CSE 3221.3
Operating System Fundamentals

No.8

Memory Management (1)

Prof. Hui Jiang
Dept of Computer Science and Engineering
York University

Memory Management

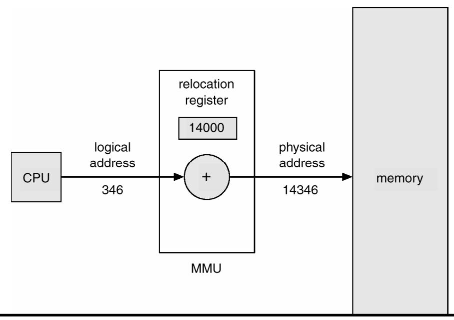
- A program usually resides on a disc as a binary executable file.
- Program must be brought into memory and placed within a process for it to be executed.
- The program can be moved between disk and memory.
- In multiprogramming, we keep several programs in memory
- Memory management algorithms:
  - Contiguous Memory Allocation.
  - Paging.
  - Segmentation.
  - Segmentation with paging
- Memory management needs hardware support – MMU.

Background

- Physical memory consists of a large array of words or bytes, each with its own address.
- In a typical instruction-execution cycle:
  - CPU fetches an instruction from memory according to PC .
  - The instruction is decoded.
  - CPU may fetch operands from memory according to the address in the instruction. (optional)
  - CPU execute in registers
  - CPU saves results into a memory addrss (optional)
- CPU generates address from instruction counter, program address,etc.
- CPU sends the address to a memory management unit (MMU), which is hardware to actually locate the memory at certain location.
  - Memory mapping.
  - Memory protection.

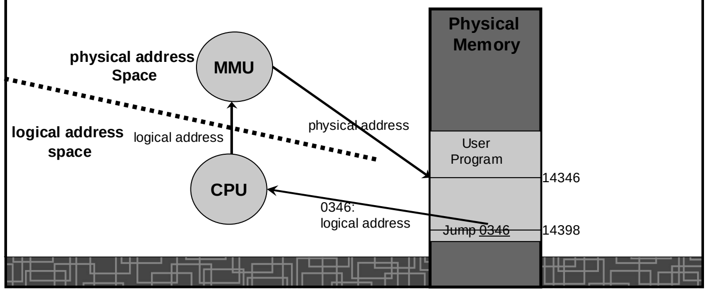
Memory-Management Unit (MMU)

- MMU: maps logical address to physical address.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
- A simple MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

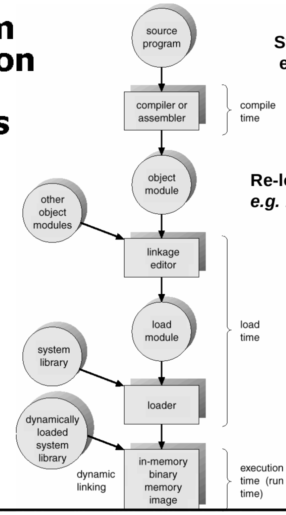


Logical vs. Physical address space (1)

- **Physical address**: the address loaded into the memory-address register to actually address the memory.
- **Logical (virtual) address**: an address generated by the CPU and the address referred by user program; address used in binary codes.



Program Generation & Address



Symbolic address:
e.g., *count,i,j,etc*

Re-locatable address:
e.g. 14 bytes from beginning of module

Logical address:
e.g. 4014, 1058, etc.

Physical address

Address Binding: run-time

- **To load a program into memory, we have to do address binding when generating in-memory binary image.**
- **Address binding: binding the addresses in instructions and data to physical memory addresses.**
  - **In source programs: symbolic addresses (e.g., *count, i, j*, etc.)**
  - **A compiler will bind each symbolic address to a relocatable address (e.g. 14 bytes from the beginning of the module)**
  - **The linkage editor or loader will bind each relocatable address to a logical address (e.g., 4014)**
  - **In run-time, MMU will bind each logical address to a physical address (e.g., 074014)**
  - **The final physical address is used to locate memory.**
- **Most systems allow a user program to be loaded in any part of the physical memory ➔ address binding in run-time**

➔ completely separate physical address from logical address

Logical vs. Physical address space (2)

- **Separating logical address from physical address:**
  - **Requires hardware support – MMI does address mapping dynamically.**
- **Why separating logical address from physical address?**
  - **Consider two old methods …**

Address Binding: compile-time

- In compiling, physical address is generated for every instruction.
- The compiler has to know where the process will reside in memory.
- The code can not change location in memory unless it is re-compiled.
- No separation of logical and physical address spaces.
- Example: .COM format in MS-DOS.

Address Binding: load-time

- The compiler generate relocatable code.
- When OS loading code to memory, physical address is generated for every instruction in the program.
- Still no separation of logical and physical address spaces
- The process can be loaded into different memory locations.
- But once loaded, it can not move during execution.

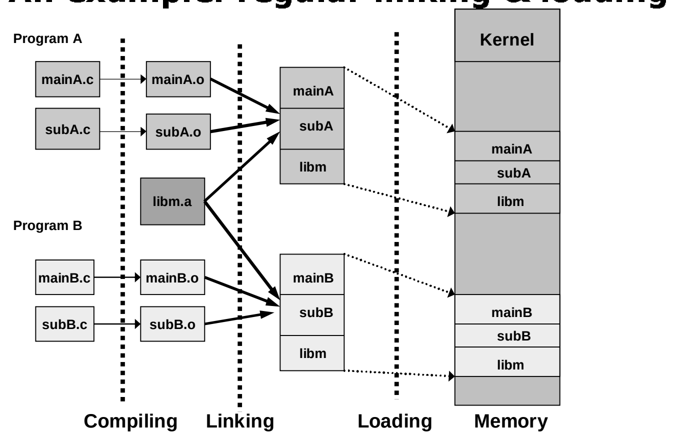
Dynamical Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required; Implemented through program design.
- Each program maintains an address table to indicate which module is in memory and which is not.

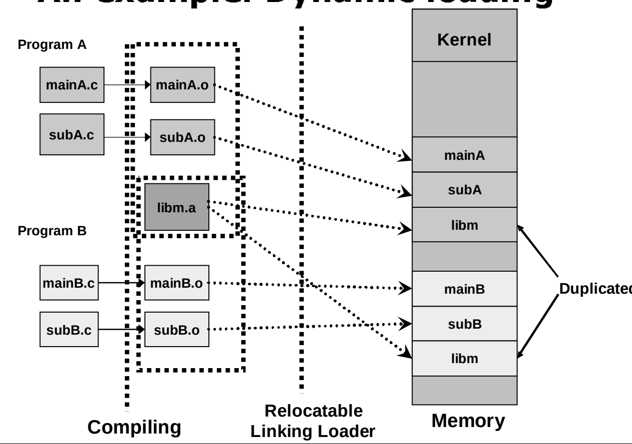
Dynamical Linking

- Linking postponed until execution time.
- In dynamic linking, a *stub,* is included in the executable image for each library-routine reference.
- *Stub*: used to locate the appropriate memory-resident library routine or load the library of it is not in memory.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if the routine is in other processes' memory address, and allow multiple processes to access the same memory space
- Dynamical linking is useful for shared libraries.

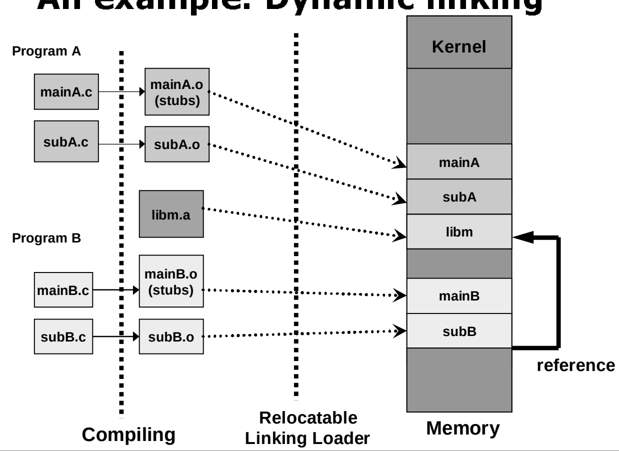
An example: regular linking & loading



An example: Dynamic loading



An example: Dynamic linking

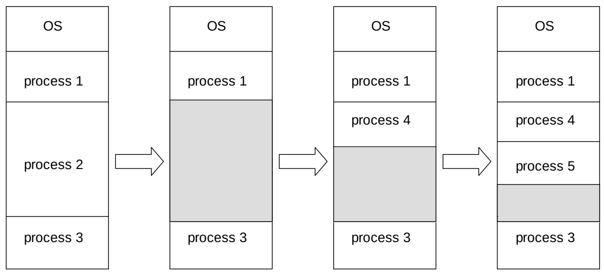


Memory Management Approaches

- Contiguous Memory Allocation
- Paging
- Segmentation
- Segmentation with paging

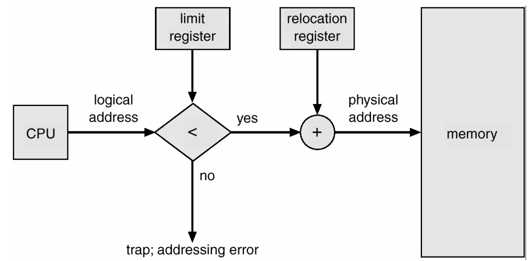
Contiguous Memory Allocation

- Every process is allocated to a single contiguous section of memory



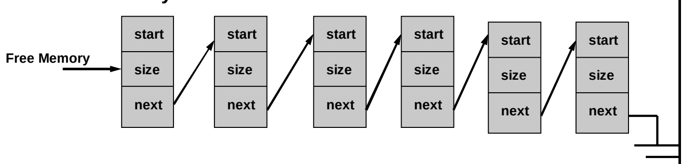
Memory Protection

- Two registers:
  - Limit register: the range of logical address
  - Relocation register: starting position of physical memory
- In context switch, the dispatcher load both registers with correct values.
- Every memory access is checked by MMU hardware as:



Memory Allocation

- OS must keep the information on which parts of memory are available and which are occupied.
- *Hole*: block of available memory
  - holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)
  - One way is to use linked list:



Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes that have various size.

- **First-fit:** Allocate the *first* hole that is big enough.
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit:** Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

1. **First-fit and best-fit better than worst-fit in terms of speed and storage utilization.**
2. **First-fit is faster than best-fit.**

Contiguous Memory Allocation: External Fragmentation

- External fragmentation – total memory space exists to satisfy a request, but it is not contiguous.
- Contiguous memory allocation suffers serious external fragmentation; Free memory is quickly broken into little pieces.
  - *50-percent rule* for first fit (1/3 is wasted)
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - Compaction is very costly
- Reduce external fragmentation by better memory management methods:
  - Paging
  - Segmentation

Contiguous Memory Allocation: Expanding memory

- How to allocate more memory to an existing process?
  - Move-and-Copy may be needed.