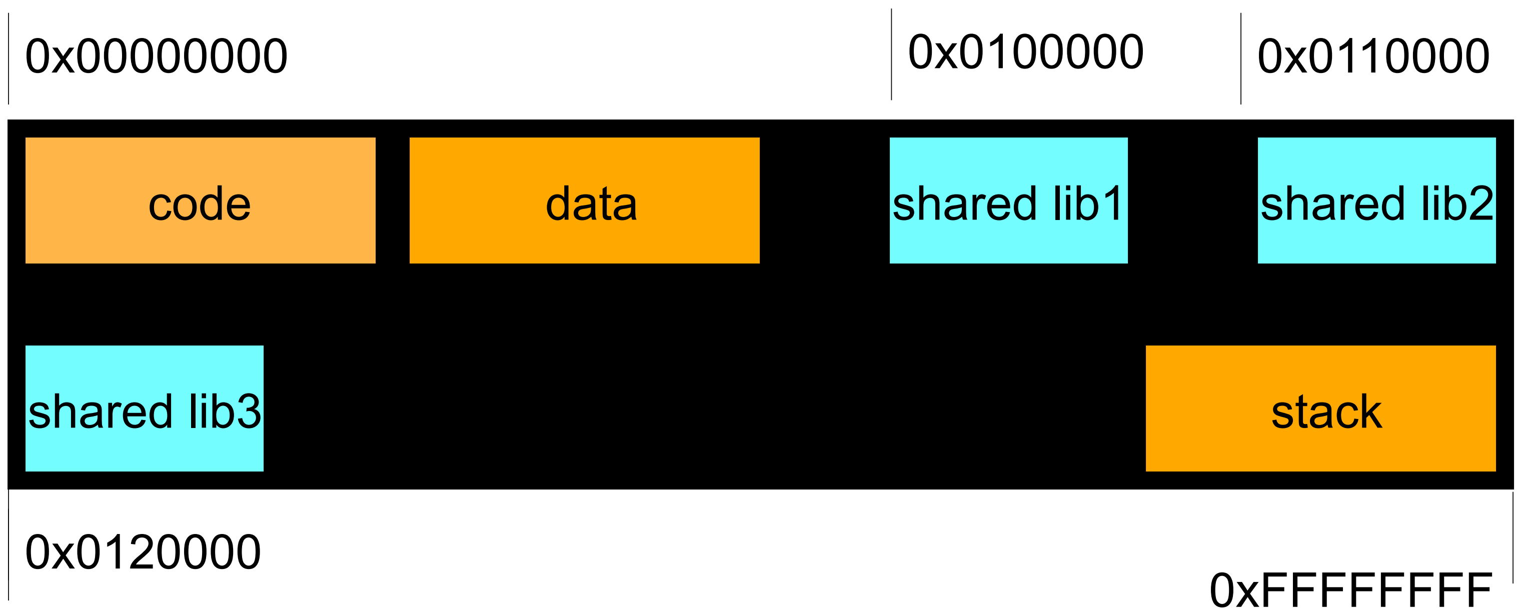
Shared Libraries

- Commonly used pieces of code
  - Like I/O routines or arithmetic functions
- Some obvious advantages:
  - Reduced memory consumption
  - Faster program start-ups, since library is often already in memory
  - Simplified updates
    - All programs using it updated by just updating the library

Limitations of Shared Libraries

- Not all modules will work in a shared library
  - They cannot define/include static data storage
- They are read into program memory
  - Whether they are actually needed or not
- Called routines must be known at compile-time
  - Only fetching the code is delayed until run-time
- Dynamically loaded libraries solve some of these problems

Layout With Shared Libraries



Dynamically Loadable Libraries

- DLLs
- Libraries that are not loaded when a process starts
- Only made available to process if it uses them
  - No space/load time expended if not used
- So action must be taken if a process does request a DLL routine
- Essentially, need to make it look like the library was there all along

Making DLLs Work

- The program load module includes a Procedure Linkage Table
  - Addresses for routines in DLL resolve to entries in PLT
  - Each PLT entry contains a system call to a run-time loader
- First time a routine is called, we call run-time loader
  - Which finds, loads, and initializes the desired routine
  - Changes the PLT entry to be a jump to loaded routine
  - Then jumps to the newly loaded routine
- Subsequent calls through that PLT entry go directly

Shared Libraries Vs. DLLs

- Both allow code sharing and run-time binding
- Shared libraries:
  - Simple method of linking into programs
  - Shared objects obtained at program load time
- Dynamically Loadable Libraries:
  - Require more complex linking and loading
  - Modules are not loaded until they are needed
  - Complex, per-routine, initialization possible
    - E.g., allocating private data area for persistent local variables

How Do Threads Fit In?

- How do multiple threads in the same process affect layout?
- Each thread has its own registers, PS, PC
- Each thread must have its own stack area
- Maximum size specified at thread creation
  - A process can contain many threads
  - They cannot all grow towards a single hole
  - Thread creator must know max required stack size
  - Stack space must be reclaimed when thread exits

Thread Stack Allocation

Problems With Fixed Size Thread Stacks

- Requires knowing exactly how deep a thread stack can get
  - Before we start running the thread
- Problematic if we do recursion
- How can developers handle this limitation?
  - The use of threads is actually relatively rare
  - Generally used to perform well understood tasks
  - Important to keep this limitation in mind when writing multi-threaded algorithms

How Does the OS Handle Processes?

- The system expects to handle multiple processes
  - Each with its own set of resources
  - Each to be protected from the others
- Memory management handles stomping on each other's memory
  - E.g., use of domain registers
- How does the OS handle the other issues?

Basic OS Process Handling

- The OS will assign processes (or their threads) to cores
  - If more processes than cores, multiplexing them as needed
- When new process assigned to a core, that core must be initialized
  - To give the process illusion that it was always running there
  - Without interruption

Process Descriptors

- Basic OS data structure for dealing with processes
- Stores all information relevant to the process
  - State to restore when process is dispatched
  - References to allocated resources
  - Information to support process operations
- Kept in an OS data structure
- Used for scheduling, security decisions, allocation issues

Linux Process Control Block

- The data structure Linux (and other Unix systems) use to handle processes
- An example of a process descriptor
- Keeps track of:
  – Unique process ID
  – State of the process (e.g., running)
  – Parent process ID
  – Address space information
  – Accounting information
  – And various other things

OS State For a Process

- The state of process's virtual computer
- Registers
  - Program counter, processor status word
  - Stack pointer, general registers
- Virtual address space
  - Text, data, and stack segments
  - Sizes, locations, and contents
- All restored when the process is dispatched
  - Creating the illusion of continuous execution

Process Resource References

- OS needs to keep track of what system resources the process has available
- Extremely important to get this right
  - Process expects them to be available when it runs next
  - If OS gives something it shouldn't, major problem
- OS maintains unforgeable handles for allocated resources
  - Encoding identity and resource state
  - Also helpful for reclamation when process ends

Why Unforgeable Handles?

- Process can ask for any resource
- But it shouldn't always get it
- Process must not be able to create its own OS-level handle to access a resource
  - OS must control which ones the process gets
  - OS data structures not accessible from user-mode
  - Only altered by trusted OS code
    - So if it's there, the OS put it there
    - And it has not been modified by anyone else