Main Memory (Part I)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Motivation and Background

Motivation

- Main memory is a large array of bytes, each with its own address.

Motivation

- Main memory is a large array of bytes, each with its own address.
- Program must be brought (from disk) into memory and placed within a process for it to be run.
  - Machine instructions may take memory addresses as arguments, but not disk addresses.

Motivation

- Main memory is a large array of bytes, each with its own address.

- Program must be brought (from disk) into memory and placed within a process for it to be run.
  - Machine instructions may take memory addresses as arguments, but not disk addresses.

- The CPU fetches instructions from memory according to the value of the program counter.

Motivation

- Main memory is a large array of bytes, each with its own address.
- Program must be brought (from disk) into memory and placed within a process for it to be run.
  - Machine instructions may take memory addresses as arguments, but not disk addresses.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.

Basic Hardware

- Main memory and registers are the only storage that the CPU can access directly.

Basic Hardware

- Main memory and registers are the only storage that the CPU can access directly.

- Register access in one CPU clock (or less)

Basic Hardware

- Main memory and registers are the only storage that the CPU can access directly.
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a stall.

Basic Hardware

- Main memory and registers are the only storage that the CPU can access directly.
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a stall.
- Cache sits between main memory and registers.

Address Protection

- We must protect the OS from access by user processes.

Address Protection

- We must protect the OS from access by user processes.
- We must protect user processes from one another.

Address Protection

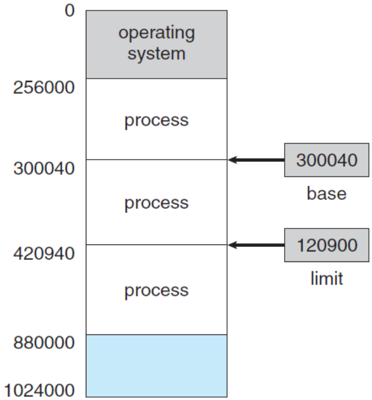
- We must protect the OS from access by user processes.
- We must protect user processes from one another.
- This protection is provided by the hardware.

Address Protection

- We must protect the OS from access by user processes.
- We must protect user processes from one another.
- This protection is provided by the hardware.
- A separate memory space for each process.
  - Determining the range of legal addresses that the process may access.

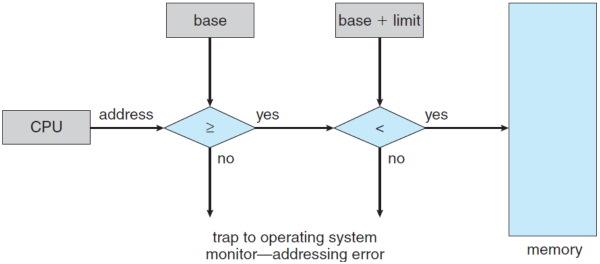
Base and Limit Registers

- A pair of base and limit registers define the logical address space.
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.



Hardware Address Protection

- Any attempt by a user program to access OS memory or other users' memory results in a trap to the OS, which treats the attempt as a fatal error.



Address Binding

Address Binding (1/2)

- Programs on disk, ready to be brought into memory to execute form an input queue.

Address Binding (1/2)

- Programs on disk, ready to be brought into memory to execute form an input queue.
- A user process can reside in any part of the physical memory.
  - Without support, must be loaded into address 00000.

Address Binding (2/2)

- A user program goes through several steps before being executed.

Address Binding (2/2)

- A user program goes through several steps before being executed.
- Addresses may be represented in different ways during these steps.

Address Binding (2/2)

- A user program goes through several steps before being executed.
- Addresses may be represented in different ways during these steps.
  - Source code addresses usually symbolic.

Address Binding (2/2)

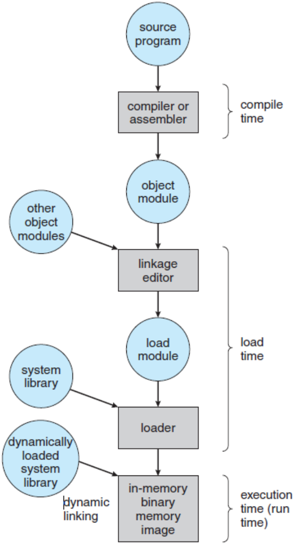
- A user program goes through several steps before being executed.
- Addresses may be represented in different ways during these steps.
  - Source code addresses usually symbolic.
  - A compiler typically binds these symbolic addresses to relocatable addresses, e.g., 14 bytes from beginning of this module.

Address Binding (2/2)

- A user program goes through several steps before being executed.
- Addresses may be represented in different ways during these steps.
  - Source code addresses usually symbolic.
  - A compiler typically binds these symbolic addresses to relocatable addresses, e.g., 14 bytes from beginning of this module.
  - Linker or loader will bind relocatable addresses to absolute addresses, e.g., 74014.

Binding of Instructions and Data to Memory (1/3)

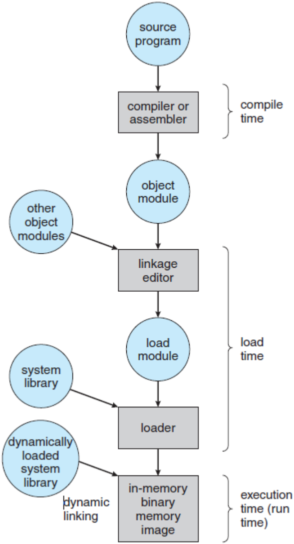
- Address binding of instructions and data to memory addresses can happen at three different stages.



Binding of Instructions and Data to Memory (1/3)

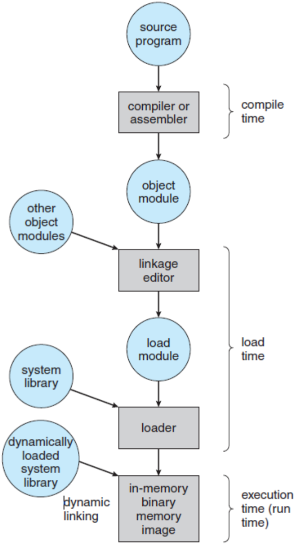
- Address binding of instructions and data to memory addresses can happen at three different stages.

- Compile time: if memory location known a priori, absolute code can be generated.
  - Must recompile code if starting location changes.



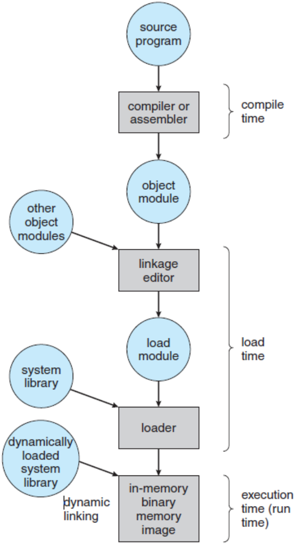
Binding of Instructions and Data to Memory (2/3)

- Load time: must generate relocatable code if memory location is not known at compile time.
  - Final binding is delayed until load time.
  - If the starting address changes, we need only reload the user code to incorporate this changed value.



Binding of Instructions and Data to Memory (3/3)

- Execution time: binding delayed until run time if the process can be moved during its execution from one memory segment to another.
  - Need hardware support



Logical vs. Physical Address Space (1/2)

- Logical address (virtual address): address generated by the CPU.
  - Logical address space is the set of all logical addresses generated by a program.

Logical vs. Physical Address Space (1/2)

- Logical address (virtual address): address generated by the CPU.
  - Logical address space is the set of all logical addresses generated by a program.
- Physical address: address seen by the memory unit.
  - Physical address space is the set of all physical addresses generated by a program.

Logical vs. Physical Address Space (1/2)

- Logical address (virtual address): address generated by the CPU.
  - Logical address space is the set of all logical addresses generated by a program.

- Physical address: address seen by the memory unit.
  - Physical address space is the set of all physical addresses generated by a program.

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Logical vs. Physical Address Space (2/2)

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.

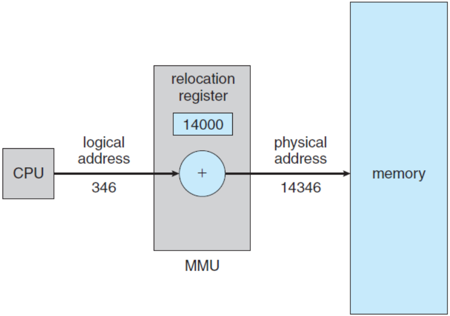
- Logical and physical addresses differ in execution-time address-binding scheme.

Memory-Management Unit (MMU) (1/2)

- Hardware device that at run time maps virtual to physical address.

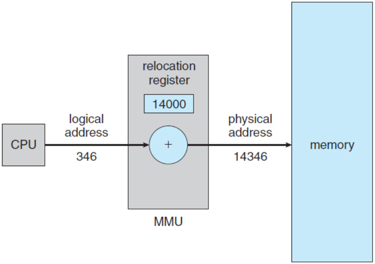
Memory-Management Unit (MMU) (1/2)

- Hardware device that at run time maps virtual to physical address.
- As a simple 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.
  - Base register now called relocation register.



Memory-Management Unit (MMU) (2/2)

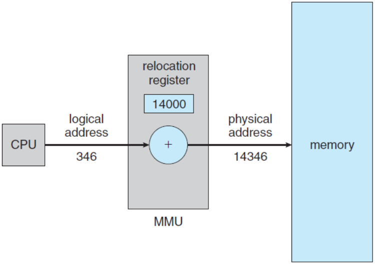
- Two different types of addresses:
  - Logical addresses: range 0 to max
  - Physical addresses: range R + 0 to R + max for a base value R



Memory-Management Unit (MMU) (2/2)

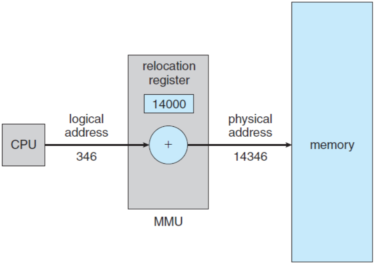
- Two different types of addresses:
  - Logical addresses: range 0 to max
  - Physical addresses: range R + 0 to R + max for a base value R

- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.



Memory-Management Unit (MMU) (2/2)

- Two different types of addresses:
  - Logical addresses: range 0 to max
  - Physical addresses: range R + 0 to R + max for a base value R
- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- These logical addresses must be mapped to physical addresses before they are used.



Dynamic Linking and Loading

Dynamic Loading (1/2)

- Routine/library is not loaded until it is called.

Dynamic Loading (1/2)

- Routine/library is not loaded until it is called.
- The main program is loaded into memory and is executed.

Dynamic Loading (1/2)

- Routine/library is not loaded until it is called.
- The main program is loaded into memory and is executed.
- When a routine is needed, if it has not been loaded, the loader loads the desired routine into memory and to update the program's address tables to reflect this change.

Dynamic Loading (1/2)

- Routine/library is not loaded until it is called.
- The main program is loaded into memory and is executed.
- When a routine is needed, if it has not been loaded, the loader loads the desired routine into memory and to update the program's address tables to reflect this change.
- Then control is passed to the newly loaded routine.

Dynamic Loading (1/2)

- Routine/library is not loaded until it is called.
- The main program is loaded into memory and is executed.
- When a routine is needed, if it has not been loaded, the loader loads the desired routine into memory and to update the program's address tables to reflect this change.
- Then control is passed to the newly loaded routine.
- All routines kept on disk in relocatable load format.

Dynamic Loading (2/2)

- Better memory-space utilization; unused routine is never loaded.

Dynamic Loading (2/2)

- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.

Dynamic Loading (2/2)

- 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 OS is required.

Dynamic Loading (2/2)

- 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 OS is required.
- Implemented through program design.

Dynamic Loading (2/2)

- 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 OS is required.
- Implemented through program design.
- OS can help by providing libraries to implement dynamic loading.

Dynamic Liking (1/2)

- Static linking: system libraries and program code combined by the loader into the binary program image.

Dynamic Liking (1/2)

- Static linking: system libraries and program code combined by the loader into the binary program image.

- Dynamic linking: linking postponed until execution time.

Dynamic Liking (1/2)

- Static linking: system libraries and program code combined by the loader into the binary program image.

- Dynamic linking: linking postponed until execution time.

- Stub: small piece of code, used to locate the appropriate memory-resident library routine.

Dynamic Liking (1/2)

- Static linking: system libraries and program code combined by the loader into the binary program image.
- Dynamic linking: linking postponed until execution time.
- Stub: small piece of code, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.

Dynamic Liking (2/2)

- OS checks if routine is in processes memory address.

Dynamic Liking (2/2)

- OS checks if routine is in processes memory address.
- If not in address space, add to address space.

Dynamic Liking (2/2)

- OS checks if routine is in processes memory address.
- If not in address space, add to address space.
- Dynamic linking is particularly useful for shared libraries.

Swapping

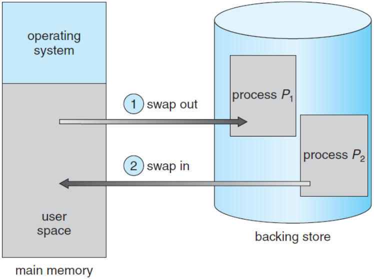
Swapping (1/3)

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

Swapping (1/3)

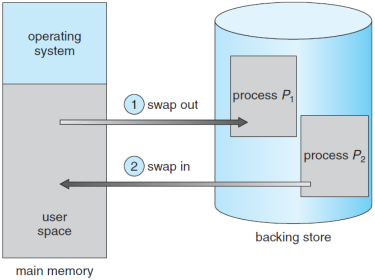
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- Backing store: fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.



Swapping (1/3)

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Backing store: fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Total memory space of processes can exceed physical memory.



Swapping (2/3)

- Does the swapped out process need to swap back in to same physical addresses?

Swapping (2/3)

- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method.

Swapping (2/3)

- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method.
- Pending I/O: cannot swap out as I/O would occur to wrong process.

Swapping (2/3)

- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method.
- Pending I/O: cannot swap out as I/O would occur to wrong process.
  - Or always transfer I/O to kernel space, then to I/O device.
  - Known as double buffering, adds overhead.

Swapping (3/3)

- Standard swapping not used in modern OSs, but modified version is common.
- Swap only when free memory extremely low.
- Disabled again once memory demand reduced below threshold.

Swapping Cost

- Major part of swap time is transfer time; which is proportional to the amount of memory swapped.

Swapping Cost

- Major part of swap time is transfer time; which is proportional to the amount of memory swapped.

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process.

Swapping Cost

- Major part of swap time is transfer time; which is proportional to the amount of memory swapped.

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process.

- Example:
  - 100MB process swapping to hard disk with transfer rate of 50MB/sec.
  - Swap out time of 2s + swap in of same sized process.
  - Total context switch swapping component time of 4s.

Swapping on Mobile Systems (1/2)

- Not typically supported.

Swapping on Mobile Systems (1/2)

- Not typically supported.
- Flash memory based
  - Small amount of space
  - Limited number of write cycles
  - Poor throughput between flash memory and CPU on mobile platform

Swapping on Mobile Systems (2/2)

- Instead use other methods to free memory if low.

Swapping on Mobile Systems (2/2)

- Instead use other methods to free memory if low.
- iOS asks apps to voluntarily relinquish allocated memory.

Swapping on Mobile Systems (2/2)

- Instead use other methods to free memory if low.
- iOS asks apps to voluntarily relinquish allocated memory.
- Read-only data thrown out and reloaded from flash if needed.

Swapping on Mobile Systems (2/2)

- Instead use other methods to free memory if low.
- iOS asks apps to voluntarily relinquish allocated memory.
- Read-only data thrown out and reloaded from flash if needed.
- Failure to free can result in termination.

Swapping on Mobile Systems (2/2)

- Instead use other methods to free memory if low.
- iOS asks apps to voluntarily relinquish allocated memory.
- Read-only data thrown out and reloaded from flash if needed.
- Failure to free can result in termination.
- Android terminates apps if low free memory, but first writes application state to flash for fast restart.

Contiguous Memory Allocation

Contiguous Allocation (1/2)

- Main memory must support both OS and user processes.

Contiguous Allocation (1/2)

- Main memory must support both OS and user processes.
- Limited resource, must allocate efficiently.

Contiguous Allocation (1/2)

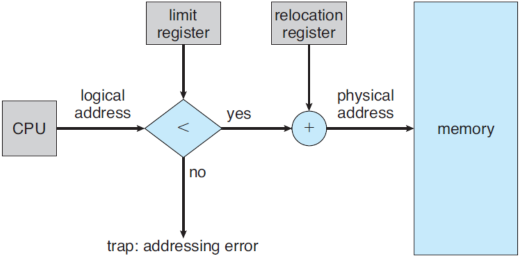
- Main memory must support both OS and user processes.
- Limited resource, must allocate efficiently.
- Contiguous allocation is one early method.

Contiguous Allocation (1/2)

- Main memory must support both OS and user processes.
- Limited resource, must allocate efficiently.
- Contiguous allocation is one early method.
- Main memory usually into two partitions:
  - Resident OS, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
  - Each process contained in single contiguous section of memory.

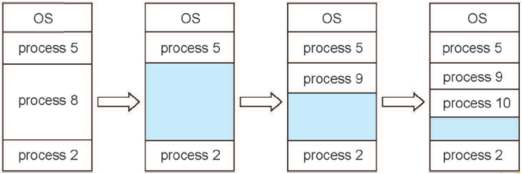
Contiguous Allocation (2/2)

- Relocation registers used to protect user processes from each other, and from changing OS code and data.
  - Base register contains value of smallest physical address.
  - Limit register contains range of logical addresses.
  - MMU maps logical address dynamically.



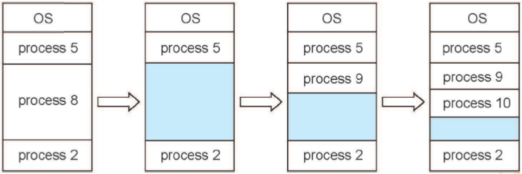
Multiple-Partition Allocation (1/2)

- Memory is to divide memory into several fixed-sized partitions.



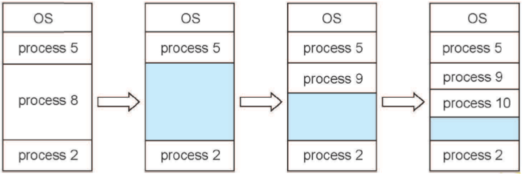
Multiple-Partition Allocation (1/2)

- Memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.



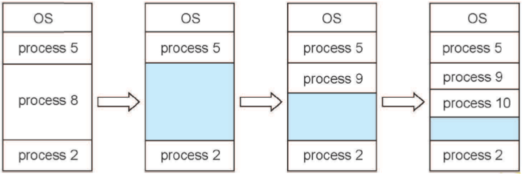
Multiple-Partition Allocation (1/2)

- Memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- Degree of multiprogramming limited by number of partitions.



Multiple-Partition Allocation (1/2)

- Memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- Degree of multiprogramming limited by number of partitions.
- When a partition is free, a process is selected from the input queue and is loaded into the free partition.



Multiple-Partition Allocation (2/2)

- Variable-partition sizes for efficiency (sized to a given process' needs).

Multiple-Partition Allocation (2/2)

- Variable-partition sizes for efficiency (sized to a given process' needs).
- Hole: block of available memory.

Multiple-Partition Allocation (2/2)

- Variable-partition sizes for efficiency (sized to a given process' needs).
- Hole: block of available memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.

Multiple-Partition Allocation (2/2)

- Variable-partition sizes for efficiency (sized to a given process' needs).
- Hole: block of available memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Process exiting frees its partition, adjacent free partitions combined

Multiple-Partition Allocation (2/2)

- Variable-partition sizes for efficiency (sized to a given process' needs).
- Hole: block of available memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Process exiting frees its partition, adjacent free partitions combined
- OS maintains information about: allocated partitions and free partitions (hole)

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
- First-fit: allocate the first hole that is big enough

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
- 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.

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
- 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.

Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
- 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.
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Fragmentation (1/3)

- External fragmentation: total memory space exists to satisfy a request, but it is not contiguous.

Fragmentation (1/3)

- External fragmentation: total memory space exists to satisfy a request, but it is not contiguous.

- Internal fragmentation: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Fragmentation (1/3)

- External fragmentation: total memory space exists to satisfy a request, but it is not contiguous.

- Internal fragmentation: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

- First fit analysis reveals that given N blocks allocated, $0.5N$ blocks lost to fragmentation: 50-percent rule

Fragmentation (2/3)

- Compaction: a solution to the problem of external fragmentation.

Fragmentation (2/3)

- Compaction: a solution to the problem of external fragmentation.
- Shuffle memory contents to place all free memory together in one large block.

Fragmentation (2/3)

- Compaction: a solution to the problem of external fragmentation.
- 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.

Fragmentation (2/3)

- Compaction: a solution to the problem of external fragmentation.
- 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.
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers.

Fragmentation (3/3)

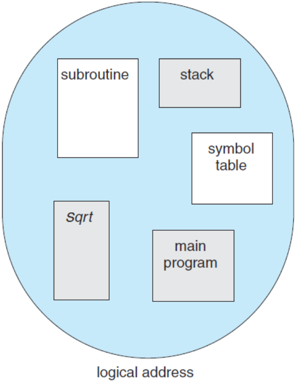
- Another possible solution to the external fragmentation problem: permit the logical address space of the processes to be noncontiguous.

- Two techniques:
  - Segmentation
  - Paging

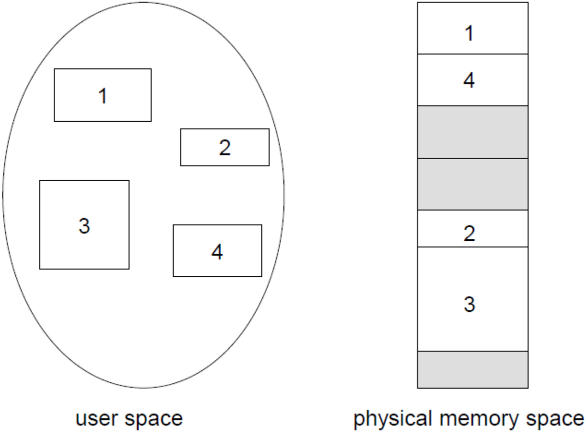
Segmentation

Fragmentation (2/2)

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments.
- A segment is a logical unit such as:
  - Main program
  - Procedure
  - Function
  - Object
  - ...



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a tuple: $\langle \texttt{segment\_number}, \texttt{offset} \rangle$

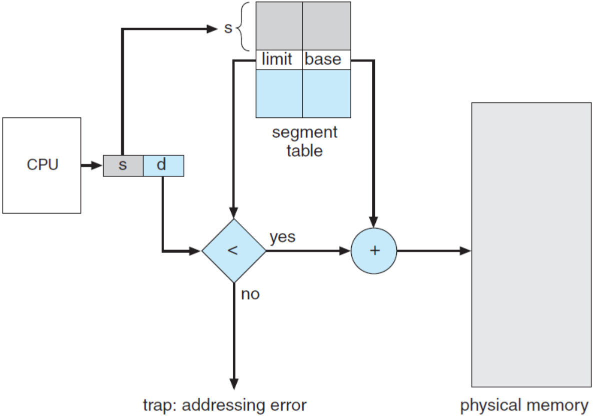
Segmentation Architecture

- Logical address consists of a tuple: $\langle \texttt{segment\_number}, \texttt{offset} \rangle$
- Segment table: maps two-dimensional user-defined addresses into one-dimensional physical address.

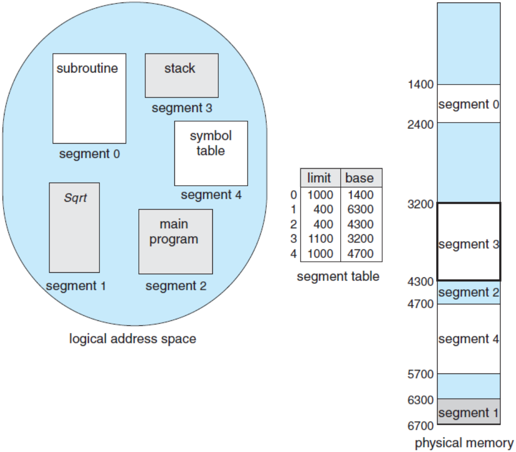
Segmentation Architecture

- Logical address consists of a tuple: $\langle \texttt{segment\_number}, \texttt{offset} \rangle$
- Segment table: maps two-dimensional user-defined addresses into one-dimensional physical address.
- Each table entry has:
  - Base: contains the starting physical address where the segments reside in memory
  - Limit: specifies the length of the segment

Segmentation Hardware

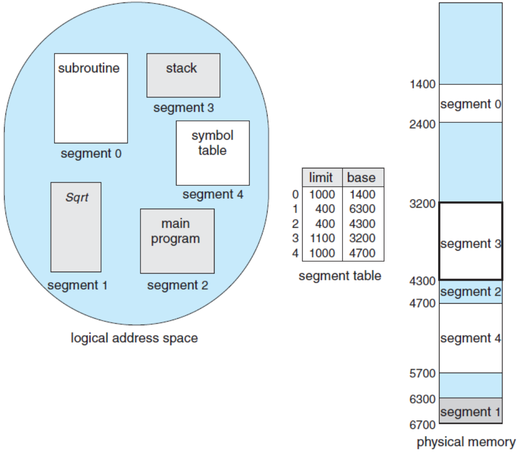


Segmentation Example



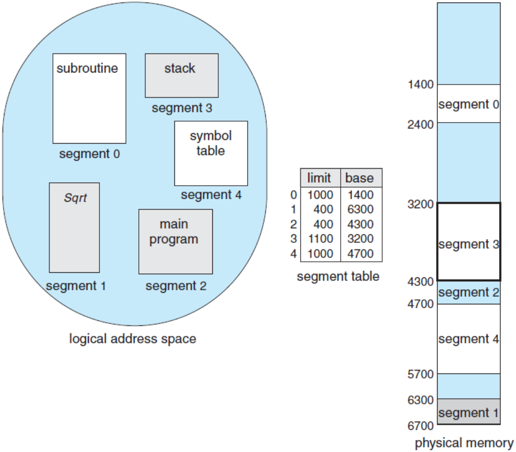
- A reference to byte 53 of segment 2:

Segmentation Example



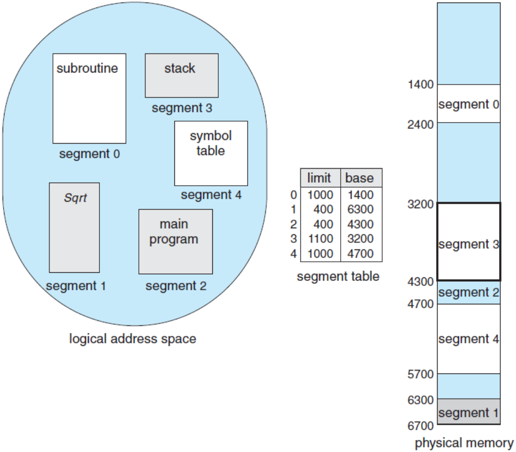
- A reference to byte 53 of segment 2: $4300 + 53 = 4353$

Segmentation Example



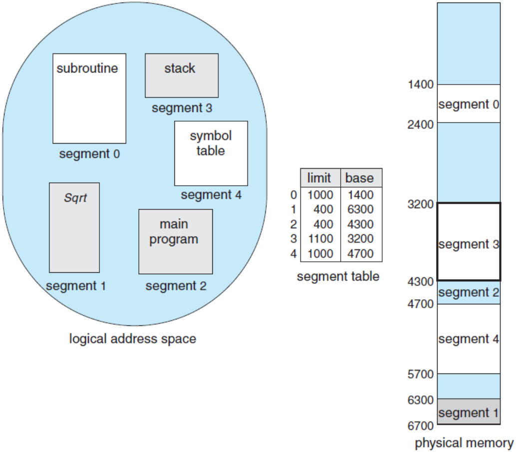
- A reference to byte 53 of segment 2: $4300 + 53 = 4353$
- A reference to byte 852 of segment 3:

Segmentation Example



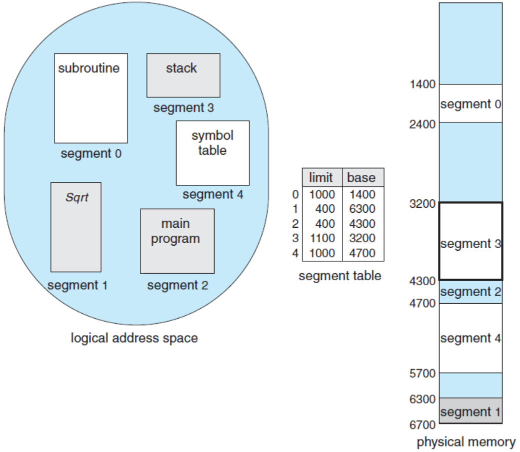
- A reference to byte 53 of segment 2: $4300 + 53 = 4353$
- A reference to byte 852 of segment 3: $3200 + 852 = 4052$

Segmentation Example



- A reference to byte 53 of segment 2: $4300 + 53 = 4353$
- A reference to byte 852 of segment 3: $3200 + 852 = 4052$
- A reference to byte 1222 of segment 0:

Segmentation Example



- A reference to byte 53 of segment 2: $4300 + 53 = 4353$
- A reference to byte 852 of segment 3: $3200 + 852 = 4052$
- A reference to byte 1222 of segment 0: trap to OS

Summary

Summary

- Main memory

Summary

- Main memory
- Address protection: base + limit

Summary

- Main memory
- Address protection: base + limit
- Address binding: compile time, load time, execution time

Summary

- Main memory
- Address protection: base + limit
- Address binding: compile time, load time, execution time
- Logical and physical address, MMU

Summary

- Main memory
- Address protection: base + limit
- Address binding: compile time, load time, execution time
- Logical and physical address, MMU
- Swapping: backing store, swapping cost

Summary

- Main memory
- Address protection: base + limit
- Address binding: compile time, load time, execution time
- Logical and physical address, MMU
- Swapping: backing store, swapping cost
- Contiguous memory allocation: partitions, holes, first-fit, best-fit, worst-fit

Summary

- Main memory
- Address protection: base + limit
- Address binding: compile time, load time, execution time
- Logical and physical address, MMU
- Swapping: backing store, swapping cost
- Contiguous memory allocation: partitions, holes, first-fit, best-fit, worst-fit
- External and internal fragmentation: compaction, segmentation, paging

Summary

- Main memory
- Address protection: base + limit
- Address binding: compile time, load time, execution time
- Logical and physical address, MMU
- Swapping: backing store, swapping cost
- Contiguous memory allocation: partitions, holes, first-fit, best-fit, worst-fit
- External and internal fragmentation: compaction, segmentation, paging
- Segmentation: noncontiguous address, user view of memory

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.