memory management

summary

- goals and requirements
- techniques that do not involve virtual memory

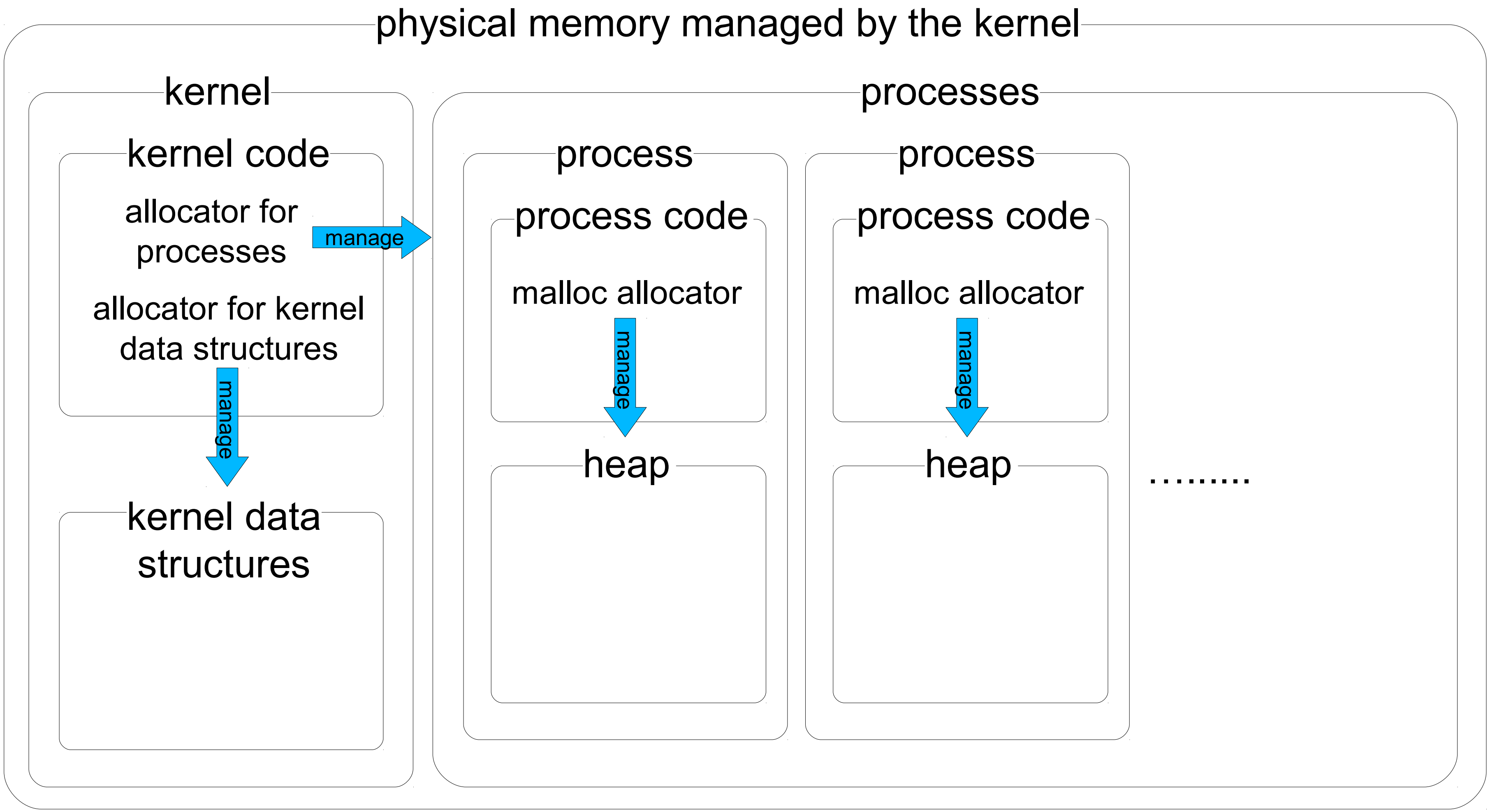
memory management

- tracking used and free memory
- primitives
  - **allocation** of a certain amount of memory
  - **de-allocation** of what allocated,
    - it permits reuse of de-allocated memory
- reason for **allocation request**
  - data structures (e.g. array, objects, ecc.)
    - kernel data structures (allocator implemented in the kernel)
    - process data structures (allocator implemented by language runtime libraries, e.g. C/C++ malloc)
  - processes (within an O.S.)

summary and applicability

- many techniques and concepts in memory management equally apply to memory allocation for processes and for data
  - fixed partitioning, dynamic compaction, fragmentation, placement algorithms, buddy system
  - the book talks about a “process” but it may be any kind of allocation request
- hardware supported techniques apply only to processes
  - virtual memory, paging, segmentation

kinds of memory and allocators



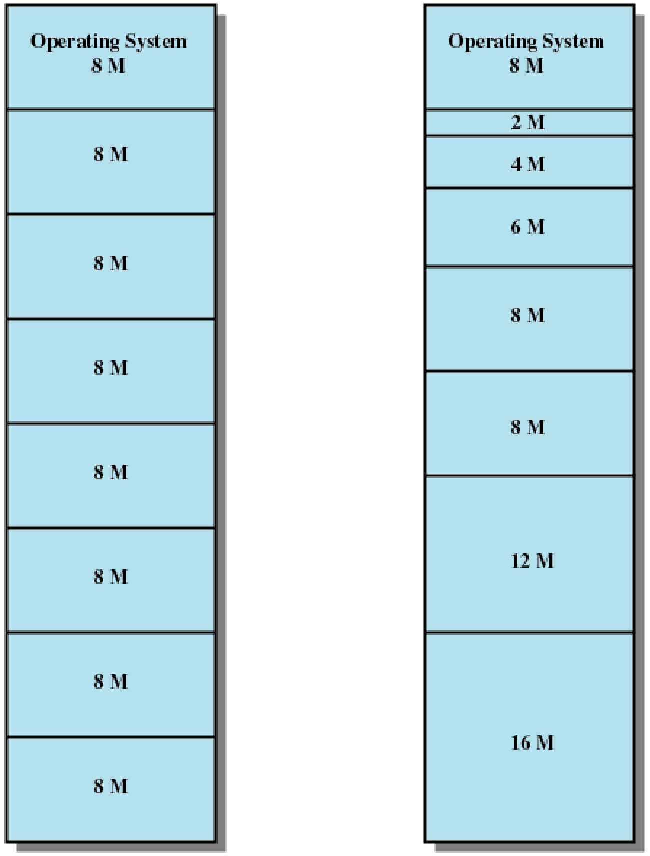
allocators inventory

- in the processes
  - heap managed by malloc
    - allocate data structures for the process
- in the kernel
  - “sort of heap” managed by a “sort of malloc” in the kernel
    - allocate data structures for the kernel
    - remember that the kernel cannot use libraries!
    - in linux this is provided by a buddy-system plus a “slab allocator”
  - allocation of images of the processes
    - for old OSes adopt the same approaches for data structures
    - in modern OSes relies on paging and virtual memory

memory management techniques that do not involves virtual memory

Fixed Partitioning

- memory is partitioned in a fixed way
- equal size
- unequal size



(a) Equal-size partitions

(b) Unequal-size partitions

Fixed Partitioning

- inefficient memory use
  - any program, no matter how small, occupies an entire partition.
- the fact that same space within a partition is wasted is called **internal fragmentation**.

Fixed Partitioning

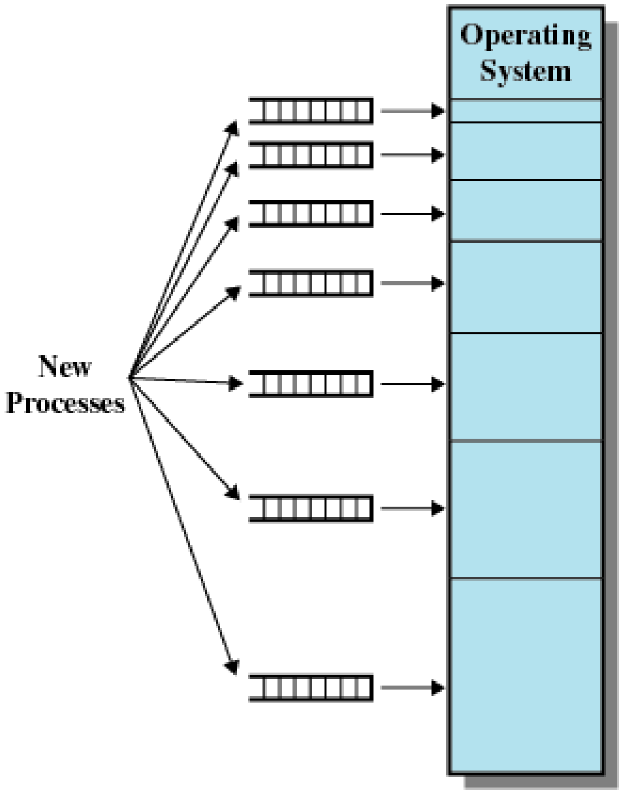
- Equal-size partitions
  - Any process or data whose size is less than or equal to the partition size can be loaded into an available partition
  - If all partitions are full, the operating system can swap a process out of a partition
  - A process/data may not fit in a partition.
    - For processes, the programmer must design the program with overlays
  - still used in hard disk partitioning
    - LVM overcome such limitation (linux)

Placement Algorithm with Partitions

- Equal-size partitions
  - Because all partitions are of equal size, it does not matter which partition is used

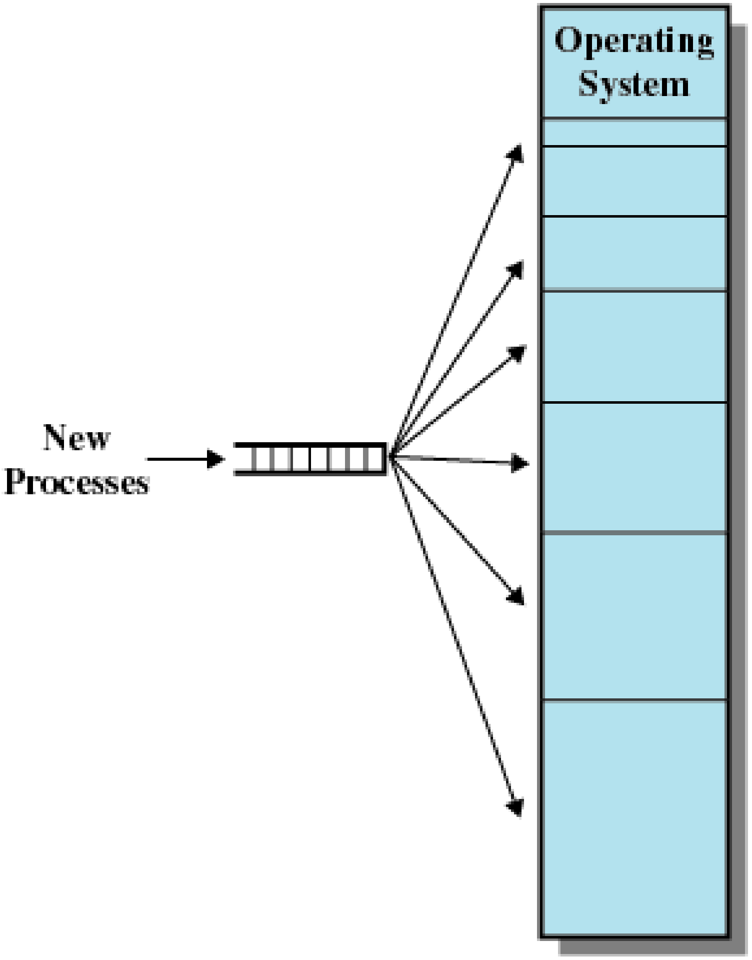
Placement Algorithm with Partitions

- Unequal-size partitions
  - minimize internal fragmentation
    - assign each process/data to the smallest partition it will fit into
    - one queue for each partition: a process might wait until it "best fit" partition is free, even if there are other partitions availableminimize wait time



Placement Algorithm with Partitions

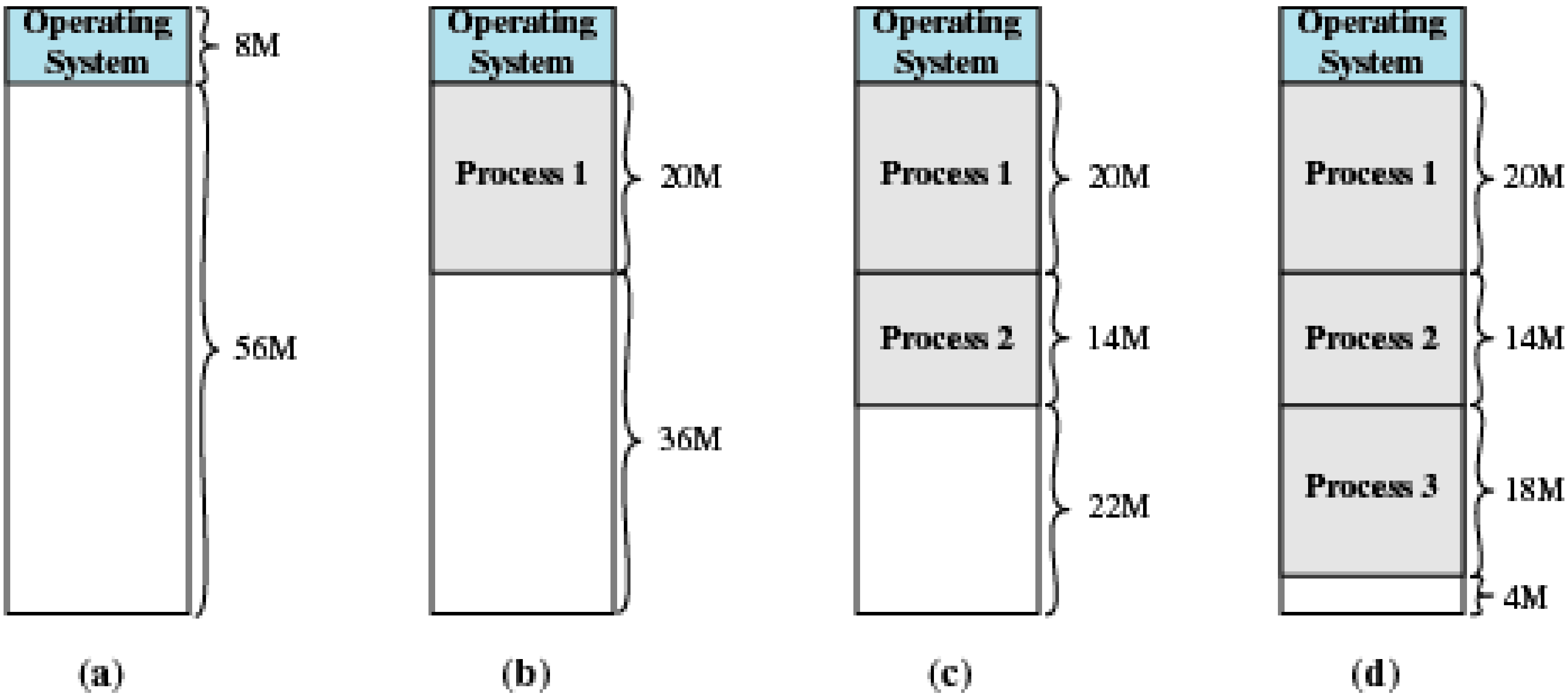
- Unequal-size partitions
  - minimize wait time and, *secondarily*, internal fragmentation
    - one single queue
    - request is assigned to the best partition available when served



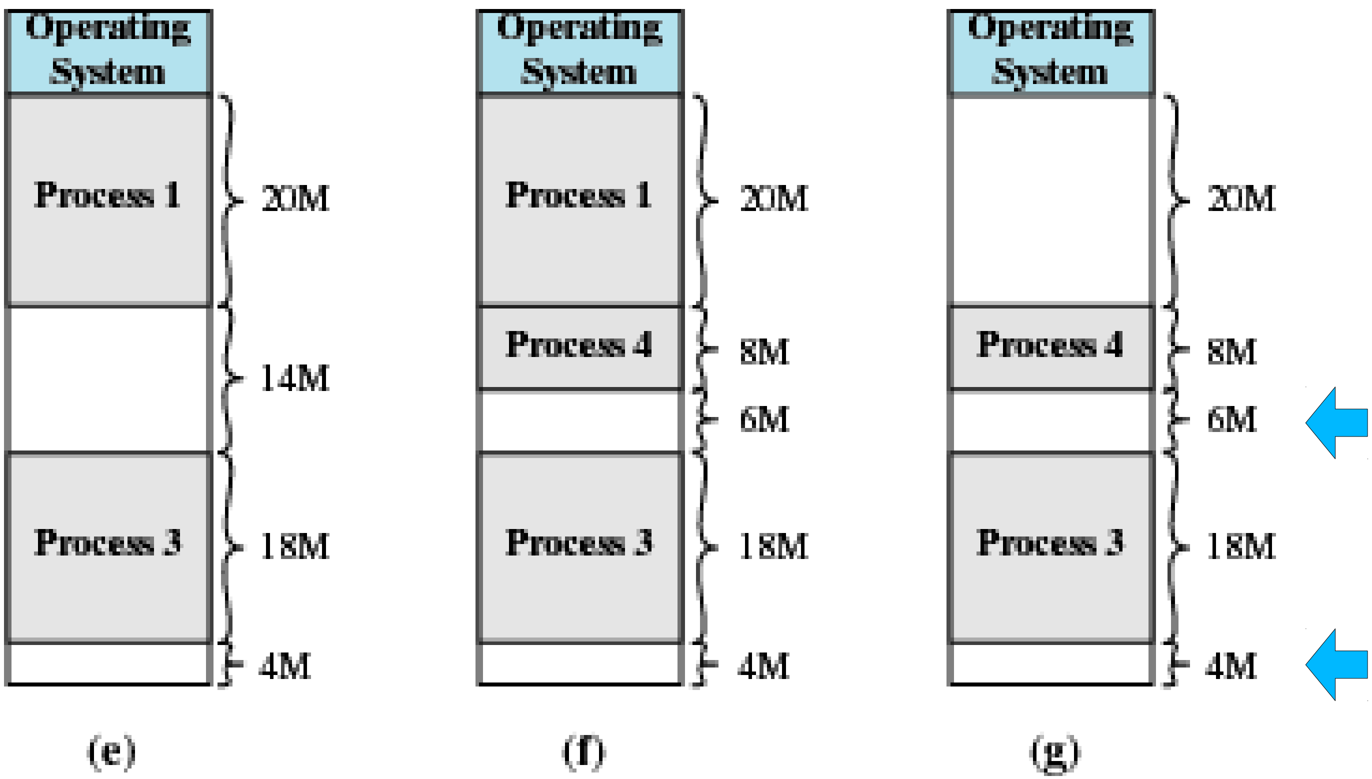
dynamic partitioning

- partitions are of variable length and number
- process/data is allocated exactly as much memory as required
- eventually, small holes in the memory remain. This is called **external fragmentation**

external fragmentation



external fragmentation



compaction

- it is a solution for external fragmentation
- compaction shifts allocated blocks so they are contiguous and all free memory is in one block
  - in the general case compaction is unfeasible
  - e.g. for C/C++ memory allocators: need for re-directing all pointers
    - but location of pointers is unknown!
    - tracking and redirecting pointers is inefficient
    - C/C++ are designed to be very very efficient
- so compaction is never used, all dynamic allocation systems stand with external fragmentation

Dynamic Partitioning Placement Algorithm

- allocators must decide which free block to allocate to an allocation request
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
    - since smallest block is found for the request, the smallest amount of fragmentation is left
  - Memory compaction must be done more often

Dynamic Partitioning Placement Algorithm

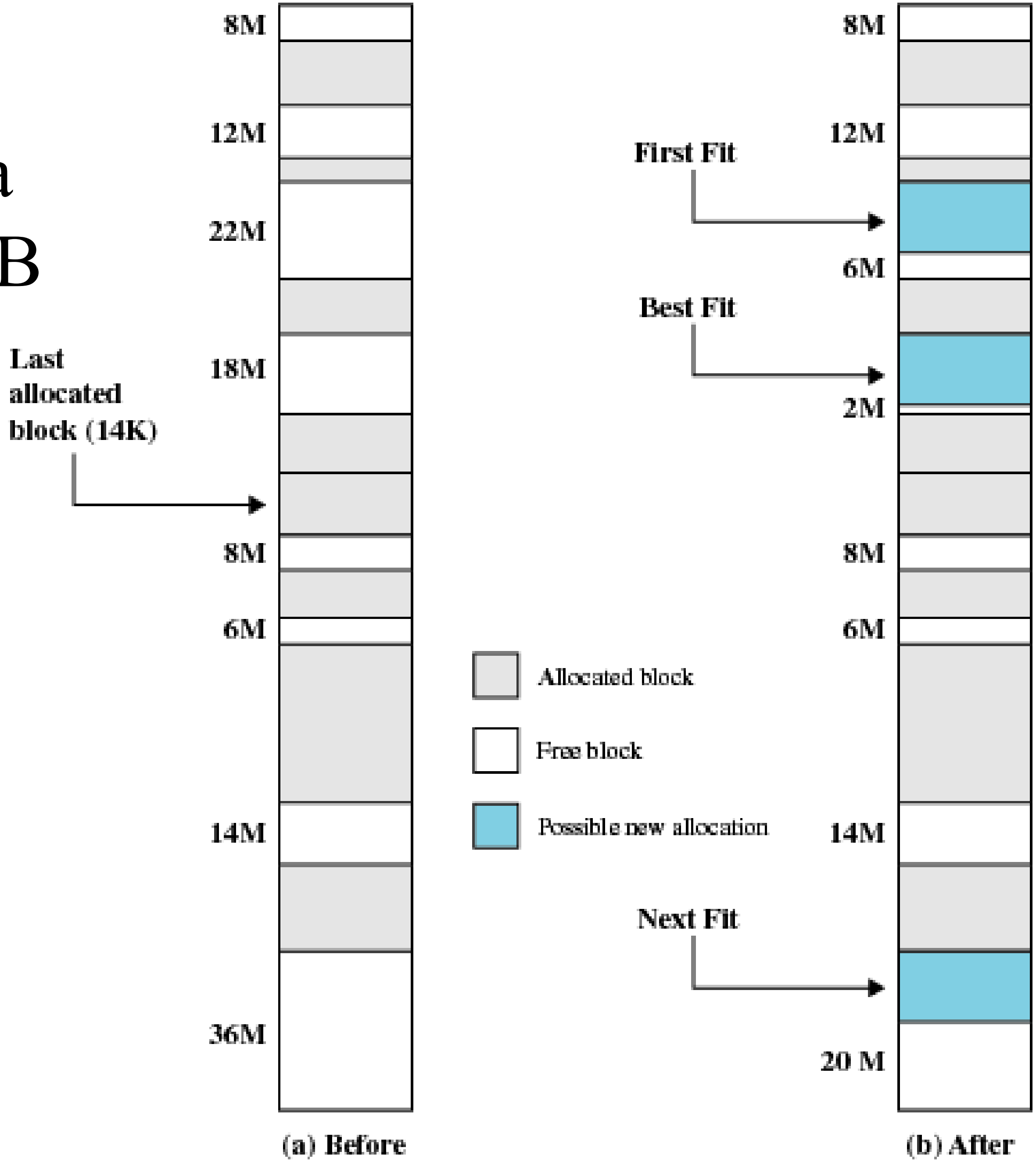
- First-fit algorithm
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many requestes loaded in the front end of memory that must be searched over when trying to find a free block

Dynamic Partitioning Placement Algorithm

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

examples

- allocation of a block of 16MB



Buddy System

- simple but powerful allocator
- widely used in O.S. to allocate large chunks of fixed size
  - e.g. 4KB pages in many architectures (x86_32)
- it can be used as a base for a more fine-grained allocator
  - which is called slab allocator in Linux and Solaris and is used for kernel data structures

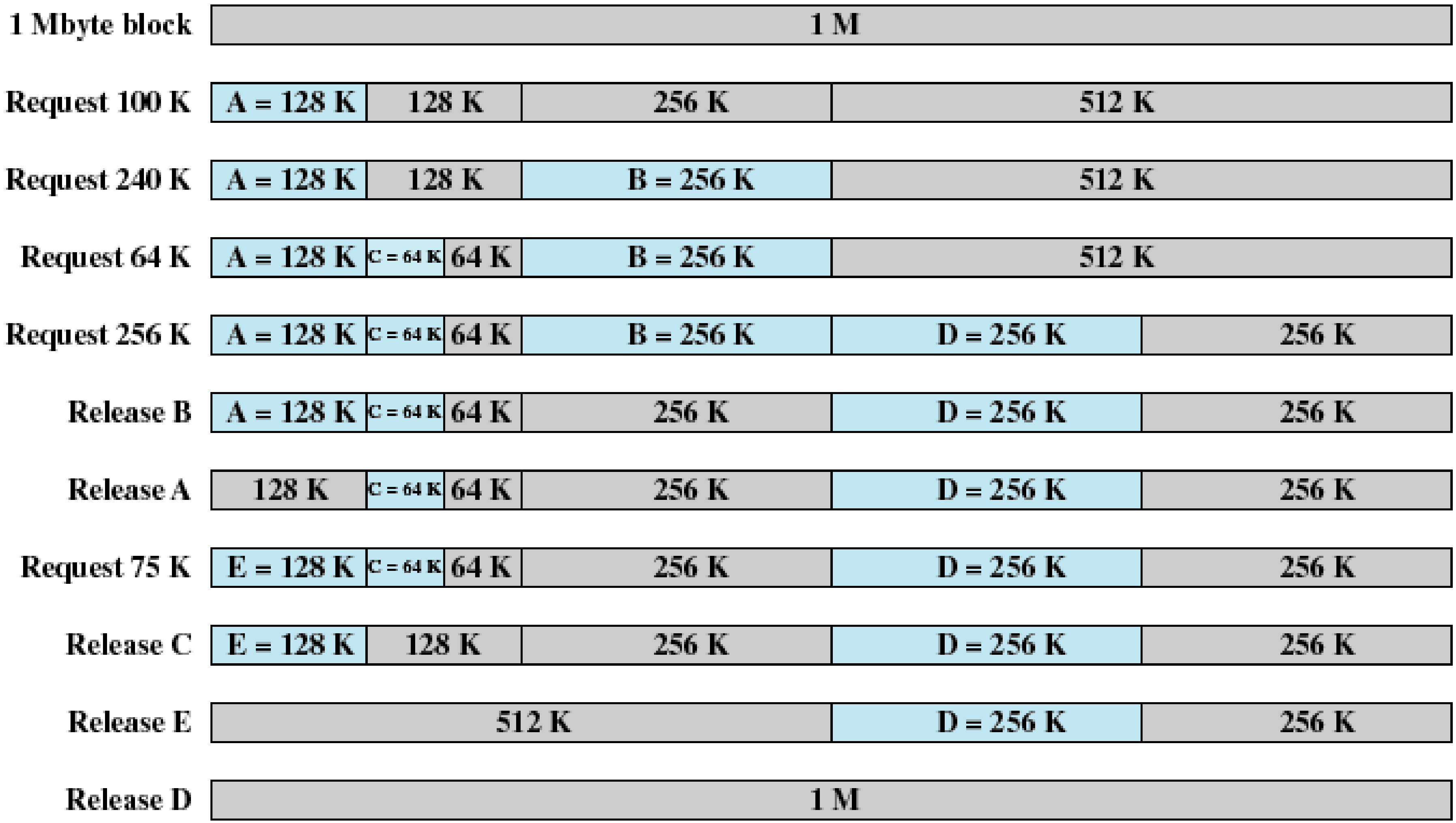
Buddy System

- entire space available is treated as a single block of 2^U
- a request of *s* bytes returns a block of ceil($\log_2 s$) bytes
  - if a request of size *s* such that $2^{i-1} < s <= 2^i$, a block of length 2^i is allocated
  - a 2^i block can be split into two equal **buddies** of 2^{i-1} *bytes*
  - for each request a “big” block is found and split until the smallest block greater than or equal to *s* is generated

Buddy System

- it maintains a lists L_i ($i=1..U$) of unallocated blocks (*holes*) of size 2^i
  - splitting: remove a hole from L_{i+1} split it, and put the two buddies it into L_i
  - coalescing: remove two unallocated buddies from L_i and put it into L_{i+1}

buddy system: example



Buddy System

procedure **get_hole**

- input: i (precondition: $i \leq U$)
- output: a block c of size 2^i (postcondition: L_i does not contain c)

if (L_i is empty)

b= get_hole(i+1);

< split b into two buddies b_1 and b_2>

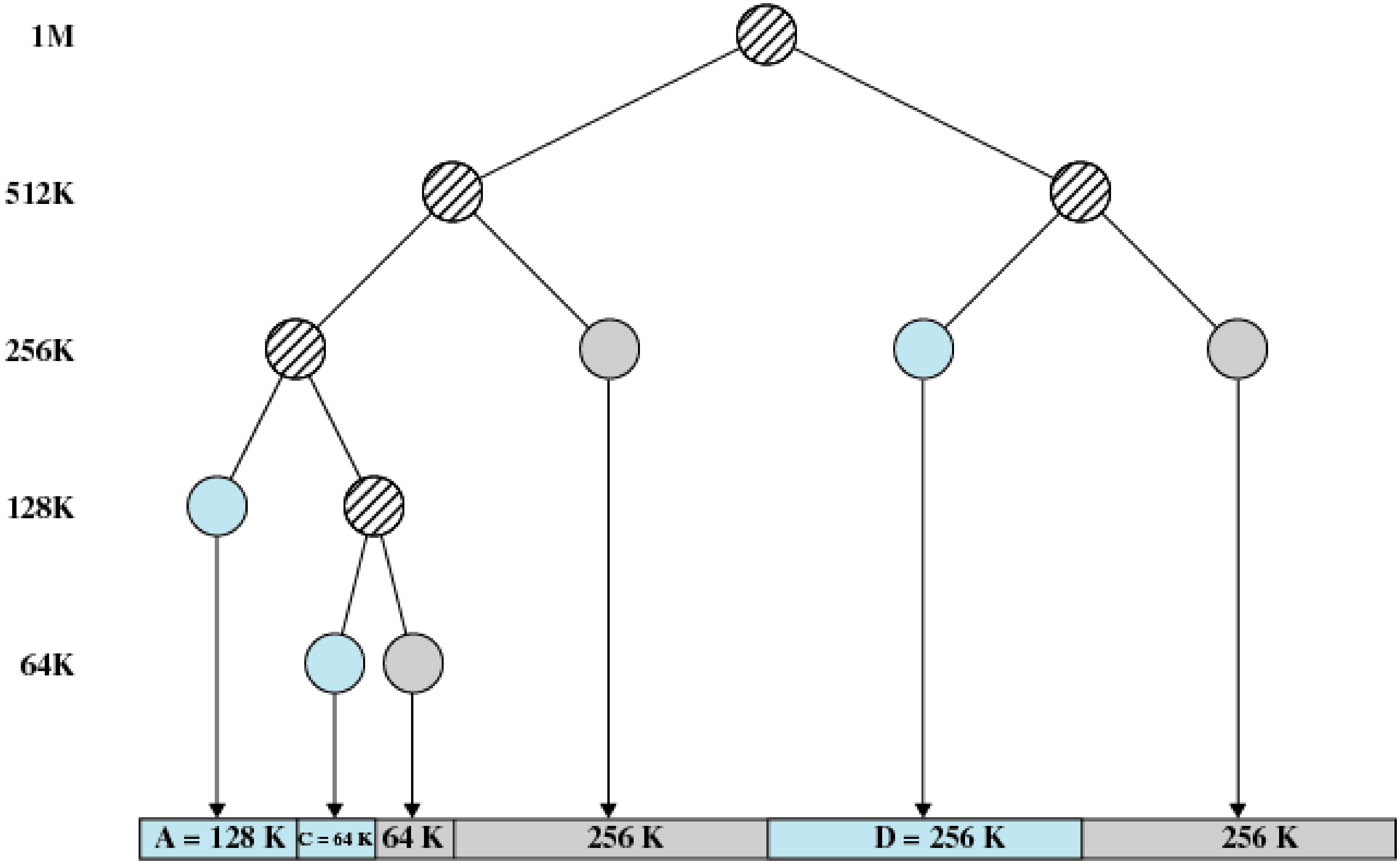
< put b_1 and b_2 into L_i>

c= < first hole in L_i>

<remove c form L_i>

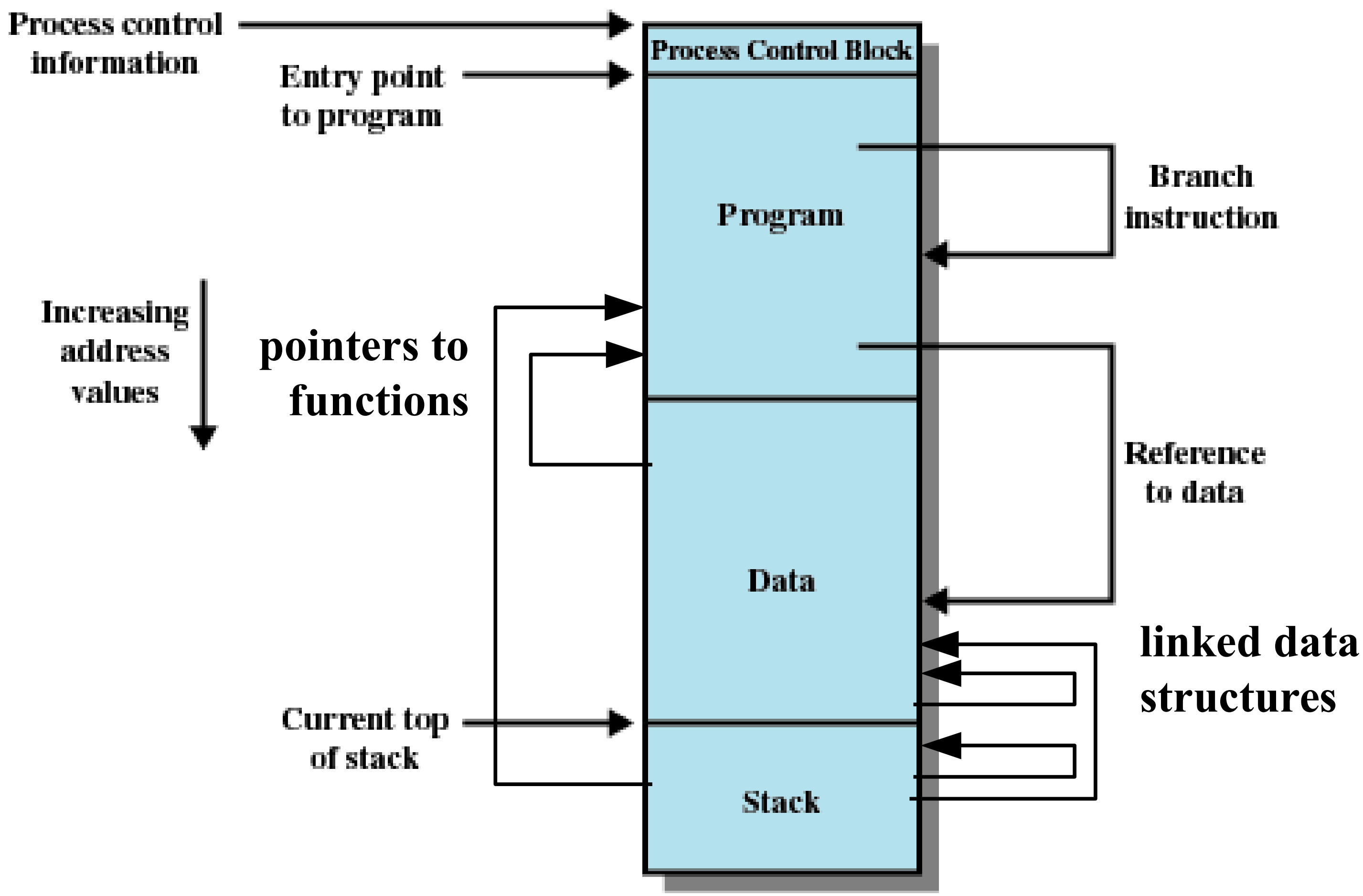
return c

buddy system: tree representation



memory requirements for processes

pointers in processes



relocation for processes
(without hw support)

- when a program is loaded into memory the absolute memory locations are determined
  - different execution may lead to different locations
  - memory references in the code must be translated to actual physical memory address
    - before run or on-the-fly
- on-the-fly relocation during execution
  - swap out and swap in
  - compaction of allocated partitions
- this kind of relocation is part of the linking phase

protection

- processes should not be able to reference memory locations in another process without permission
- references must be checked at run time
  - impossible to check memory references at compile time (may directly depend on the input)
  - exercise: given a generic input and program prove that reference check is not computable! (reduce stopping problem to it)
- memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
  - Operating system cannot anticipate all of the memory references a process will perform

sharing

- allow several processes to access the same portion of memory
- better to allow each process access to the same copy of the program rather than have their own separate copy

logical organization

- programs are written in modules
  - sw engineering reasons: divide the responsibility for development, maintenance, testing, ecc
- modules can be written and compiled independently
- different degrees of protection given to modules (read-only, execute-only)
- share modules among processes

physical organization

- memory available for a program plus its data may be insufficient
  - overlaying allows various modules to be assigned the same region of memory
- programmer does not know how much memory will be available

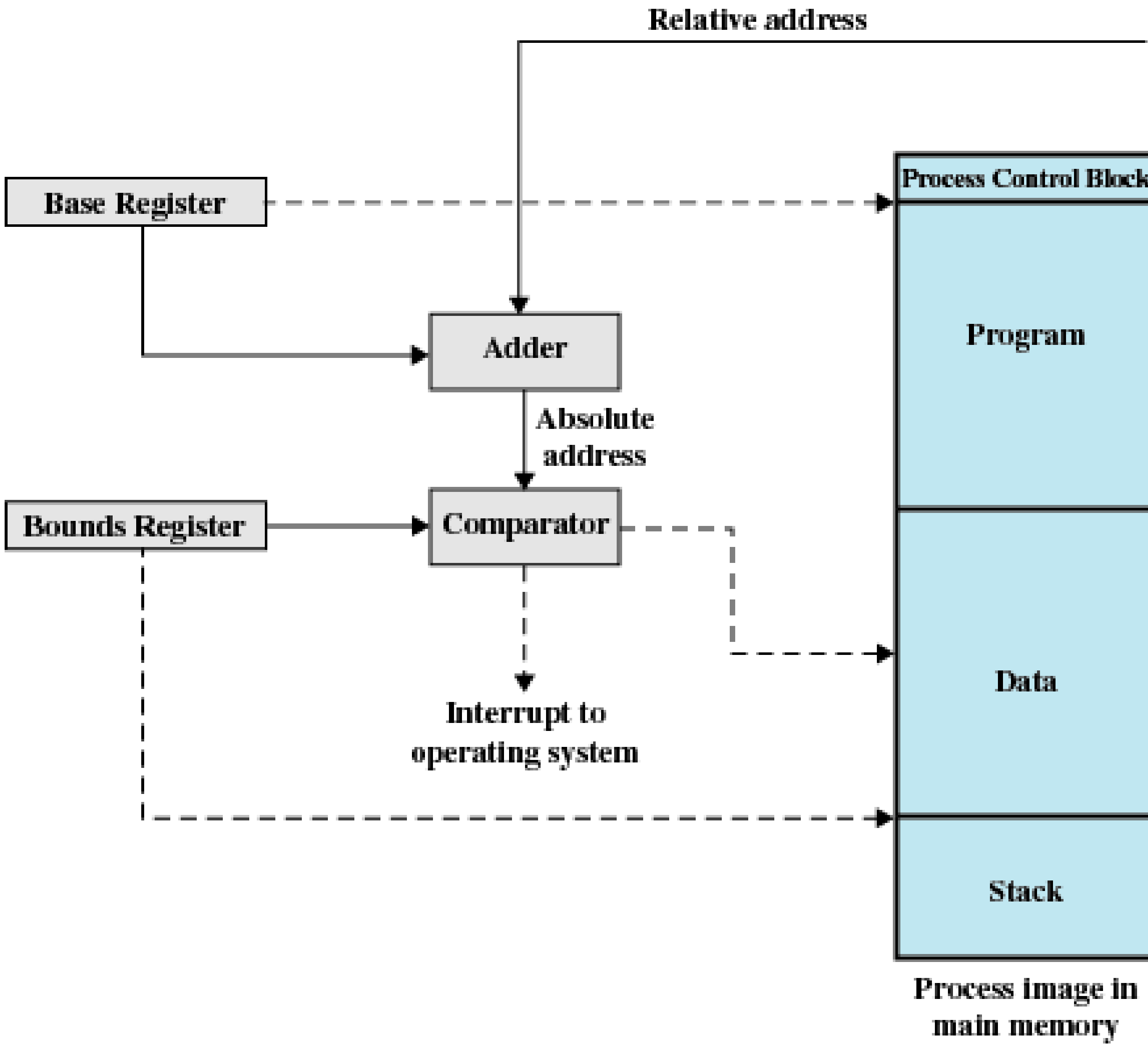
addresses in the program

- Physical
  - The absolute address or actual location in main memory
- Logical
  - Reference to a location in a “logical” memory independent of the current assignment of data to memory
  - Translation must be made to the physical address by the hardware (MMU)
- Relative (logical or physical)
  - Address expressed as a location relative to some known point

hardware support for relocation

- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in

hardware support for relocation



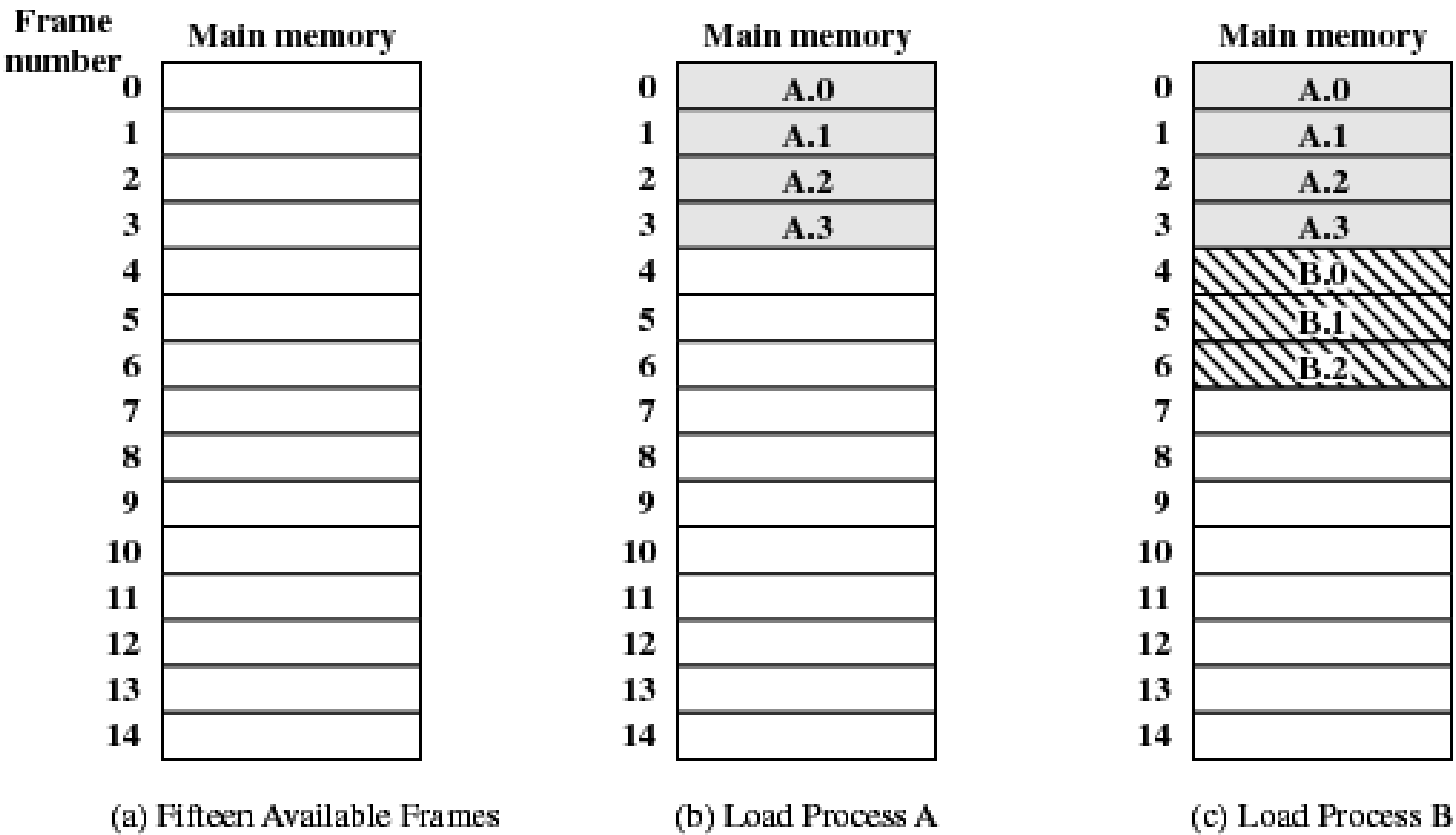
hardware support for relocation

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system
- If the address is ok it is used to access memory
- relocation is performed by setting appropriate value in the registers

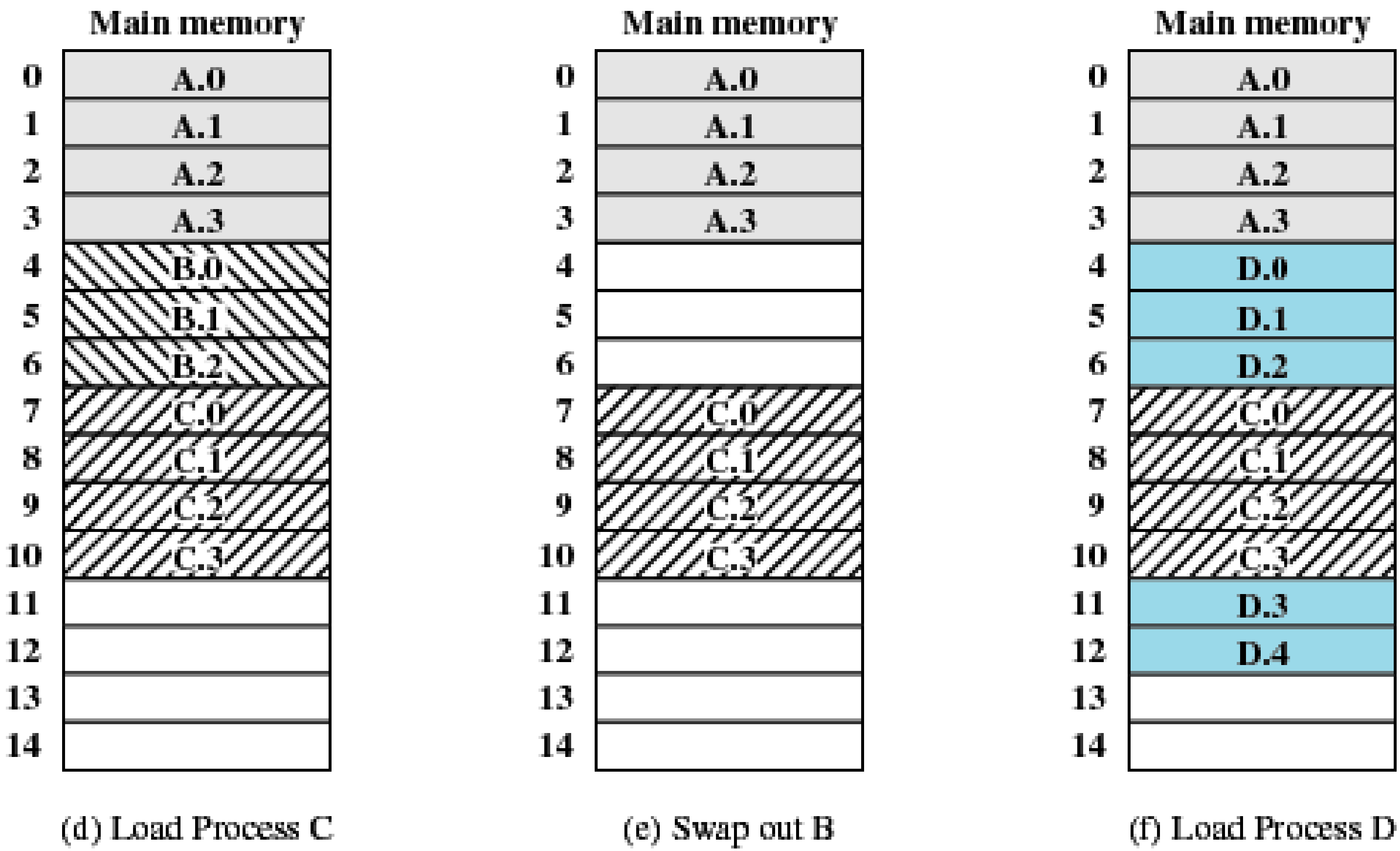
Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called **pages** and chunks of memory are called **frames**
- Operating system maintains a **page table** for each process
  – Contains the **frame location for each page** in the process
  – **Memory address consist of a page number and offset within the page**

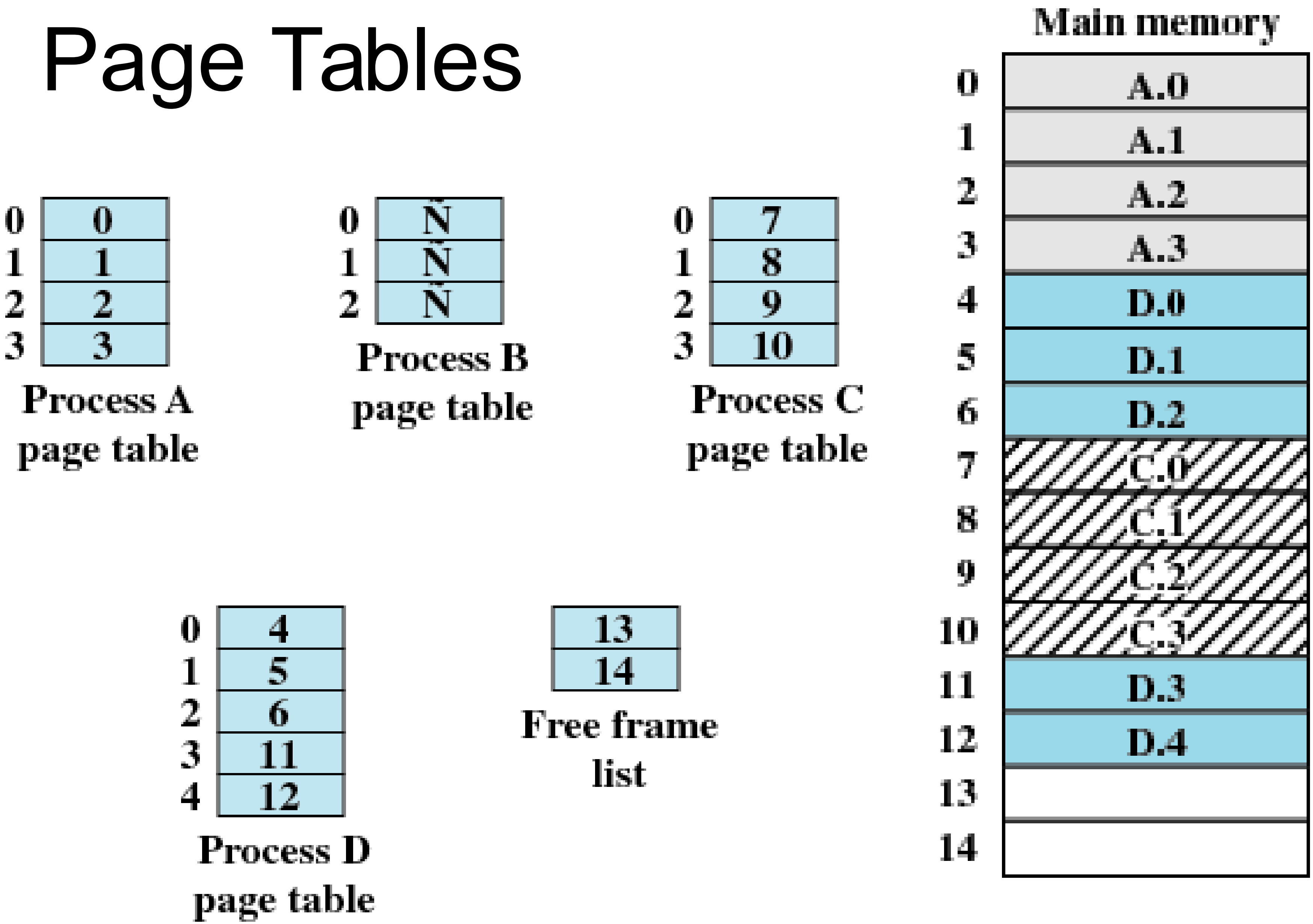
Assignment of Process Pages to Free Frames



Assignment of Process Pages to Free Frames



Page Tables

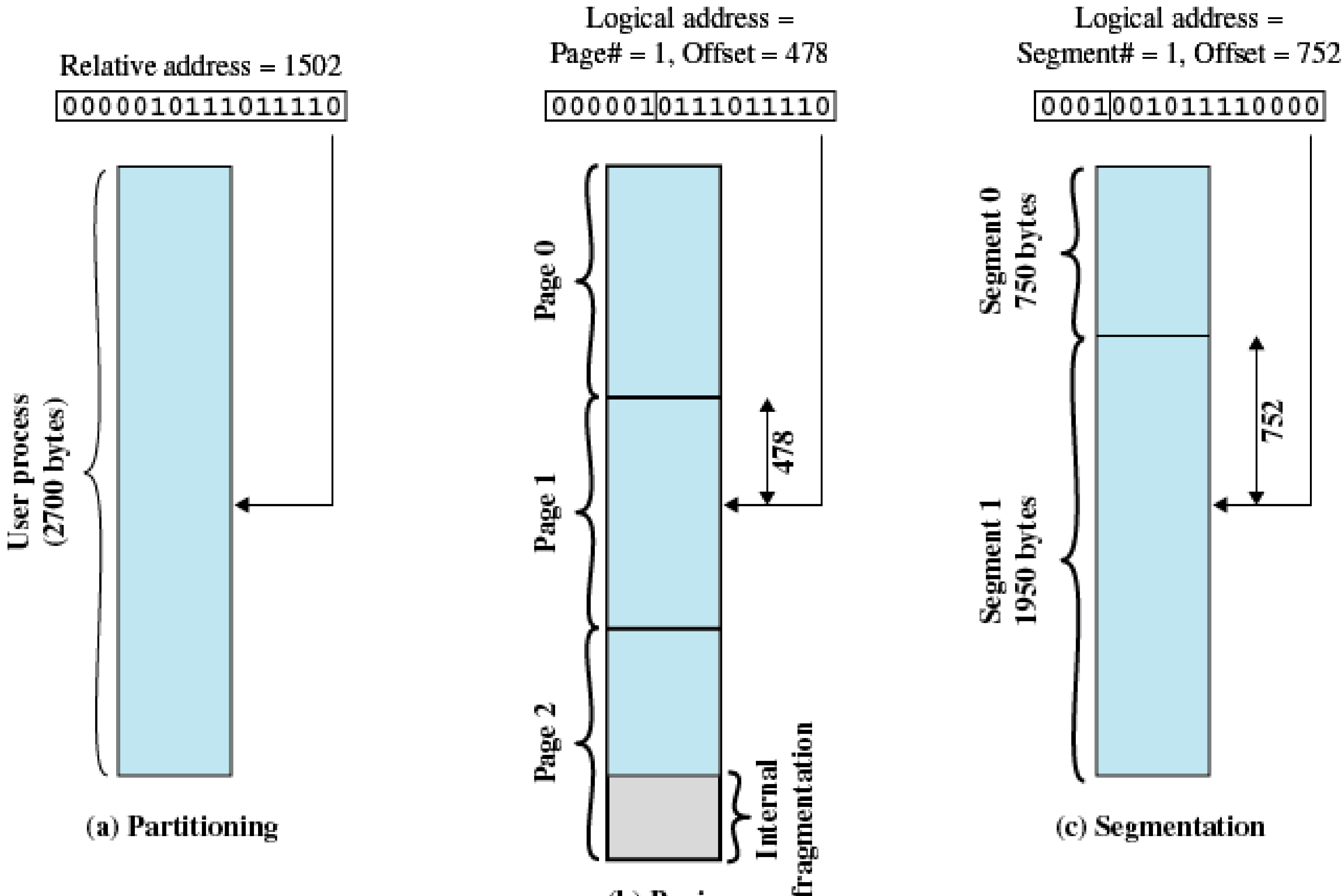


(f) Load Process D

Segmentation

- All segments of all programs do **not** have to be of the **same length**
- There is a maximum segment length
- Addressing consist of two parts - a **segment number and an offset**
- Since segments are not equal, segmentation is similar to dynamic partitioning

paging vs. segmentation

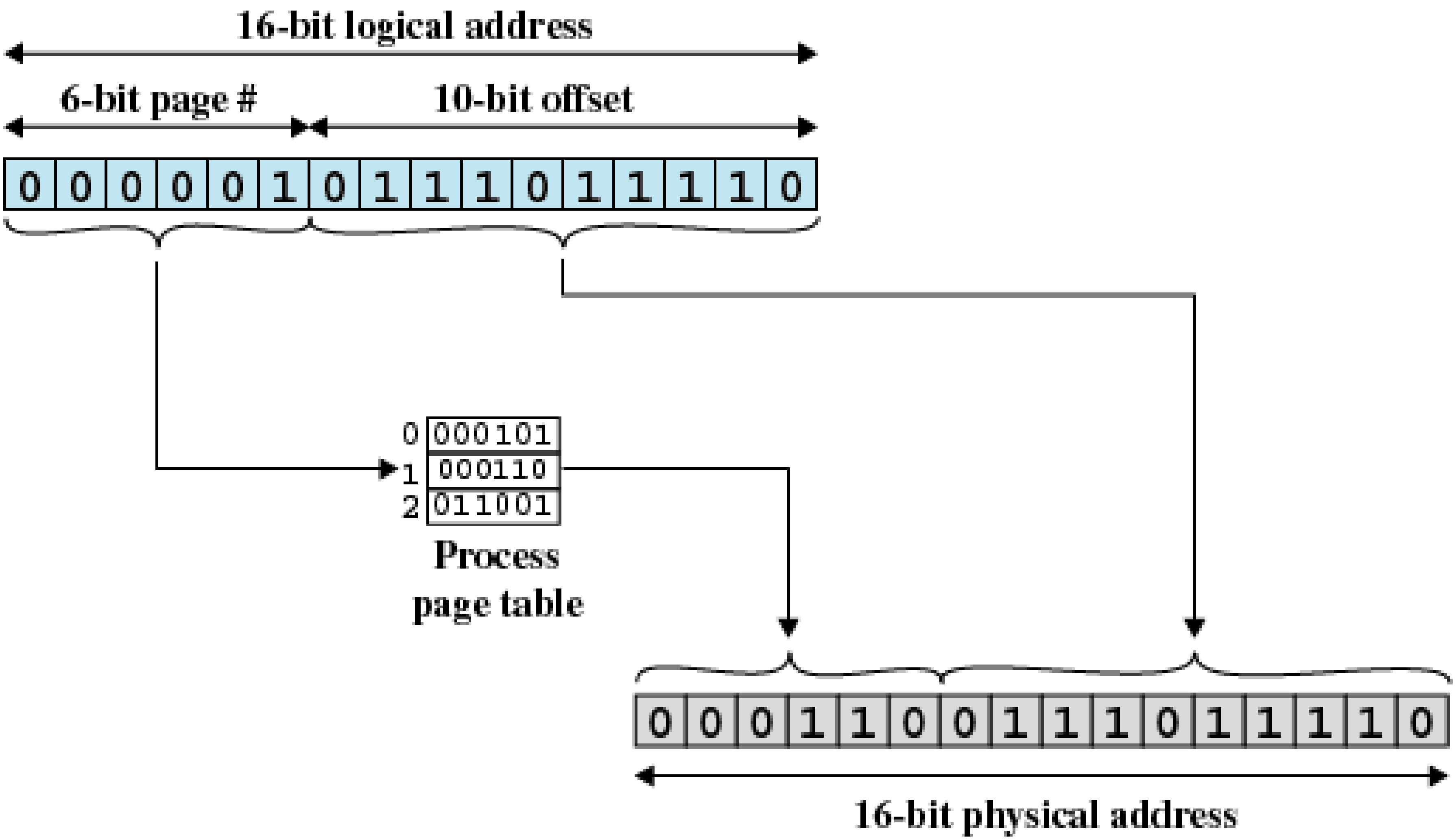


(a) Partitioning

(b) Paging (page size = 1K)

(c) Segmentation

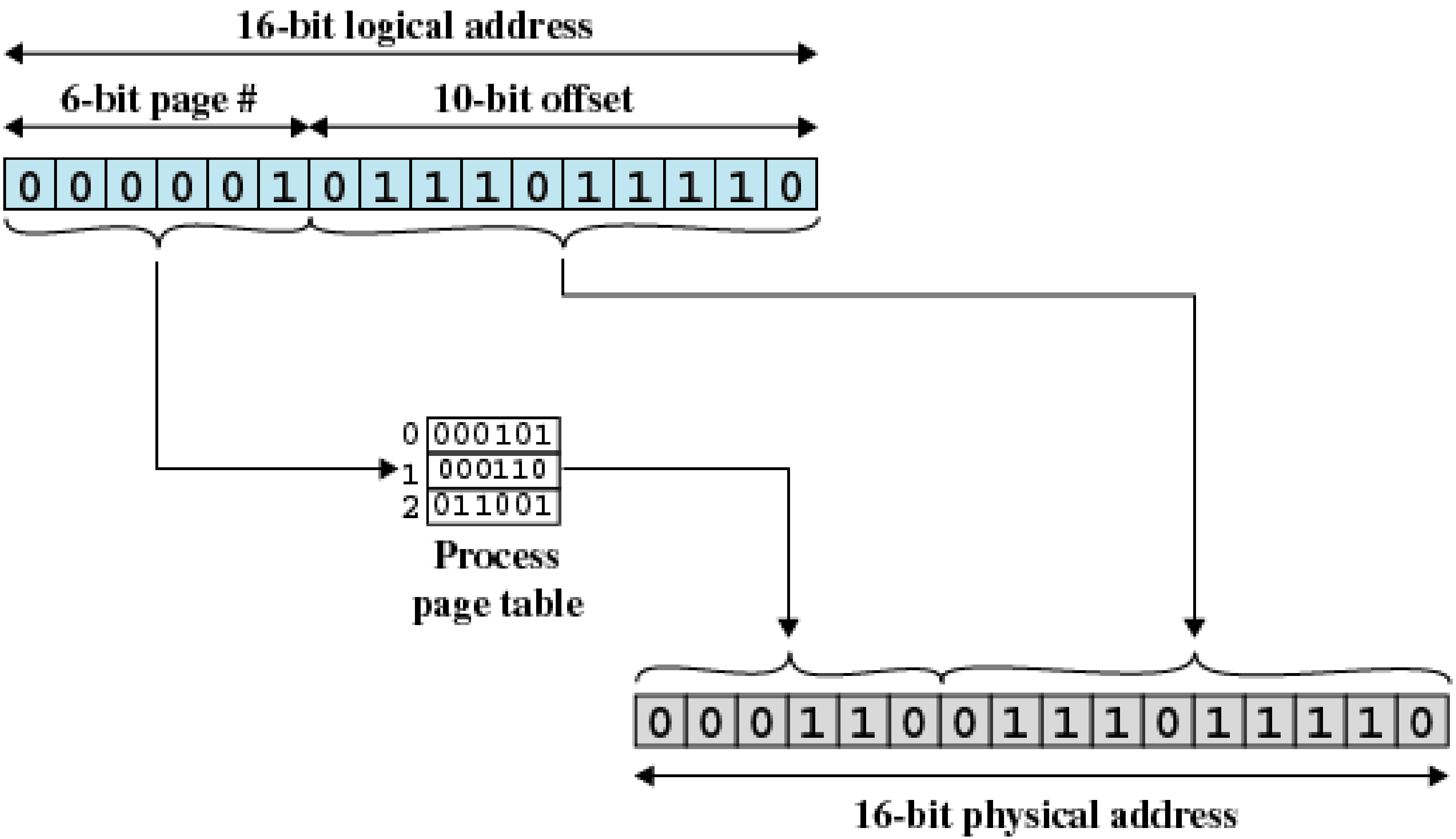
logical to physical translation
paging



(a) Paging

logical to physical translation

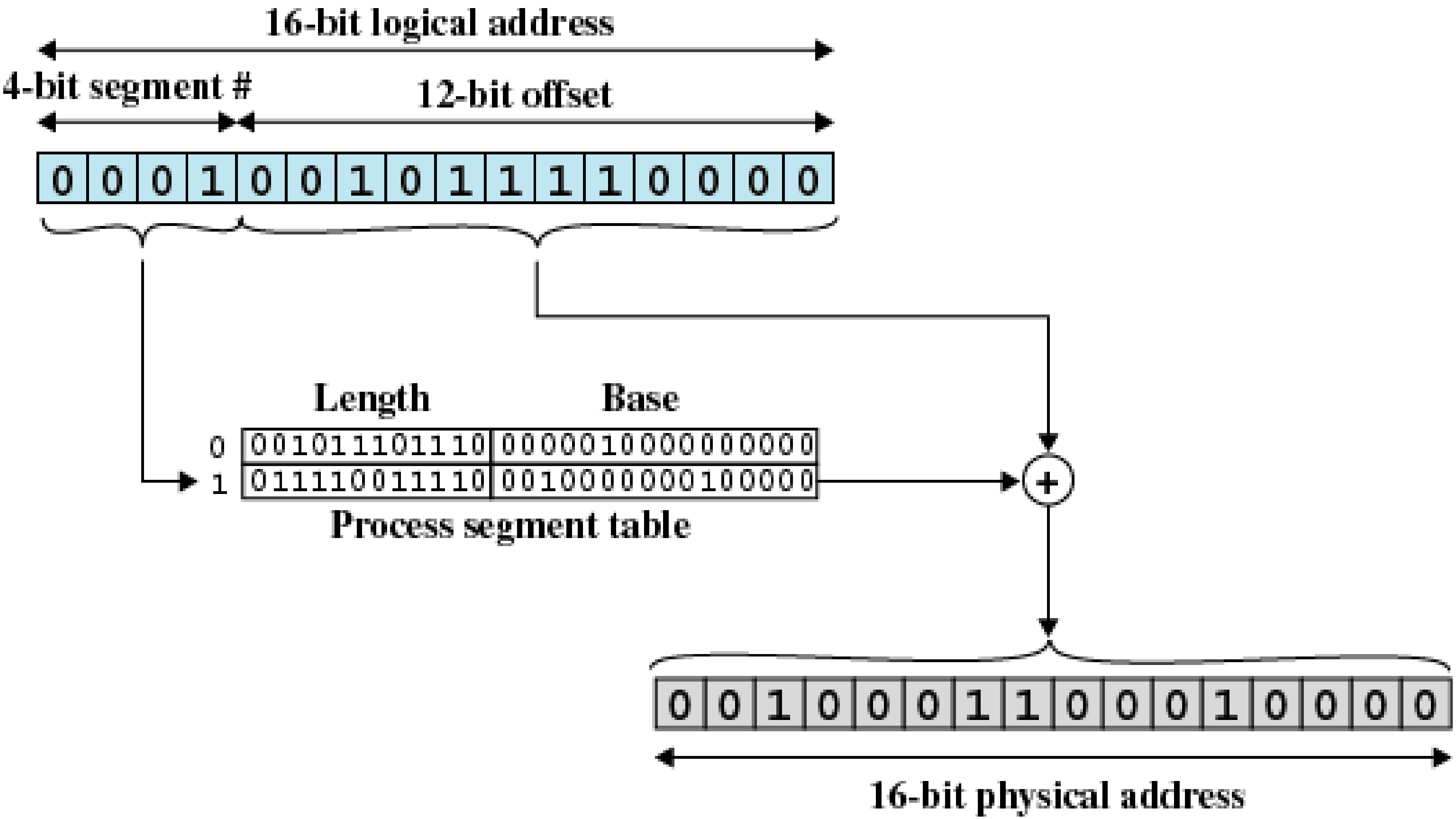
paging



(a) Paging

logical to physical translation

segmentation



(b) Segmentation