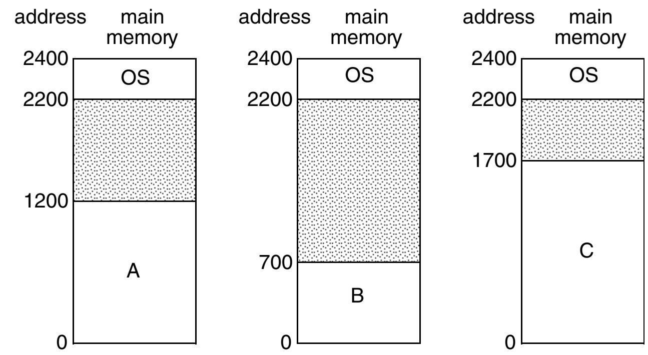
Classifying Information Stored in Memory

- By role in program:
  - Program instructions (unchangeable)
  - Constants: (unchangeable)
    - pi, maxnum, strings used by printf/scanf
  - Variables: (changeable)
    - Locals, globals, function parameters, dynamic storage (from malloc or new)
    - Initialized or uninitialized
- By protection status:
  - Readable and writable: variables
  - Read-only: code, constants
  - Important for sharing data and/or code
- Addresses vs. data:
  - Must modify addresses if program is moved (relocation, garbage collection)

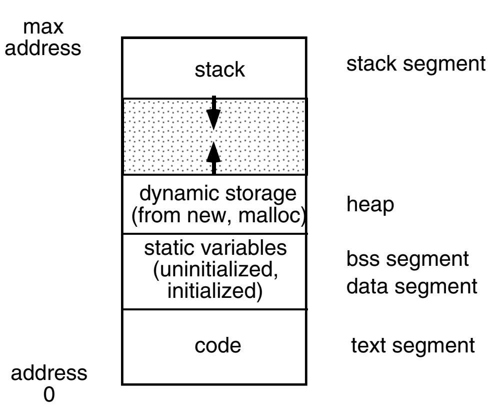
Memory Management in a Uniprogrammed System



- OS gets a fixed segment of memory (usually highest memory)
- One process executes at a time in a single memory segment
  - Process is always loaded at address 0
  - Compiler and linker generate physical addresses
  - Maximum address = memory size – OS size

Classifying Information Stored in Memory (cont.)

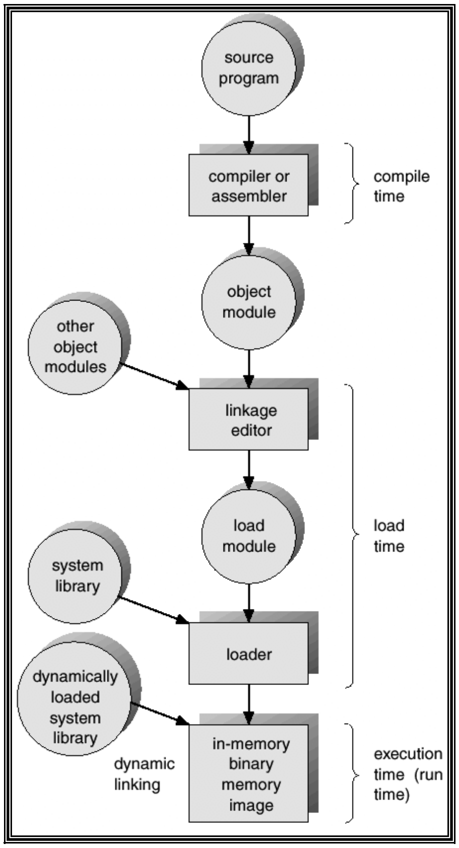
- Binding time (when is space allocated?):
  - Static: before program starts running
    - Program code, static global variables (initialized and uninitialized)
  - Dynamic: as program runs
    - Procedure stack, dynamic storage (space allocated by malloc or new)
- UNIX view of a process's memory (doesn't consider threads):



Segments of a Process

- Process' memory is divided into logical *segments* (text, data, bss, heap, stack)
  - Some are read-only, others read-write
  - Some are known at compile time, others grow dynamically as program runs
- Who assigns memory to segments?
  - *Compiler* and *assembler* generate an *object file* (containing code and data segments) from each *source file*
  - *Linker* combines all the object files for a program into a single executable object file, which is complete and self-sufficient
  - *Loader* (part of OS) loads an executable object file into memory at location(s) determined by the operating system
  - *Program* (as it runs) uses new and malloc to dynamically allocate memory, gets space on stack during function calls

Processing a User Program



Linking

- Contents of object files:
  - File header — size and starting address (in memory) of each segment
  - Segments for code and initialized data
  - Symbol table (symbols, addresses)
  - Patch list (symbols, location)
  - Relocation information (symbols, location)
  - Debugging information
  - For UNIX details, type “man a.out”
- Functions of a *linker*:
  - Combine all files and libraries of a program
  - Regroup all the segments from each file together (one big data segment, etc.)
  - Adjust addresses to match regrouping
  - Result is an executable program

Why is Linking Difficult?

- When assembler assembles a file, it may find *external references* — symbols it doesn’t know about (e.g., printf, scanf)
  - Compiler just puts in an address of 0 when producing the object code
  - Compiler records external symbols and their location (in object file) in a *patch list*, and stores that list in the object file
  - Linker must *resolve* those external references as it links the files together
- Compiler doesn’t know where program will go in memory (if multiprogramming, always 0 for uniprogramming)
  - Compiler just assumes program starts at 0
  - Compiler records *relocation information* (location of addresses to be adjusted later), and stores it in the object file

Loading

- The *loader* loads the completed program into memory where it can be executed
  - Loads code and initialized data segments into memory at specified location
  - Leaves space for uninitialized data (bss)
  - Returns value of start address to operating system
- Alternatives in loading (*next 2 lectures*…)
  - *Absolute loader* — loads executable file at fixed location
  - *Relocatable loader* — loads the program at an arbitrary memory location specified by OS (needed for multiprogramming, not for uniprogramming)
    - Assembler and linker assume program will start at location 0
    - When program is loaded, loader modifies all addresses by adding the real start location to those addresses

Running the Program — Static Memory Allocation

- Compiling, linking, and loading is sufficient for static memory
  - Code, constants, static variables
- In other cases, static allocation is not sufficient:
  - Need dynamic storage — programmer may not know how much memory will be needed when program runs
    - Use malloc or new to get what's necessary when it's necessary
    - For complex data structures (e.g., trees), allocate space for nodes on demand
  - OS doesn't know in advance which procedures will be called (would be wasteful to allocate space for every variable in every procedure in advance)
  - OS must be able to handle recursive procedures

Running the Program — Dynamic Memory Allocation

- Dynamic memory requires two fundamental operations:
  - Allocate dynamic storage
  - Free memory when it's no longer needed
  - Methods vary for stack and heap
- Two basic methods of allocation:
  - Stack (hierarchical)
    - Good when allocation and freeing are somewhat predictable
    - Typically used:
      - to pass parameters to procedures
      - for allocating space for local variables inside a procedure
      - for tree traversal, expression evaluation, parsing, etc.
    - Use stack operations: **push** and **pop**
    - Keeps all free space together in a structured organization
    - Simple and efficient, but restricted

Running the Program — Dynamic Memory Allocation (cont.)

- Two basic methods of allocation:
  - Heap
    - Used when allocation and freeing are not predictable
    - Typically used:
      - for arbitrary list structures, complex data organizations, etc.
    - Use **new** or **malloc** to allocate space, use **delete** or **free** to release space
    - System memory consists of allocated areas and free areas (holes)
    - Problem: eventually end up with many small holes, each too small to be useful
      - This is called *fragmentation*, and it leads to wasted memory
      - Fragmentation wasn't a problem with stack allocation, since we always add/delete from top of stack
      - Solution goal: reuse the space in the holes in such a way as to keep the number of holes small, and their size large
    - Compared to stack: more general, less efficient, more difficult to implement