Department of Electronics & Communication Engg.

Course : Operating Systems -15EC553. **Sem.: 5th (2018-19 ODD)**

Course Coordinator:

Prof. Nyamatulla M Patel

Operating Systems-15EC553

Module-03
Memory Management

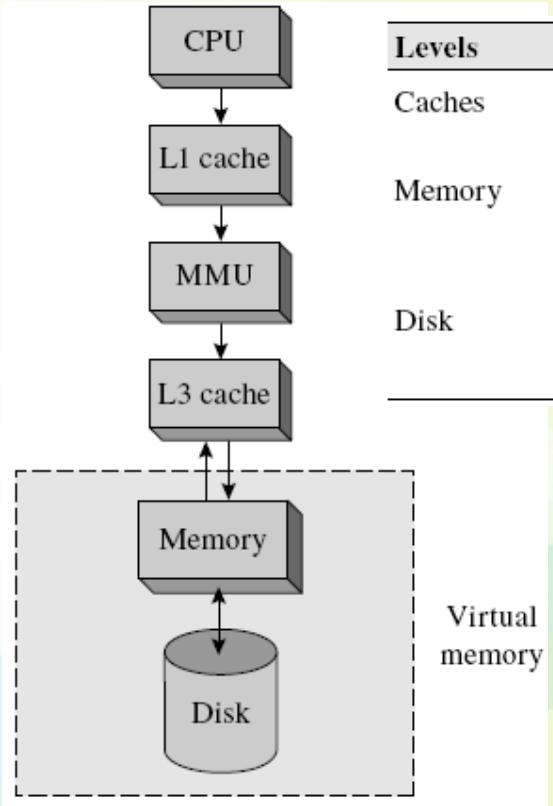
Introduction

- Managing the Memory Hierarchy
- Static and Dynamic Memory Allocation
- Execution of Programs
- Memory Allocation to a Process
- Heap Management
- Contiguous Memory Allocation

Introduction (continued)

- Noncontiguous Memory Allocation
- Paging
- Segmentation
- Segmentation with Paging
- Kernel Memory Allocation
- Using Idle RAM Effectively

Managing the Memory Hierarchy



Levels	How managed	Performance issues
Caches	Allocation and use is managed by hardware	Ensuring high hit ratios
Memory	Allocation is managed by the kernel and use of allocated memory is managed by run-time libraries	(1) Accommodating more process in memory, (2) Ensuring high hit ratios
Disk	Allocation and use is managed by the kernel	Quick loading and storing of parts of process address spaces

Figure 11.1 Managing the memory hierarchy.

Static and Dynamic Memory Allocation

- Memory allocation is an aspect of a more general action in software operation known as *binding*

Definition 11.1 Static Binding A binding performed before the execution of a program (or operation of a software system) is set in motion.

Definition 11.2 Dynamic Binding A binding performed during the execution of a program (or operation of a software system).

  - *Static allocation* performed by compiler, linker, or loader
    - Sizes of data structures must be known a priori
  - *Dynamic allocation* provides flexibility
    - Memory allocation actions constitute an overhead during operation

Execution of Programs

- A has to be transformed before it can be executed
  - Many of these transformations perform memory bindings
    - Accordingly, an address is called compiled address, linked address, etc

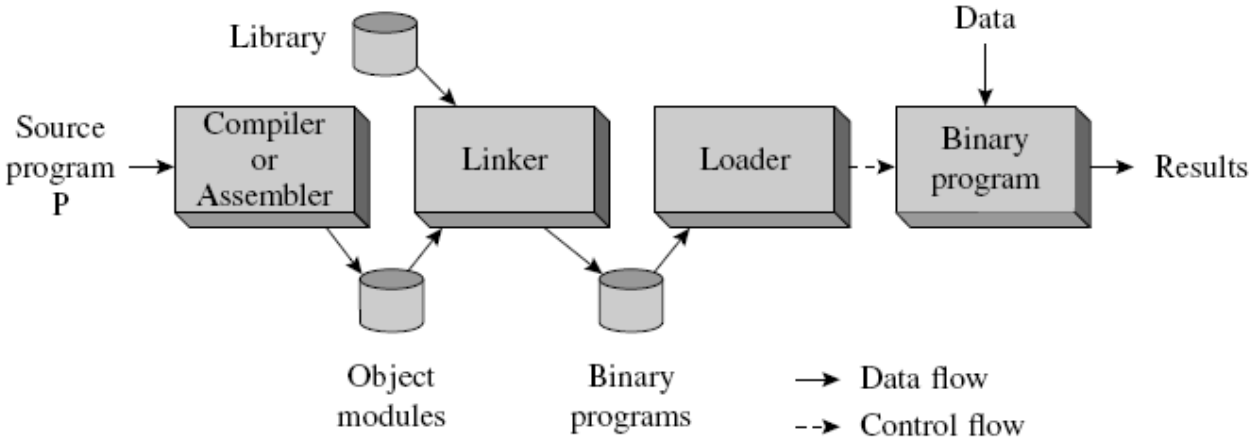


Figure 11.2 Schematic diagram of transformation and execution of a program.

A Simple Assembly Language

- Format of an assembly language statement:

  [Label] *<Opcode>* *<operand spec>* ,*<operand spec>*

  - First operand is always a GPR
    - **AREG, BREG, CREG** or **DREG**
  - Second operand is a GPR or a symbolic name that corresponds to a memory byte
  - Opcodes are self-explanatory
    - **ADD, MULT, MOVER, MOVEM, BC**
  - For simplicity, assume that addresses and constants are in decimal, and instructions occupy 4 bytes

Relocation

	Assembly statement		*Generated code*	
			Address	*Code*
	START	500		
	ENTRY	TOTAL		
	EXTRN	MAX, ALPHA		
	READ	A	500)	+ 09 0 540
LOOP			504)	
	⋮			
	MOVER	AREG, ALPHA	516)	+ 04 1 000
	BC	ANY, MAX	520)	+ 06 6 000
	⋮			
	BC	LT, LOOP	532)	+ 06 1 504
	STOP		536)	+ 00 0 000
A	DS	1	540)	
TOTAL	DS	3	541)	
	END			

Figure 11.3 Assembly program P and its generated code.

- Instructions using memory addresses are *address-sensitive*
  - *Relocation* is needed if program is to execute correctly in some other memory area: involves changing addresses

Relocation (continued)

- Relocation may be performed in two ways:
  - Static (before program is executed)
  - Dynamic (during program execution)
    - Alternative 1: suspend execution and relocate
    - Alternative 2: use a *relocation register*

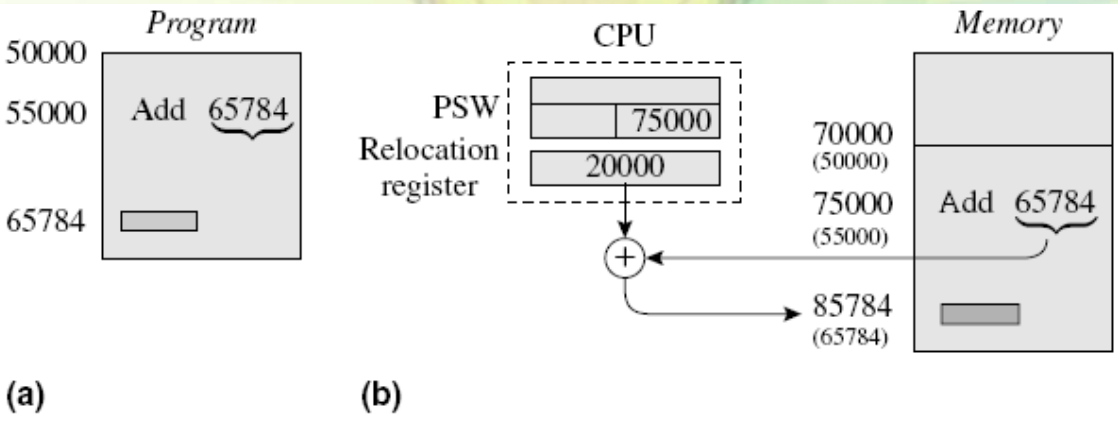


Figure 11.4 Program relocation using a relocation register: (a) program; (b) its execution.

Linking

- Assembler puts information about **ENTRY** and **EXTRN** statements in an object module for linker's use
  - Called *entry points* and *external references*
- *Linking:* binding external reference to correct address
  - *Linker* links modules to form an executable program
  - *Loader* loads program in memory for execution
  - *Static linking* produces binaries with no unresolved external references
  - *Dynamic linking* enables sharing of a single copy of a module and dynamic updating of library modules
    - E.g., Dynamically linked libraries (DLLs)

Program Forms Employed in Operating Systems

Table 11.1 Program Forms Employed in Operating Systems

Program form	Features
Object module	Contains instructions and data of a program and information required for its relocation and linking.
Binary program	Ready-to-execute form of a program.
Dynamically linked program	Linking is performed in a lazy manner, i.e., an object module defining a symbol is linked to a program only when that symbol is referenced during the program's execution.
Self-relocating program	The program can relocate itself to execute in any area of memory.
Reentrant program	The program can be executed on several sets of data concurrently.

Self-Relocating Programs

- A self-relocating program:
  - Knows its own translated origin and translated addresses of its address-sensitive instructions
  - Contains *relocating logic*
    - Start address of the relocating logic is specified as the execution start address of the program
  - Starts off by calling a dummy function
    - Return address provides its own execution-time address
    - Now performs its own relocation using this address
  - Passes control to first instruction to begin its own execution

Re-entrant Programs

- Can be executed concurrently by many users
  - Code accesses its data structures through the GPR

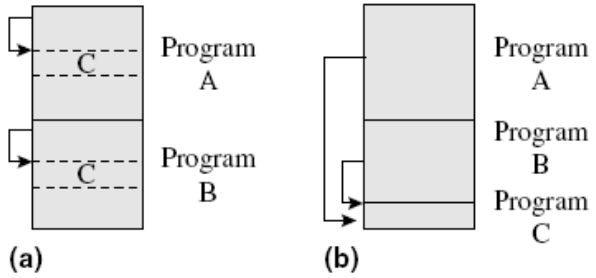


Figure 11.5 Sharing of program C by programs A and B: (a) static sharing; (b) dynamic sharing.

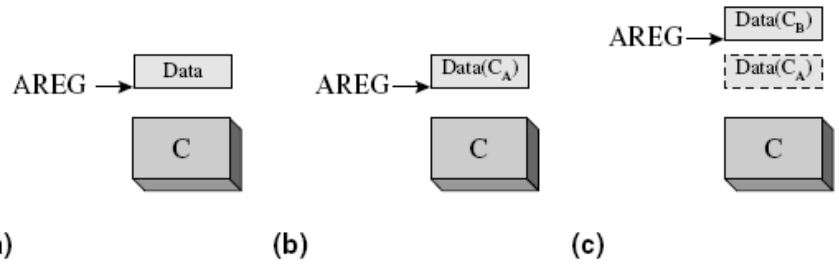


Figure 11.6 (a) Structure of a reentrant program; (b)–(c) concurrent invocations of the program.

Stacks and Heaps

- *Stack*: LIFO allocations/deallocations (*push* and *pop*)
  - Memory is allocated when a function, procedure or block is entered and is deallocated when it is exited

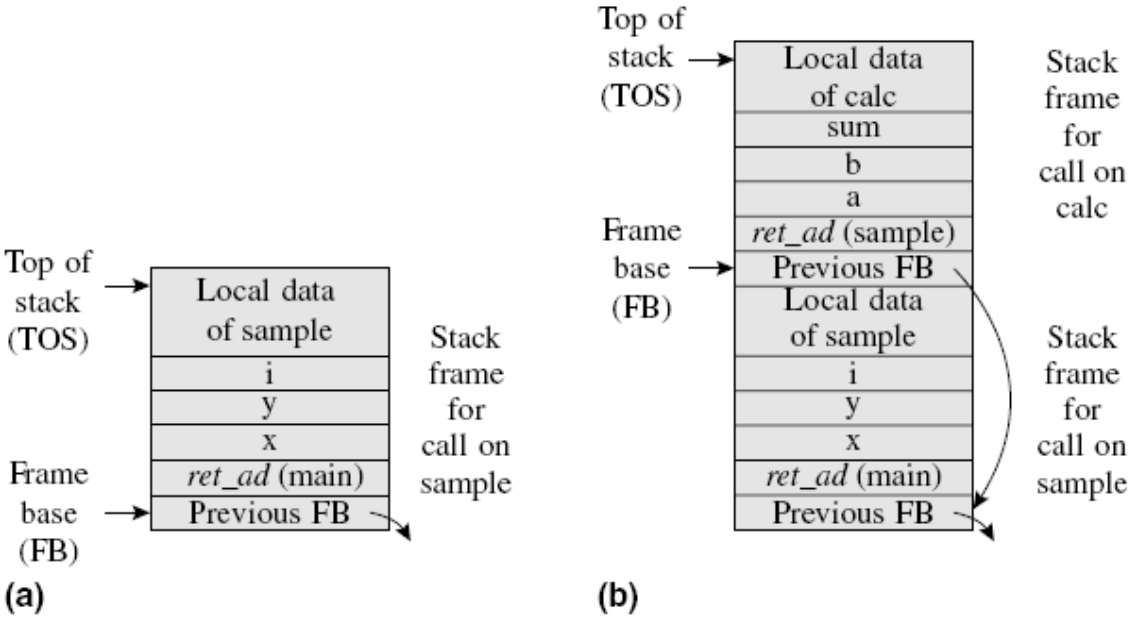


Figure 11.7 Stack after (a) `main` calls `sample`; (b) `sample` calls `calc`.

Stacks and Heaps (continued)

```
float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc(5, sizeof(float));
floatptr2 = (float *) calloc(4, sizeof(float));
intptr = (int *) calloc(10, sizeof(int));
free(floatptr2);
```

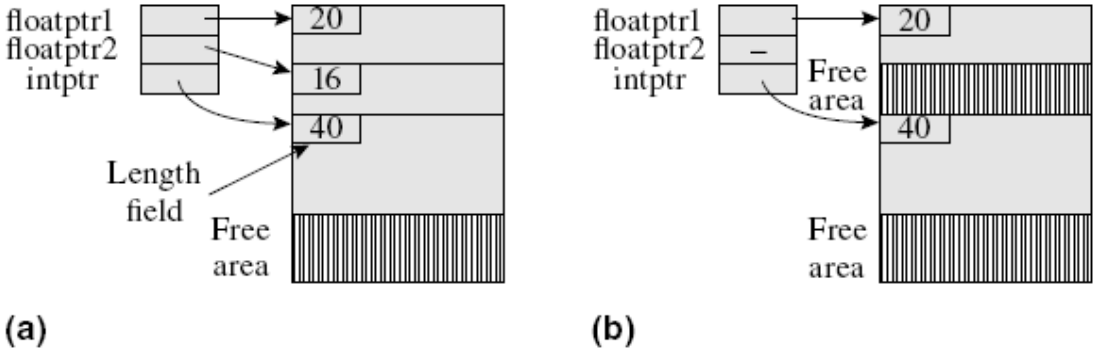


Figure 11.8 (a) A heap; (b) A "hole" in the allocation when memory is deallocated.

- A *heap* permits random allocation/deallocation
  - Used for *program-controlled dynamic data (PCD* data*)*

Memory Allocation to a Process

- Stacks and Heaps
- The Memory Allocation Model
- Memory Protection

The Memory Allocation Model

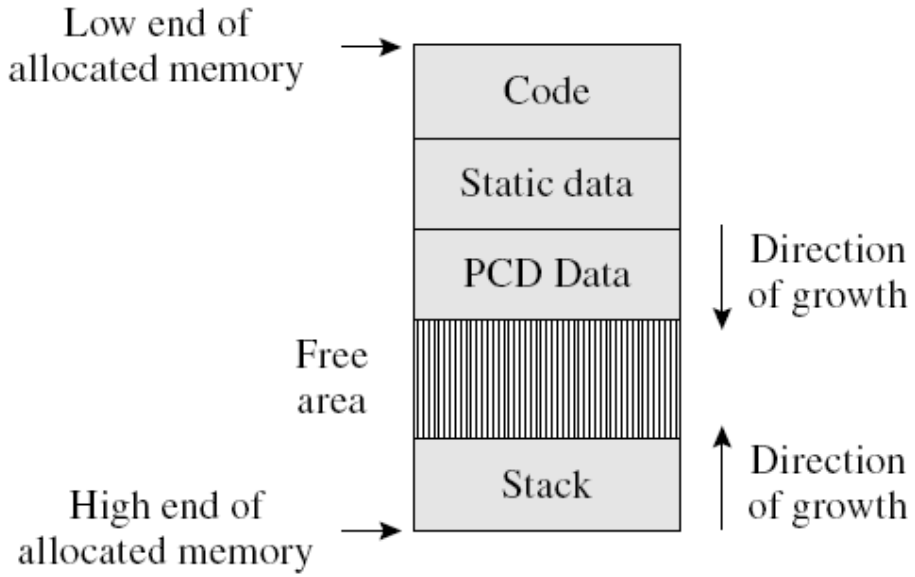


Figure 11.9 Memory allocation model for a process.

Memory Protection

- Memory protection uses *base* and *size* register
  - *Memory protection violation* interrupt is raised if an address used in a program lies outside their range
    - On processing interrupt, kernel aborts erring process
  - Base/size registers constitute the memory protection information (MPI) field of PSW
    - Kernel loads appropriate values while scheduling a process
      - Loading and saving are privileged instructions
    - When a *relocation* register is used, this register and the size register constitute MPI field of PSW

Heap Management

- Reuse of Memory
  - Maintaining a Free List
  - Performing Fresh Allocations by Using a Free List
  - Memory Fragmentation
  - Merging of Free Memory Areas
- Buddy System and Power-of-2 Allocators
- Comparing Memory Allocators
- Heap Management in Windows

Reuse of Memory

Table 11.2 Kernel Functions for Reuse of Memory

Function	Description
Maintain a free list	The *free list* contains information about each free memory area. When a process frees some memory, information about the freed memory is entered in the free list. When a process terminates, each memory area allocated to it is freed, and information about it is entered in the free list.
Select a memory area for allocation	When a new memory request is made, the kernel selects the most suitable memory area from which memory should be allocated to satisfy the request.
Merge free memory areas	Two or more adjoining free areas of memory can be merged to form a single larger free area. The areas being merged are removed from the free list and the newly formed larger free area is entered in it.

Maintaining a Free List

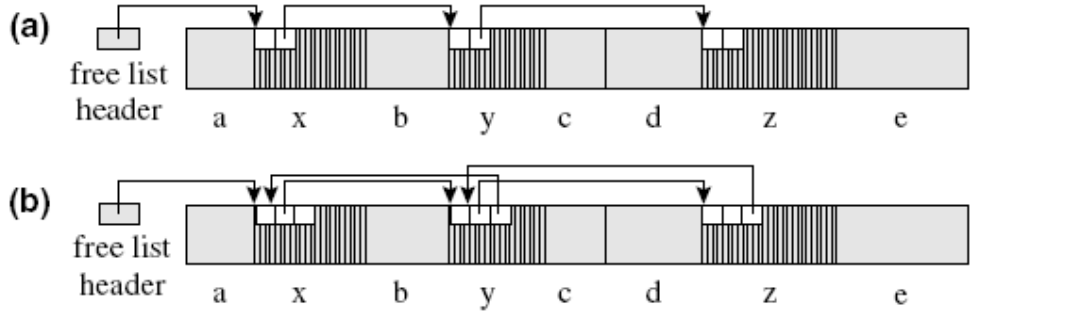


Figure 11.10 Free area management: (a) singly linked free list; (b) doubly linked free list.

- For each memory area in free list, kernel maintains:
  - Size of the memory area
  - Pointers used for forming the list
- Kernel stores this information it in the first few bytes of a free memory area itself

Performing Fresh Allocations by Using a Free List

- Three techniques can be used:
  - First-fit technique: uses first large-enough area
  - Best-fit technique: uses smallest large-enough area
  - Next-fit technique: uses next large-enough area

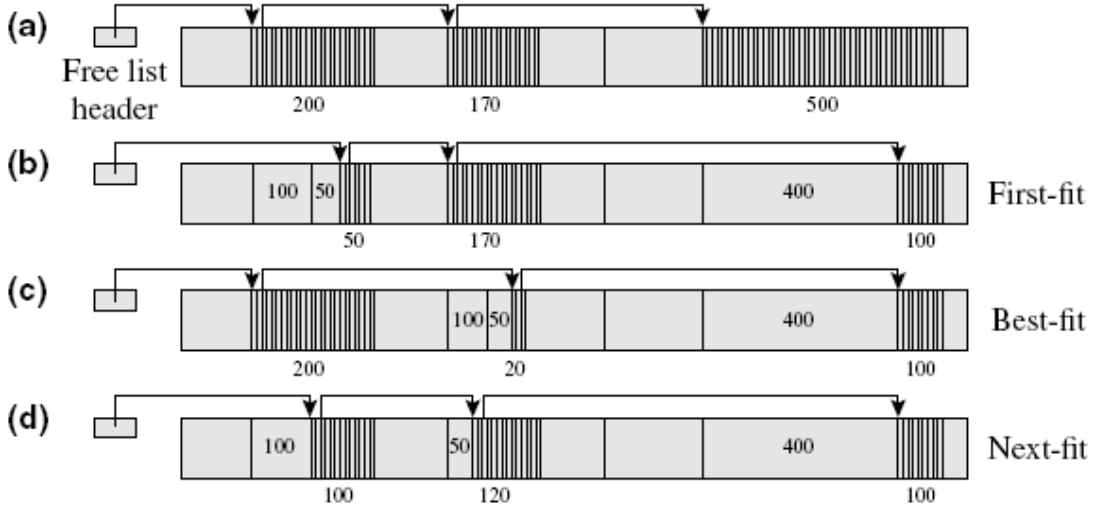


Figure 11.11 (a) Free list; (b)–(d) allocation using first-fit, best-fit and next-fit.

Memory Fragmentation

- Fragmentation leads to poor memory utilization

Definition 11.3 Memory Fragmentation The existence of unusable areas in the memory of a computer system.

Table 11.3 Forms of Memory Fragmentation

Form of fragmentation	Description
External fragmentation	Some area of memory is too small to be allocated.
Internal fragmentation	More memory is allocated than requested by a process, hence some of the allocated memory remains unused.

Merging of Free Memory Areas

- External fragmentation can be countered by merging free areas of memory
- Two generic techniques:
  - Boundary tags
  - Memory compaction

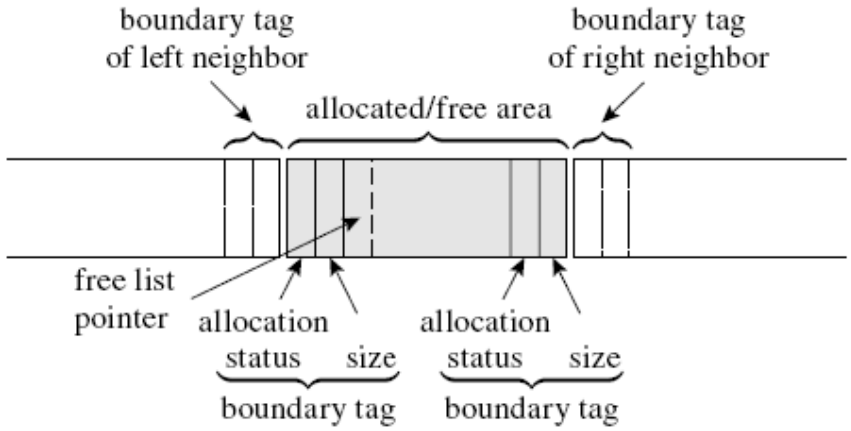


Figure 11.12 Boundary tags and the free list pointer.

Merging of Free Memory Areas (continued)

- A *tag* is a status descriptor for a memory area
  - When an area of memory becomes free, kernel checks the boundary tags of its neighboring areas
    - If a neighbor is free, it is merged with newly freed area

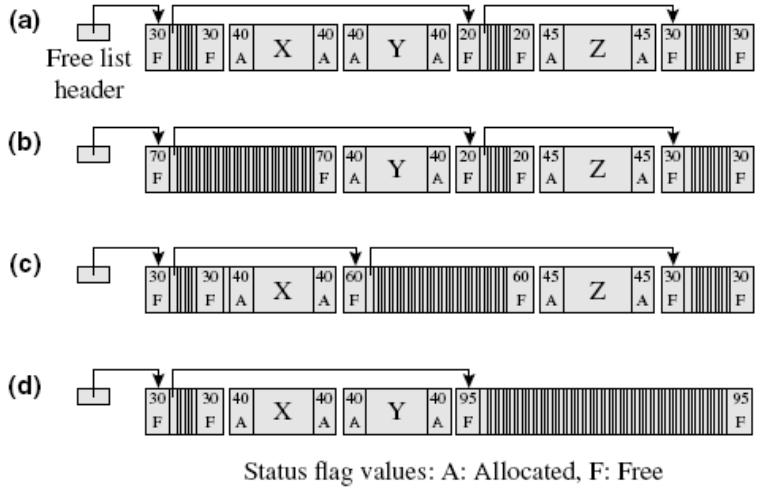


Figure 11.13 Merging using boundary tags: (a) free list; (b)–(d) freeing of areas X, Y, and Z, respectively.

Merging of Free Memory Areas (continued)

- The *50-percent rule* holds when merging is performed

Number of allocated areas, $n = \#A + \#B + \#C$
Number of free areas, $m = \frac{1}{2}(2 \times \#A + \#B)$
In the steady state $\#A = \#C$. Hence $m = n/2$

Merging of Free Memory Areas (continued)

- Memory compaction is achieved by "packing" all allocated areas toward one end of the memory
  - Possible only if a relocation register is provided

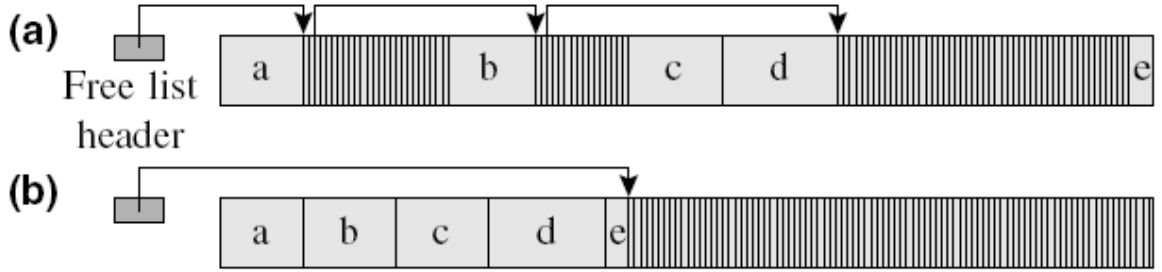


Figure 11.14 Memory compaction.

Buddy System and Power-of-2 Allocators

- These allocators perform allocation of memory in blocks of a few standard sizes
  - Leads to internal fragmentation
  - Enables the allocator to maintain separate free lists for blocks of different block sizes
    - Avoids expensive searches in a free list
    - Leads to fast allocation and deallocation
- Buddy system allocator performs restricted merging
- Power-of-2 allocator does not perform merging

Buddy System Allocator

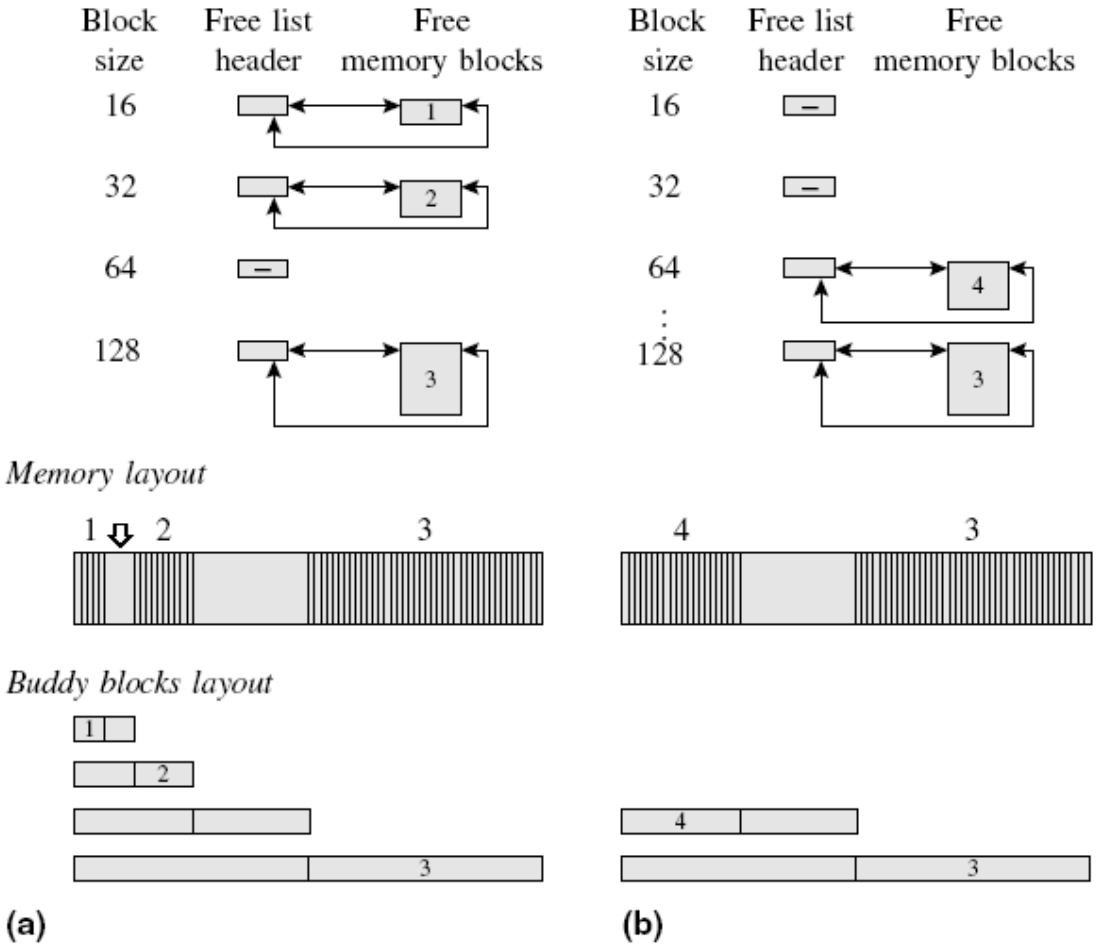


Figure 11.15 Buddy system operation when a block is released.

Power-of-2 Allocator

- Sizes of memory blocks are powers of 2
- Separate free lists are maintained for blocks of different sizes
- Each block contains a *header element*
  - Contains address of free list to which it should be added when it becomes free
- An entire block is allocated to a request
  - No splitting of blocks takes place
- No effort is made to coalesce adjoining blocks
  - When released, a block is returned to its free list

Heap Management in Windows

- Heap management aims at low allocation overhead and low fragmentation
  - By default, uses free list and best-fit allocation policy
    - Not adequate: (1) when process makes heavy use of heap, and (2) in a multiprocessor environment
  - Alternative: use the *low-fragmentation heap* (LFH)
    - Maintains many free lists; each for areas of a specific size
      - Neither splitting, nor merging is performed
    - Analogous to power-of-2 allocator
    - OS monitors requests and adjusts sizes to fine-tune performance

Contiguous Memory Allocation

- In contiguous memory allocation each process is allocated a single contiguous area in memory
  - Faces the problem of memory fragmentation
    - Apply techniques of memory compaction and reuse
      - Compaction requires a relocation register
      - Lack of this register is also a problem for swapping

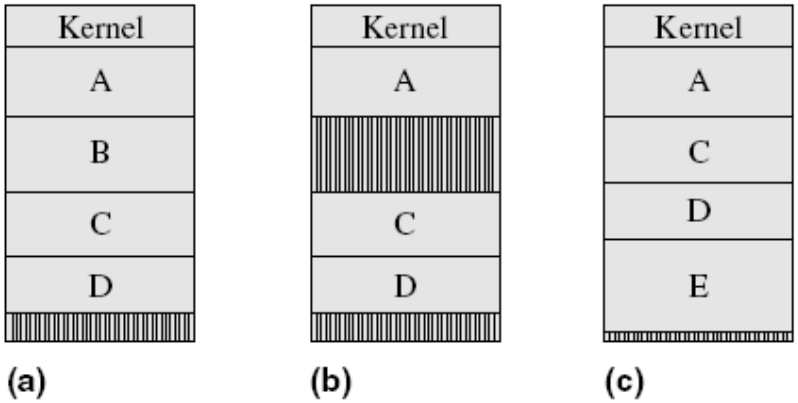


Figure 11.16 Memory compaction.

Non-contiguous Memory Allocation

- Portions of a process address space are distributed among different memory areas
  - Reduces external fragmentation

Process component	Size	Memory start address
P-1	50 KB	100K
P-2	30 KB	300K
P-3	60 KB	450K

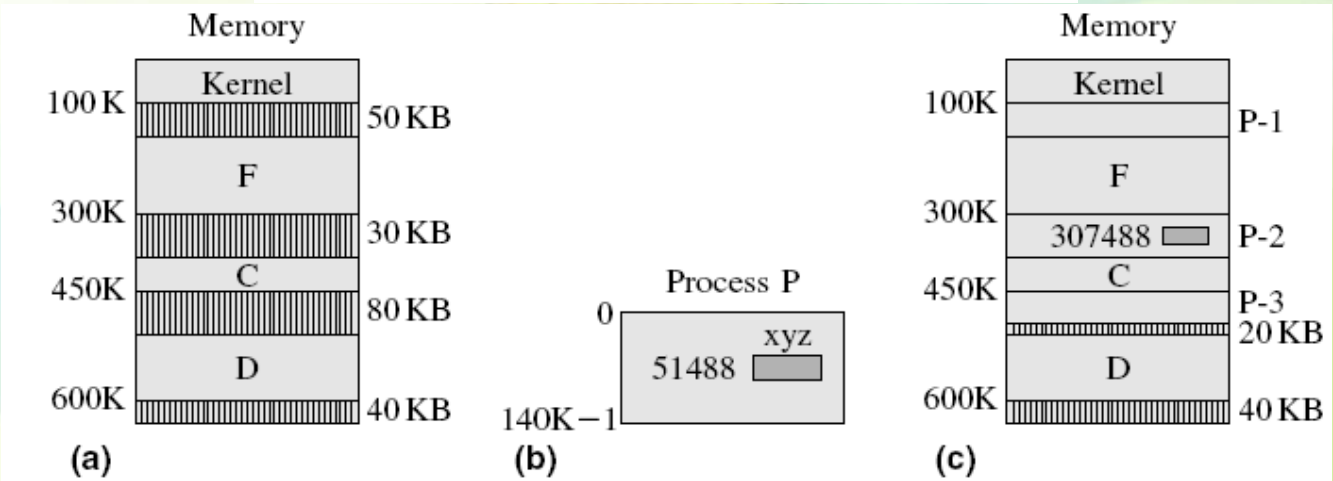


Figure 11.17 Noncontiguous memory allocation to process P.

Logical Addresses, Physical Addresses, and Address Translation

- *Logical address:* address of an instruction or data byte as used in a process
  - Viewed as a pair ($comp_i$, $byte_i$)
- *Physical address:* address in memory where an instruction or data byte exists

Effective memory address of ($comp_i$, $byte_i$)
= start address of memory area allocated to $comp_i$
+ byte number of $byte_i$ within $comp_i$

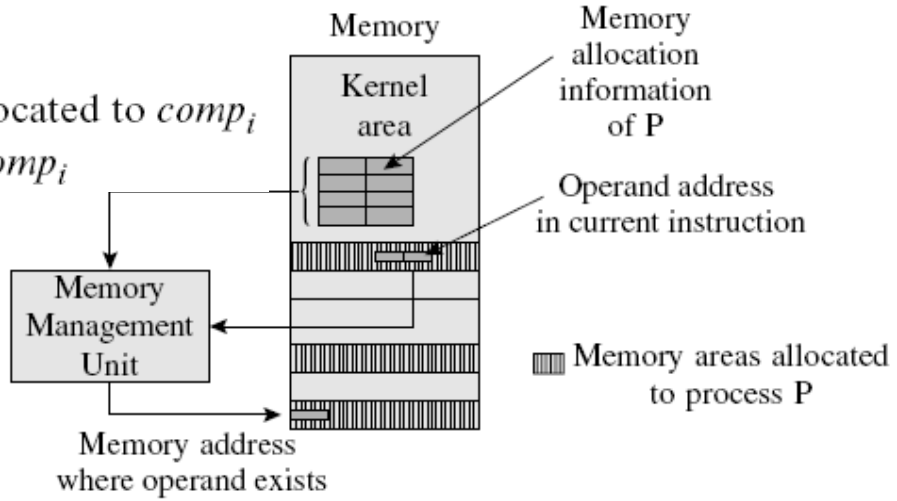


Figure 11.18 A schematic of address translation in noncontiguous memory allocation.

Approaches to Non-contiguous Memory Allocation

- Two approaches:
  - Paging
    - Process consists of fixed-size components called *pages*
    - Eliminates external fragmentation
    - The page size is defined by hardware
  - Segmentation
    - Programmer identifies logical entities in a program; each is called a *segment*
    - Facilitates sharing of code, data, and program modules between processes
- Hybrid approach: *segmentation with paging*
  - Avoids external fragmentation

Table 11.4 Comparison of Contiguous and Noncontiguous Memory Allocation

Function	Contiguous allocation	Noncontiguous allocation
Memory allocation	The kernel allocates a single memory area to a process.	The kernel allocates several memory areas to a process—each memory area holds one component of the process.
Address translation	Address translation is not required.	Address translation is performed by the MMU during program execution.
Memory fragmentation	External fragmentation arises if first-fit, best-fit, or next-fit allocation is used. Internal fragmentation arises if memory allocation is performed in blocks of a few standard sizes.	In paging, external fragmentation does not occur but internal fragmentation can occur. In segmentation, external fragmentation occurs, but internal fragmentation does not occur.
Swapping	Unless the computer system provides a relocation register, a swapped-in process must be placed in its originally allocated area.	Components of a swapped-in process can be placed anywhere in memory.

Memory Protection

- Memory areas allocated to a program have to be protected against interference from other programs
  - MMU performs this through a bounds check
    - While performing address translation for a logical address ($comp_i$, $byte_i$), MMU checks if $comp_i$ actually exists in program and whether $byte_i$ exists in $comp_i$
      - Protection violation interrupt raised if check fails
- Bounds check can be simplified in paging
  - $byte_i$ cannot exceed size of a page

Paging

- In the logical view, the address space of a process consists of a linear arrangement of pages
- Each page has *s* bytes in it, where *s* is a power of 2
  - The value of *s* is specified in the architecture of the computer system
- Processes use numeric logical addresses

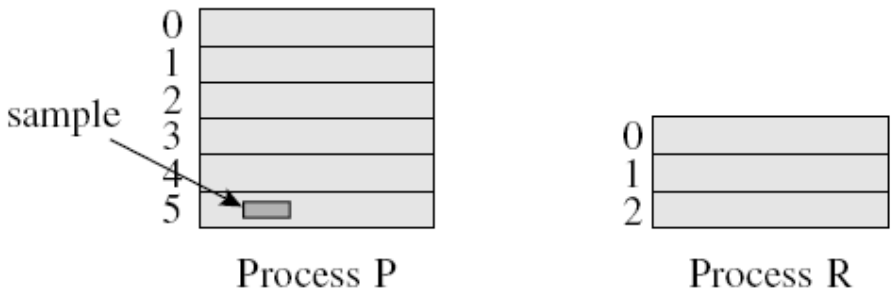


Figure 11.19 Logical view of processes in paging.

Paging (continued)

- Memory is divided into areas called page frames
- A page frame is the same size as a page

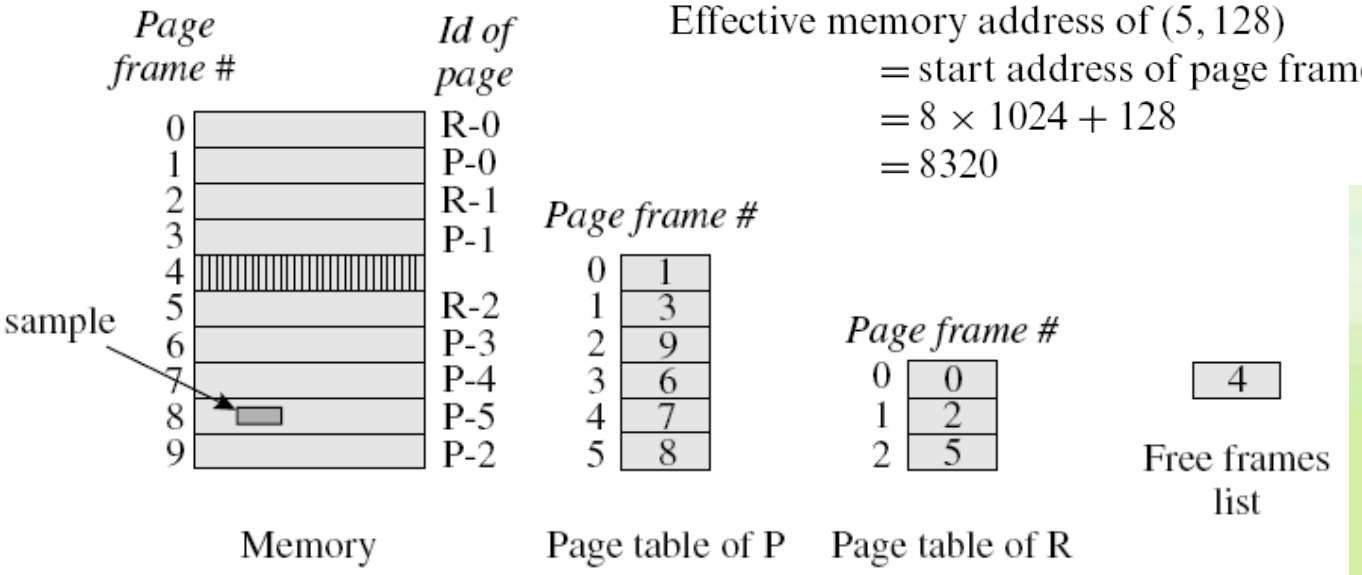
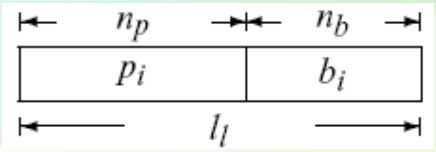


Figure 11.20 Physical organization in paging.

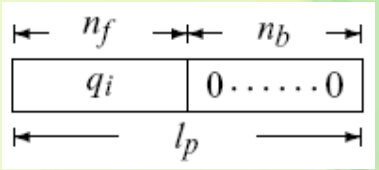
Paging (continued)

- Notation used to describe address translation:
  - s Size of a page
  - l_l Length of a logical address (i.e., number of bits in it)
  - l_p Length of a physical address
  - n_b Number of bits used to represent the byte number in a logical address
  - n_p Number of bits used to represent the page number in a logical address
  - n_f Number of bits used to represent frame number in a physical address
- The size of a page, s, is a power of 2
  - n_b is chosen such that $s = 2^{n_b}$

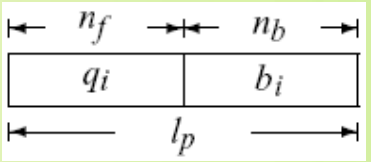
logical address



start address of frame q_i

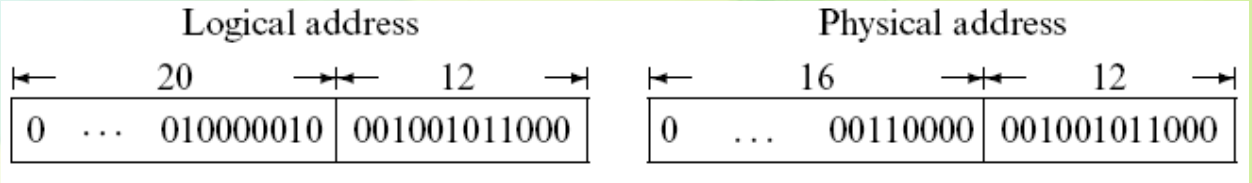


effective physical address



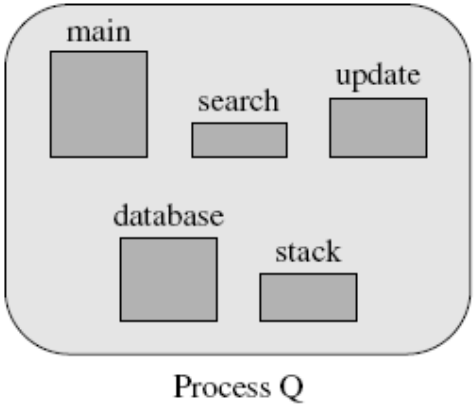
Example: Address Translation in Paging

- 32-bit logical addresses
- Page size of 4 KB
  - 12 bits are adequate to address the bytes in a page
    - 2^{12} = 4KB
- For a memory size of 256 MB, l_p = 28
- If page 130 exists in page frame 48,
  - p_i = 130, and q_i = 48
  - If b_i = 600, the logical and physical addresses are:



Segmentation

- A *segment* is a logical entity in a program
  - E.g., a function, a data structure, or an object



Name	*Size*	*Address*
main	476	23500
database	20240	32012
search	378	76248
update	642	91376
stack	500	54500

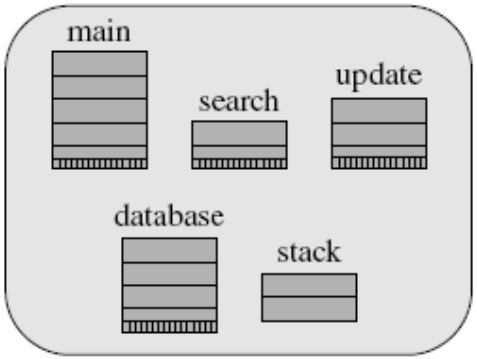
Segment table of Q

Figure 11.21 A process Q in segmentation.

- Each logical address used in Q has the form (s_i, b_i)
  - s_i and b_i are the ids of a segment and a byte within a segment

Segmentation with Paging

- Each segment in a program is paged separately
- Integral number of pages allocated to each segment
- Simplifies memory allocation and speeds it up
- Avoids external fragmentation



Process Q

Name	*Size*	*Page table address*
main	476	
database	20240	
search	378	
update	642	
stack	500	

Segment table of Q

Figure 11.22 A process Q in segmentation with paging.

Kernel Memory Allocation

- Kernel creates and destroys data structures at a high rate during its operation
  - Mostly *control blocks*
    - E.g., PCB, ECB, IOCB, FCB
  - Sizes of control blocks known in OS design stage
    - Helps make memory allocation simple and efficient
- Modern OSs use noncontiguous memory allocation with paging
  - McKusick–Karels allocator
  - Lazy buddy allocator
  - Slab allocator

Kernel Memory Allocation (continued)

- McKusick–Karels and lazy buddy allocators allocate memory areas that are powers of 2 in size *within* a page
  - Start address of each allocated memory area of size 2^n is a multiple of 2^n
    - Boundary alignment on a power of 2
      - Leads to a cache performance problem
      - Some parts of the cache face a lot of contention leading to poor cache performance of kernel code
- Slab allocator uses an interesting technique to avoid this cache performance problem

Kernel Memory Allocation (continued)

- Slab allocator was first used in Solaris 2.4
  - Has been used in Linux since version 2.2
- A *slab* consists of many *slots;* each can hold an object
  - Coloring areas are chosen such that objects in different slabs of pool have different alignments with respect to the closest multiples of a power of 2
    - Map into different areas of a set-associative cache

Using Idle RAM Effectively

- Memory is idle when applications are not active
- How can idle memory be exploited by OS?
  - Run utilities during idle periods of a computer
    - E.g., antivirus software
    - Can have a negative impact on performance by displacing applications from memory
  - Windows Vista uses techniques that use idle RAM to enhance system performance
    - *SuperFetch: preloads frequently used applications in idle RAM*
    - *Readyboost: uses USB drive as a cache between disk and RAM*

Summary

- Compiler assumes a specific memory area to be available to program and generates *object module*
- *Linker* performs *relocation* of a program, and performs *linking* to connect the program with library functions
- *Self-relocating* programs perform their own relocation
- CPU has a *relocation register* to facilitate relocation
- Memory allocation can be: *static* or *dynamic*
  - Both combined in programs through *stack* and *heap*

Summary

- Allocation/deallocation of memory can lead to fragmentation: *internal* or *external*
  - *First-fit, next-fit* and *best-fit strategies* try to reduce fragmentation
  - *buddy systems* and *power-of-2 allocators* eliminate external fragmentation
  - *Noncontiguous allocation* reduces external fragmentation
    - Requires use of the memory management unit (MMU) of CPU
- Kernel creates and destroys data structures at high rate
  - Uses special techniques to make memory reuse fast and efficient

Queries ...?