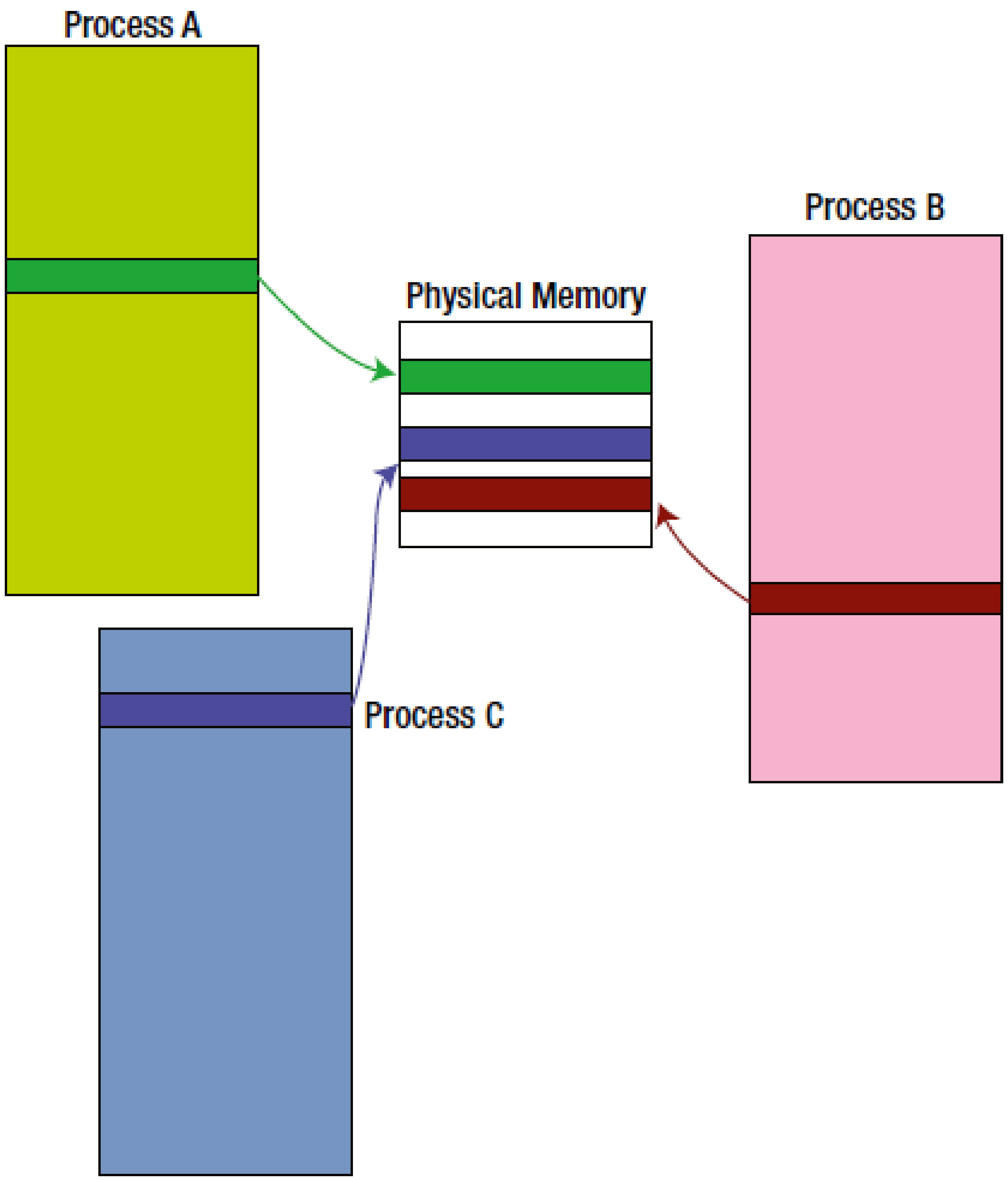
תכנות מתקדם
מצגת 8

זיכרון

זיכרון מדומה

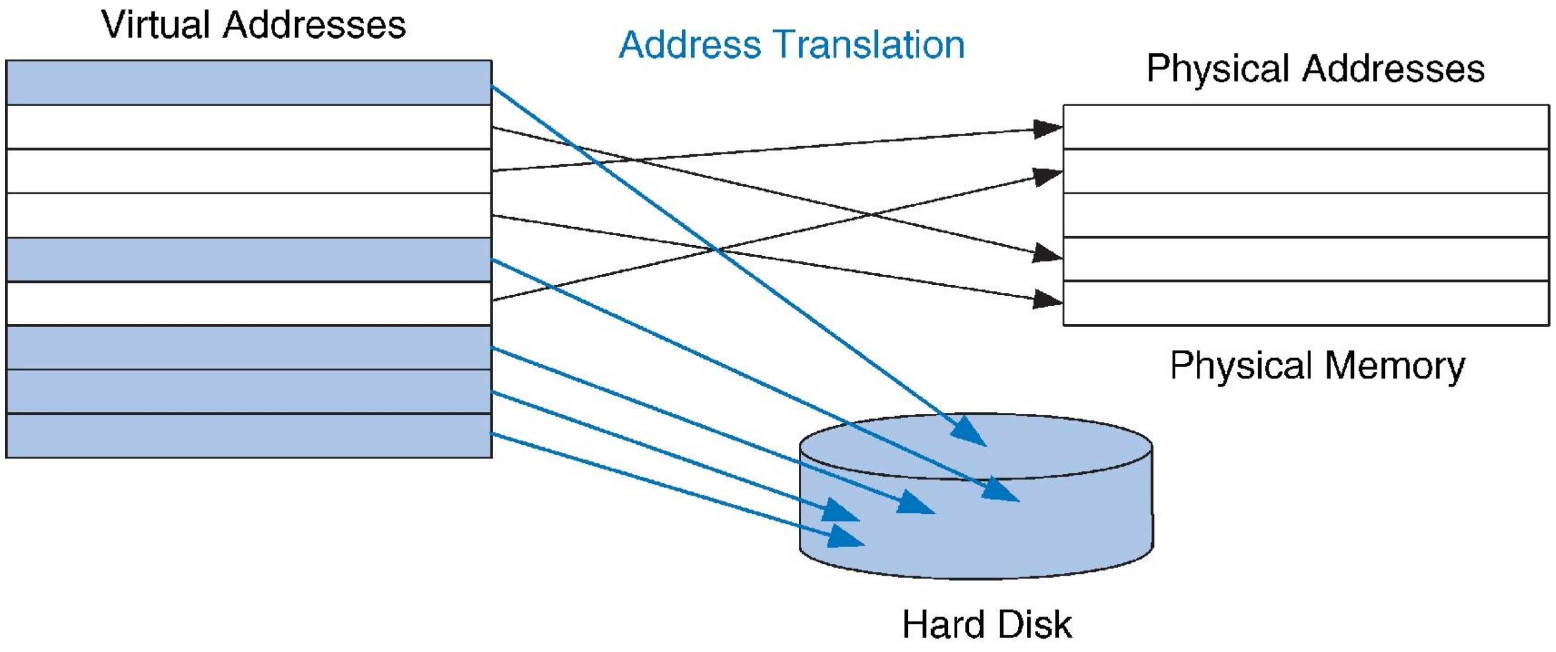
Virtual memory זיכרון מדומה

- Virtual memory enables each process to use of the **entire memory space**
- Subset of **virtual addresses** stored in **physical memory**
- CPU **translates** virtual addresses into physical addresses
- Data not in physical memory fetched from **hard drive**



Virtual memory זיכרון מדומה

- Virtual memory enables each process to use of the **entire memory space**
- Subset of **virtual addresses** stored in **physical memory**
- CPU **translates** virtual addresses into physical addresses
- Data not in physical memory fetched from **hard drive**
- Physical memory acts as **cache** for virtual memory



Locality in time and space

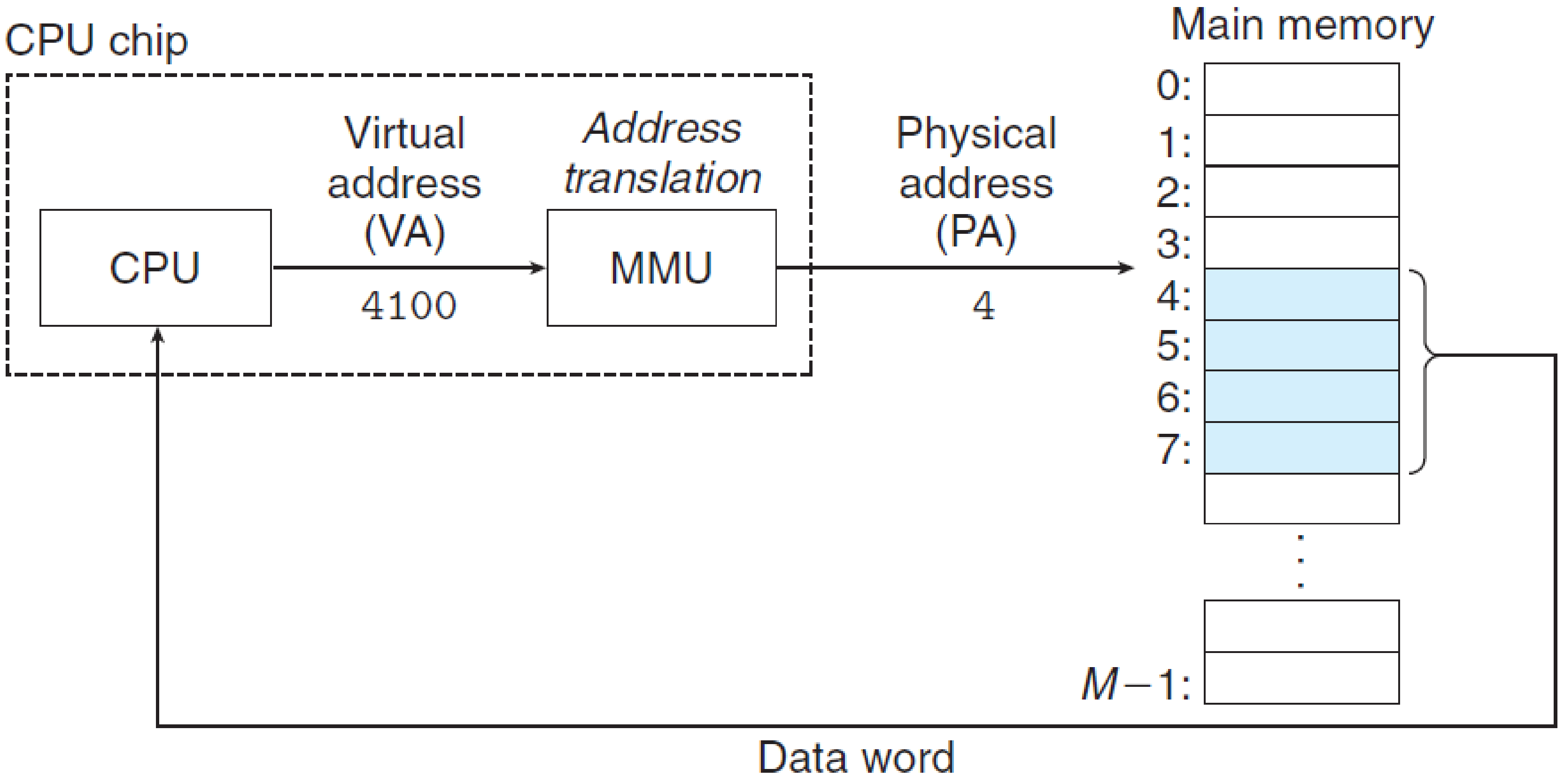
- If a program accesses data in a random order, it would not benefit from a cache
- Caches use time and space locality to store the most commonly data
- **Locality in time**: If data used recently, likely to use it again soon
  - If a variable is used, it is likely to be used again (index in loop)

```
for (i = 0 ; i != 1000 ; ++i) A[i] += x;
```

- **keep recently accessed data in higher levels of memory hierarchy**
- **Locality in space**: If data used recently, likely to use nearby data soon
  - If an element in an array is used, other elements in the same array are also likely to be used (array in loop)
- **bring nearby data into higher levels of memory hierarchy too**

כתובת פיזית וכתובת לוגית

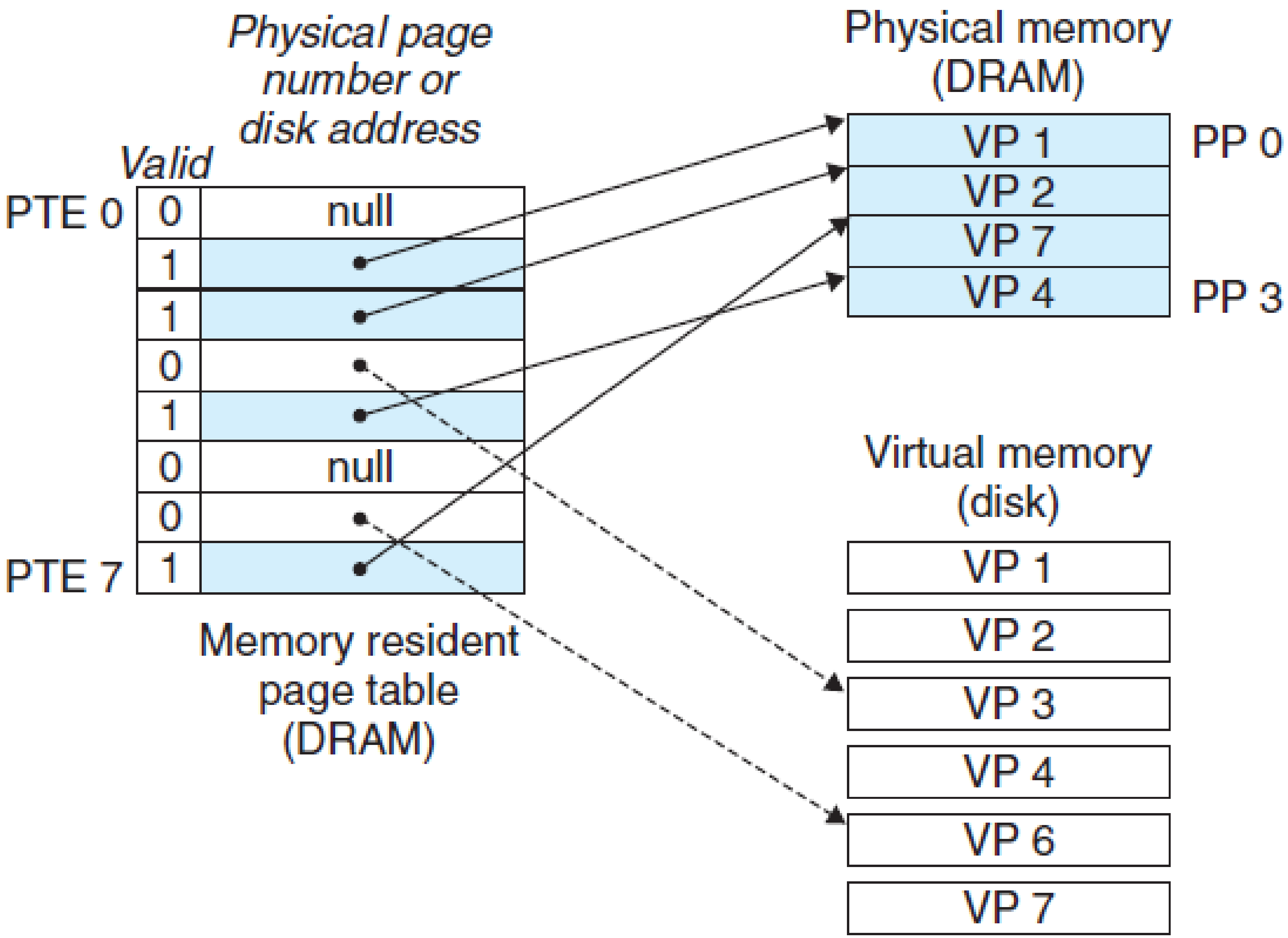
- Each byte in memory has a unique **physical address** (PA)
- The natural way for a CPU to access memory would be to use physical addresses
- With virtual addressing, the CPU uses a **virtual address** (VA), which is **translated** to a physical address
- Hardware on the CPU called the **MMU (**memory management unit) translates virtual addresses on the fly
- The MMU is using a **look-up table** stored in main memory whose contents are **managed** by the operating system



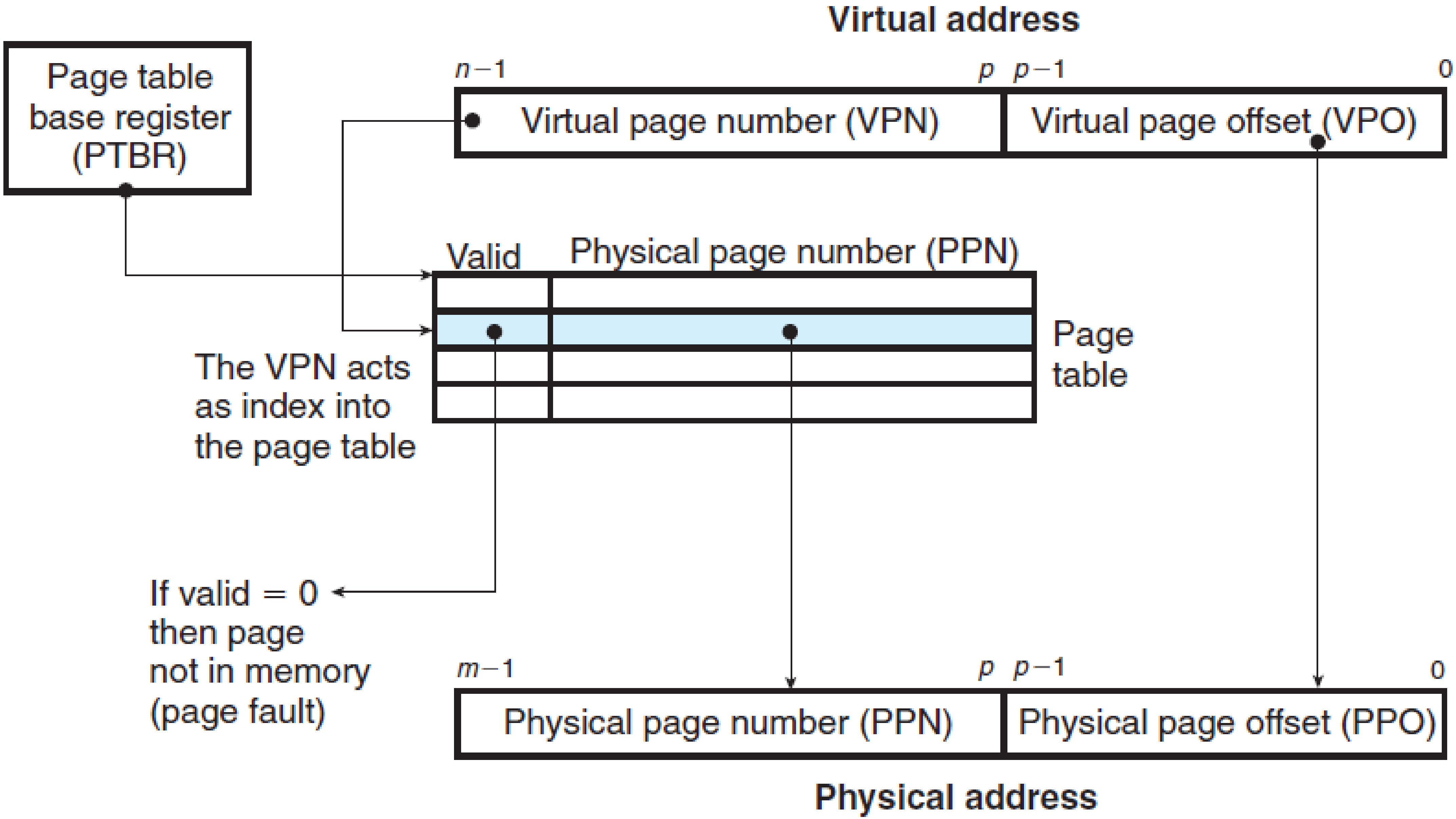
תרגום כתובת לוגית לפיזית באמצעות טבלת דפים

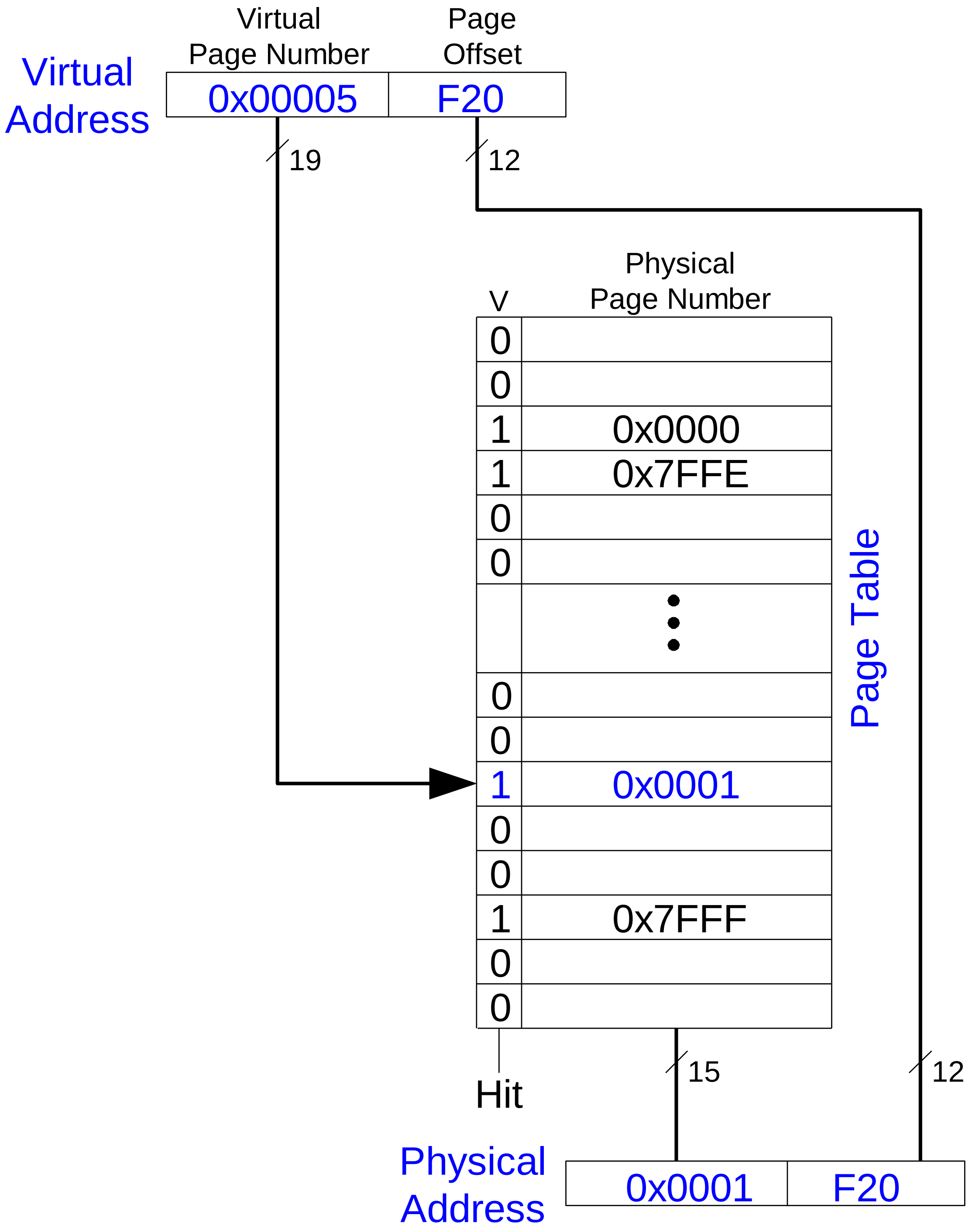
- virtual memory is partitioned into fixed-sized blocks called **virtual pages**
- Similarly, physical memory is partitioned into **physical pages** (page frames) having the same size
- A **page table** maps virtual pages to physical pages
- The MMU reads the page table each time it converts a virtual address to a physical address
- Each page in the virtual address space has a **page table entry** (PTE)
  - If the valid bit is **set**, the address field indicates the start of the corresponding **physical page**
  - If the valid bit is **not set**, then a **null address** indicates that the virtual page is not allocated, it does not occupy any space on disk
  - Otherwise, the address points to the start of the **virtual page on disk**, the page is allocated but is not in physical memory

טבלת דפים


Physical page number or disk address
Valid
PTE 0
0
null
1
1
0
1
0
null
0
PTE 7
1
Memory resident page table (DRAM)
Physical memory (DRAM)
VP 1
VP 2
VP 7
VP 4
PP 0
PP 3
Virtual memory (disk)
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

תרגום כתובת לוגית לפיזית באמצעות טבלת דפים





Example

- Entry for each virtual page
- VPN is index into page table
- Page size: 4 KB = 2^{12} bytes
- Page offset: 12 bits

What is the physical address of virtual address **0x5F20** ?

VPN = 5

Entry 5: physical page **1**

Physical address: **0x1F20**

Example

Given a 32-bit virtual address space, determine the number of bits in the **Virtual Page Number and Virtual Page Offset** for page size **1KB** and **2KB**:

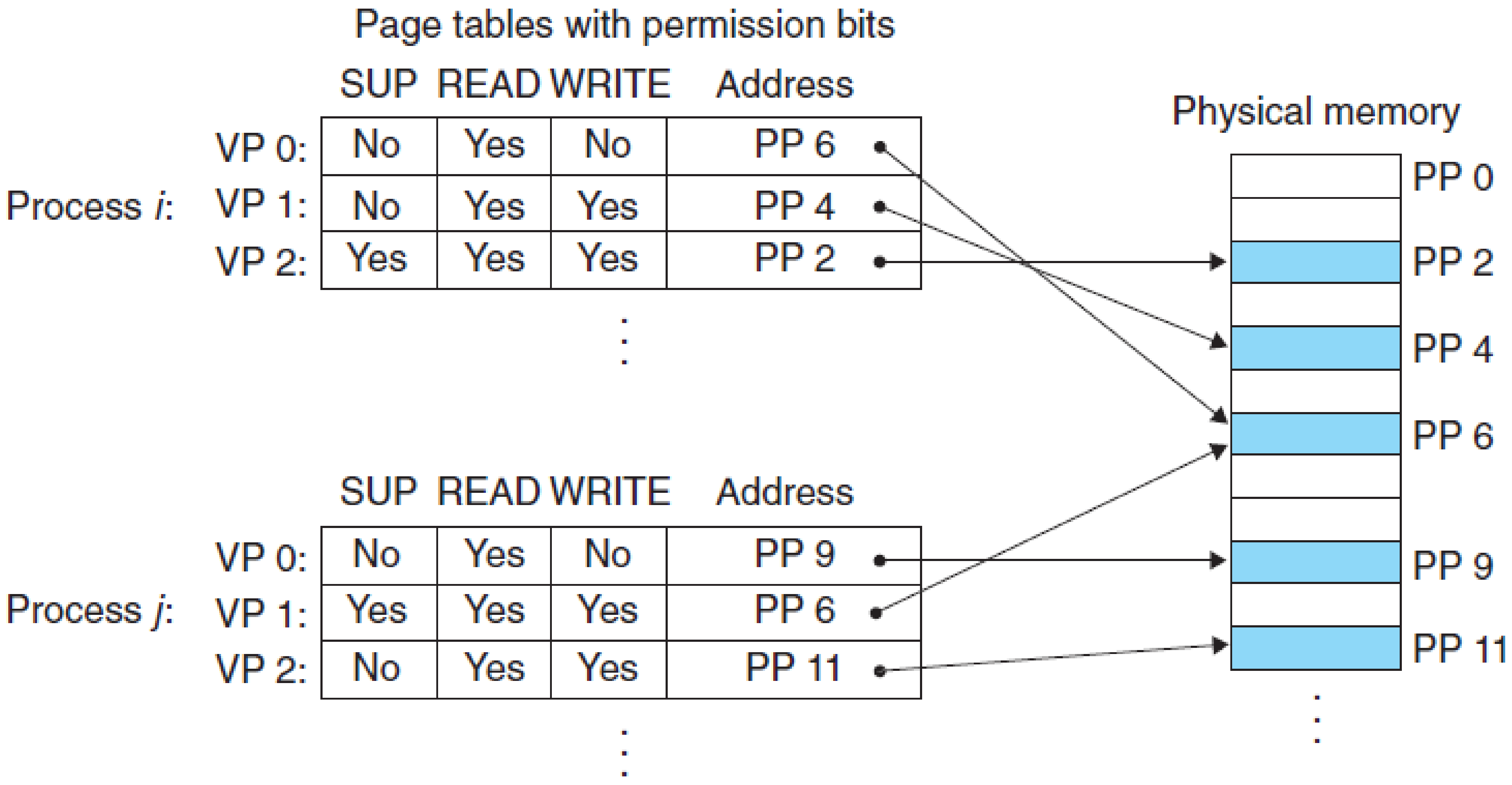
For **1KB** pages the VPO is 0 – 1023 (0x000 – 0x3FF) which requires 10 bits
The number of bits in VPN is 32 – 10 = 22

For **2KB** pages the VPO is 0 – 2047 (0x000 – 0x7FF) which requires 11 bits
The number of bits in VPN is 32 – 11 = 21

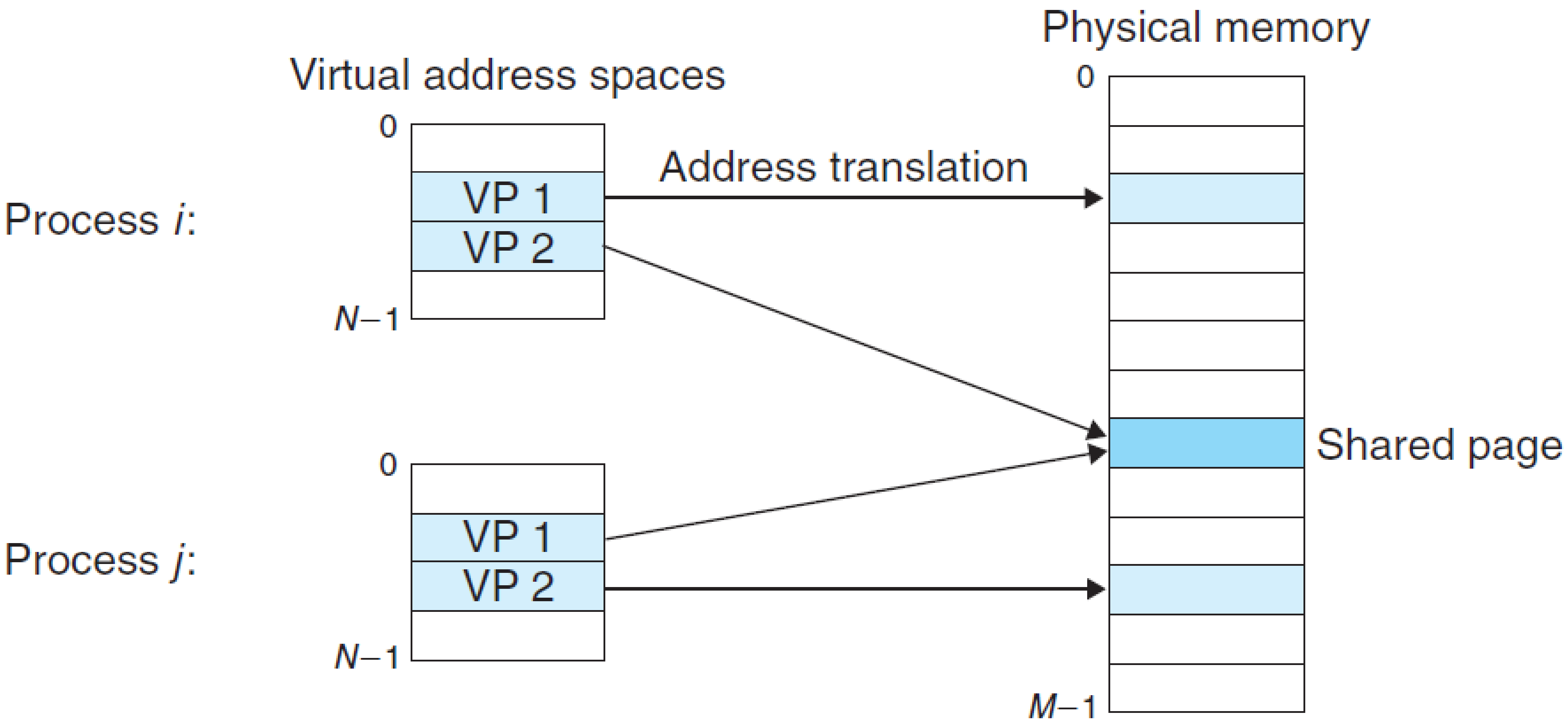
זיכרון מדומה כאמצעי להגנה על זיכרון

- A user process should not be allowed to modify its **read-only** text section
- Nor should it be allowed to read or modify any of the code and data structures in the **kernel**
- It should not be allowed to read or write the private memory of **other** processes
- Each process is provided with a separate **page table**, and thus a separate **virtual address space**
- Providing **separate virtual address** spaces makes it easy to **isolate** the private memories of different processes
- The address translation mechanism can be extended to provide even **finer** access control, by adding some additional permission bits to the PTE
- The **SUP** bit indicates whether processes must be running in kernel (supervisor) mode to access the page

זיכרון מדומה כאמצעי להגנה על זיכרון



זיכרון מדומה כאמצעי לשיתוף זיכרון



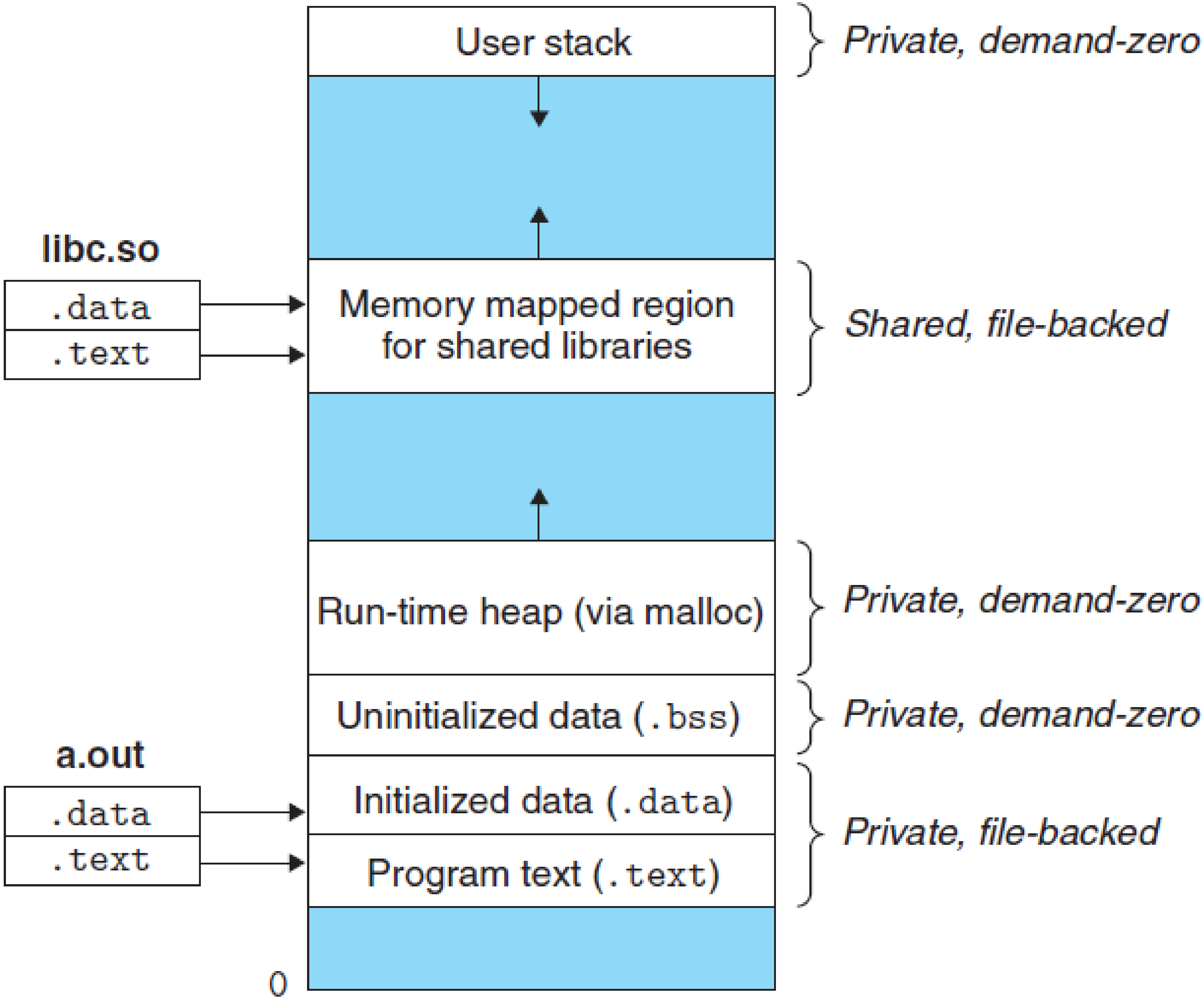
יתרונות נוספים של זיכרון מדומה

- Uses main memory **efficiently** by treating it as a **cache** for an address space stored on **disk**
- **Simplifies memory allocation**. There is no need for the operating system to locate contiguous pages of physical memory
- **Protects** the address space of each process **from corruption** by other processes
- **Simplifies sharing** Multiple processes can share a single copy of code
- **Simplifies linking** Allows each process to use the same basic format for its memory image
- **Fast loading** The loader allocates a contiguous chunk of virtual pages for code and data, marks them as **invalid**, the data is paged in **on demand** the first time each page is referenced

פסיקת דף

- If the CPU reads a word of virtual memory which is **cached** in DRAM, the MMU uses the physical memory address in the PTE to construct the physical address of the word
- A DRAM cache **miss** is known as a **page fault**, the MMU triggers a page fault exception
- The **exception handler** in the kernel, selects a physical page, copies to it the page from disk and updates the page table
- When the handler returns, it **restarts** the faulting instruction
- The strategy of waiting until the last moment to swap in a page, when a miss occurs, is known as **demand paging**

How the loader (execve) maps the areas

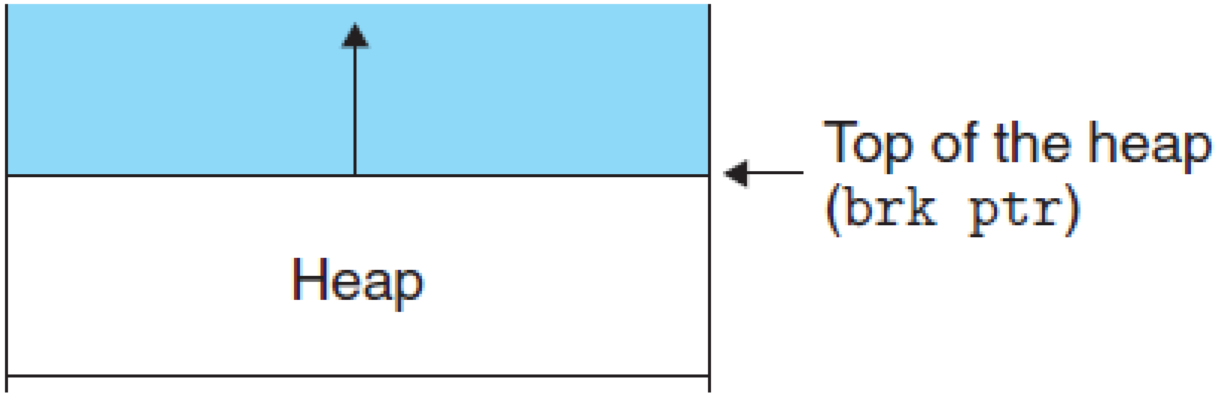


הקצאת זיכרון דינמית

הקצאת זיכרון דינמית

- A dynamic memory allocator maintains an area of a process's virtual memory known as the **heap**
- An allocator maintains the heap as a collection of various-sized blocks
- Each block is a contiguous chunk of memory that is either allocated or free
- A variable **brk** (pronounced "break") points to the top of the heap

- **Explicit allocators** require the application to explicitly free allocated blocks
- **Implicit allocators** (garbage collectors), require the allocator to detect when an allocated block is no longer being used



malloc() and free()

- The **malloc** function returns a pointer to a block of memory of at least size bytes

```
int* array = (int*) malloc(10 * sizeof(int));
```

- **malloc** returns a block that is **aligned to an 8-byte** (double word) boundary
- Programs free allocated heap blocks by calling the **free** function
- **malloc** allocates more heap memory by using the **sbrk** function:

```
void *sbrk(intptr_t incr);
```

- The **sbrk** function grows or shrinks the heap by adding **incr** to the kernel's **brk** pointer

new and delete

- Using **malloc( )** and **free(**) in C++ will not initialize and cleanup the object:

```
p = (cls*) malloc(sizeof(cls));
```

- Use **new** and **delete**:
- **new** is an **operator** that perform the combined act of dynamic storage allocation and initialization
- **delete** is an **operator** that perform the combined act of cleanup and releasing storage
- Since they are operators, the compiler can guarantee that constructors and destructors will be called for all objects

```
MyClass *fp = new MyClass(1,2);
```

- malloc(sizeof(MyClass)) is called, then the constructor for MyClass is called using (1,2) as the argument list

new & delete for arrays

```
MyType* fp = new MyType[100];
```

- The constructor is called for each object in the array
- There must be a default constructor, because a constructor with no arguments must be called for every object

```
delete [] fp;
```

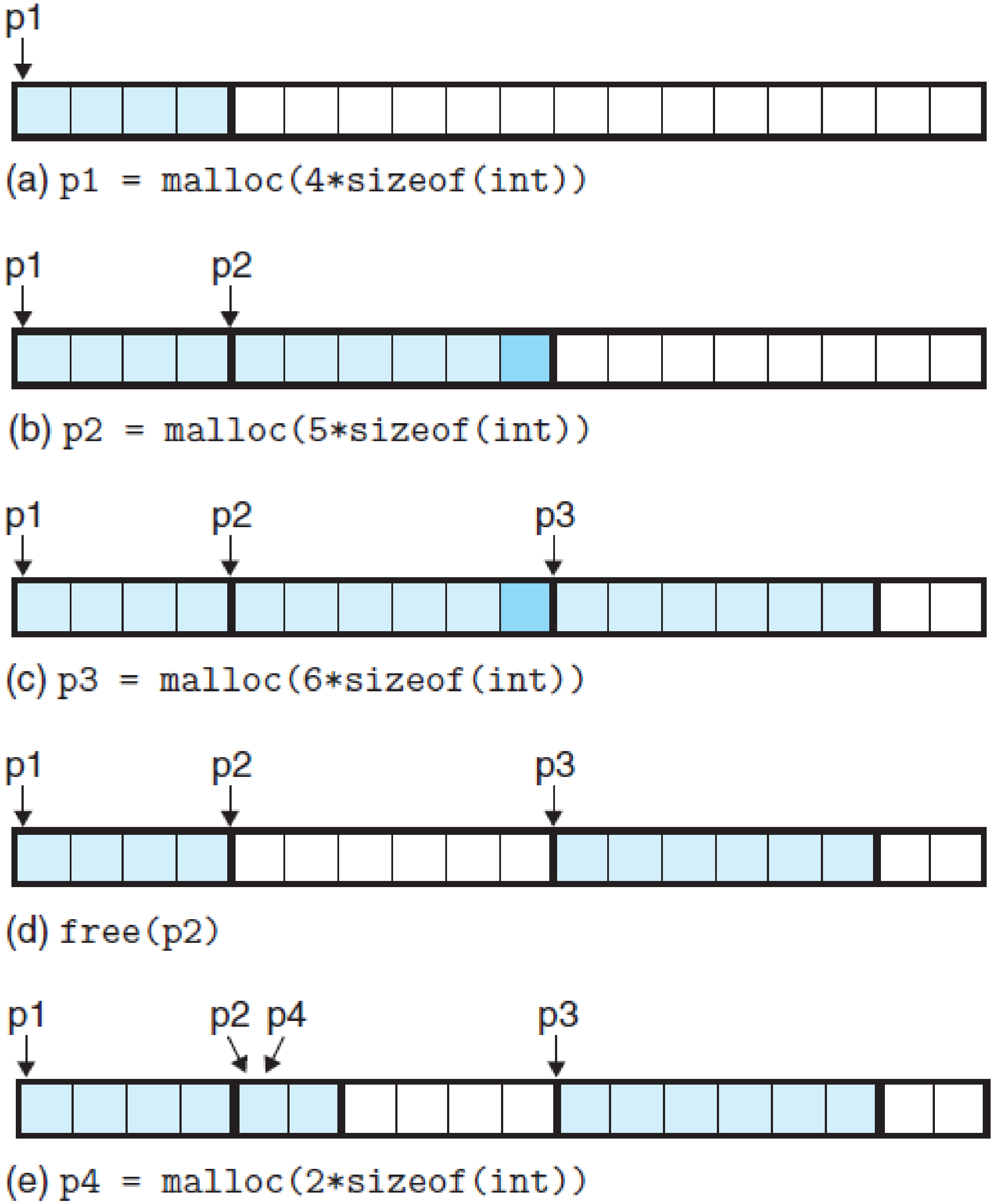
- calls the destructor for all objects in the array

```
delete fp;
```

- Will release storage, but the destructor will be called for just the first object

Dynamic Memory Allocation Example

- The heap consists of (4 bytes) words (each box)
- Allocations are aligned on (8 bytes) double words
- **Notice** that after the call to free, the pointer **p2 still points** to the freed block



Allocator Goals

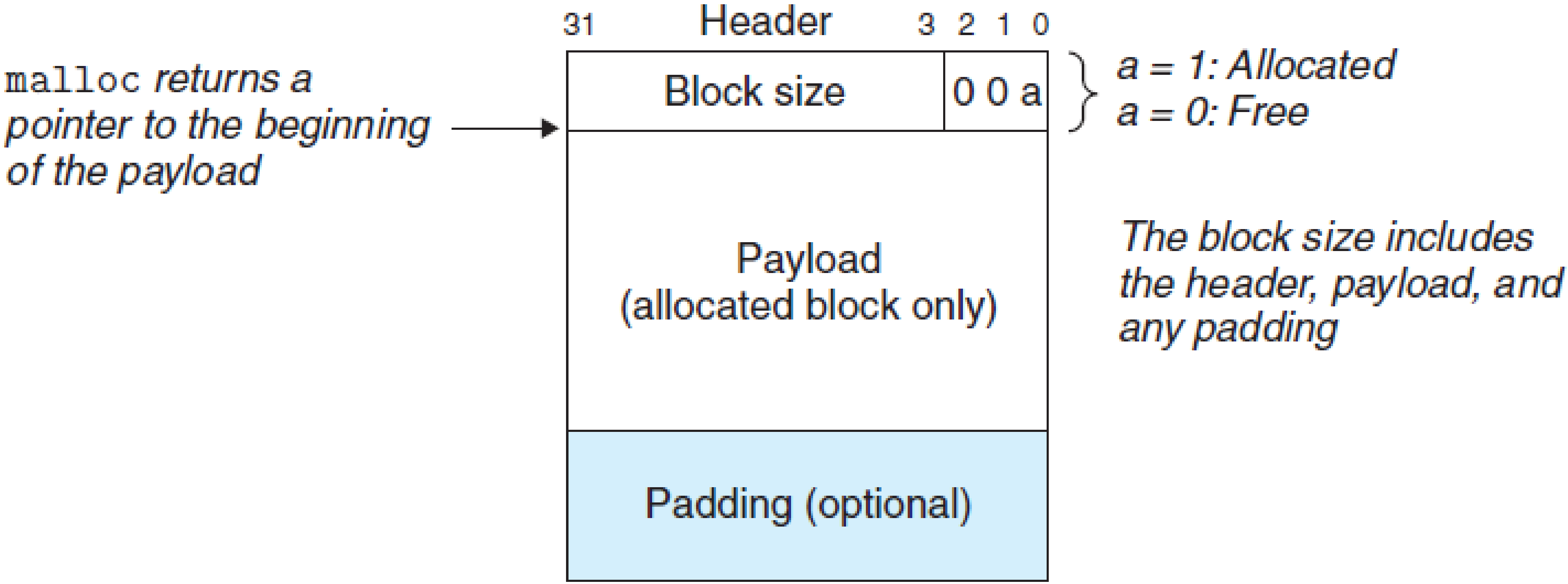
- **Maximize throughput**
  - number of requests completed per unit time
- **Minimize fragmentation**
  - **Internal fragmentation** occurs when an allocated block is larger than the payload
    - alignment constraints
    - minimum size on allocated blocks
  - **External fragmentation** occurs when there is enough aggregate free memory to satisfy an allocate request, but no single free block is large enough to handle the request

Allocator Implementation

- **Free block organization** How do we keep track of free blocks?
- **Placement** How do we choose an appropriate free block in which to place a newly allocated block?
- **Splitting** After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
- **Coalescing** What do we do with a block that has just been freed?

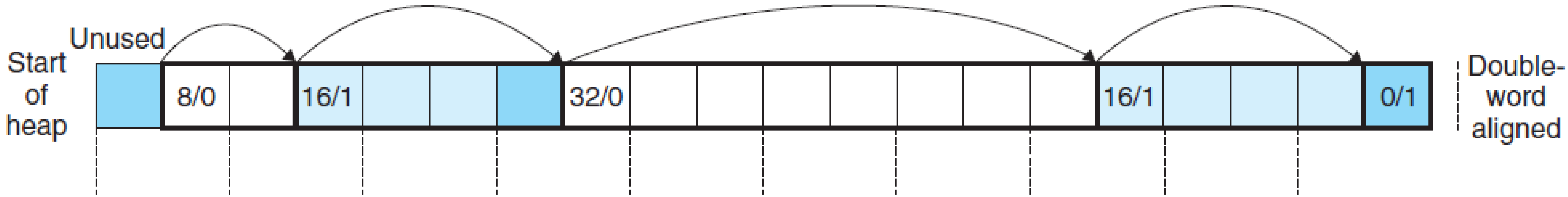
Implicit Free Lists

- Header of block contains size and whether allocated or free
- Since blocks are double-word aligned, we need only 29 bits for the block size, freeing the remaining 3 bits for other information



Implicit Free Lists

- The free blocks are linked **implicitly** by the size fields in the headers
- We are using the allocated bit to indicate whether the block is allocated
- The allocator can traverse the entire set of **free** blocks by traversing **all** of the blocks in the heap



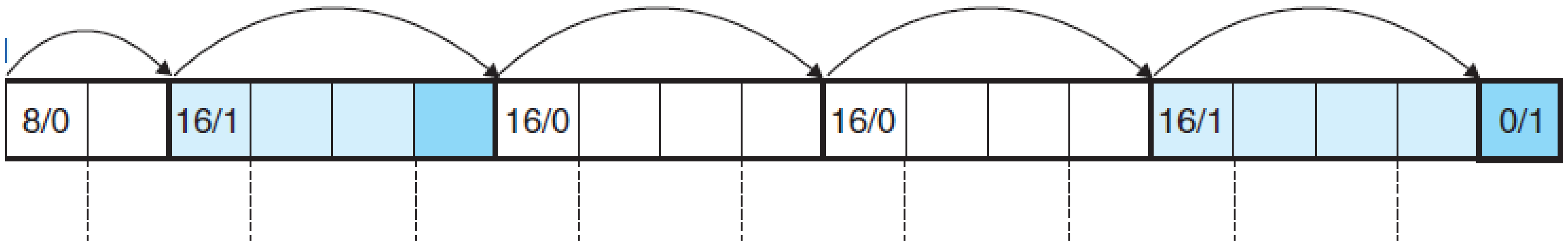
Allocated blocks are shaded, headers are labeled (size/allocated bit)

Placing Allocated Blocks

- When an application requests a block of k bytes, the allocator searches the free list for a free block that fits
- **First fit** searches the free list from the beginning and chooses the first free block that fits
  - advantage - leaves large free blocks at the end of the list
  - disadvantage - creates small free blocks toward the beginning
- **Next fit** starts each search where the previous search left off
  - Next fit can run significantly faster than first fit
- **Best fit** chooses the free block with the smallest size that fits
  - has better memory utilization but requires an exhaustive search

Splitting and Coalescing Free Blocks

- Once the allocator has located a free block, it must decide weather to use the entire free block or to **split** the free block into two parts
- When the allocator frees an allocated block, there might be other free blocks that are adjacent to the newly freed block
  - Such adjacent free blocks can cause **false fragmentation**
  - To avoid false fragmentation, the allocator must coalesce (merge) adjacent free blocks

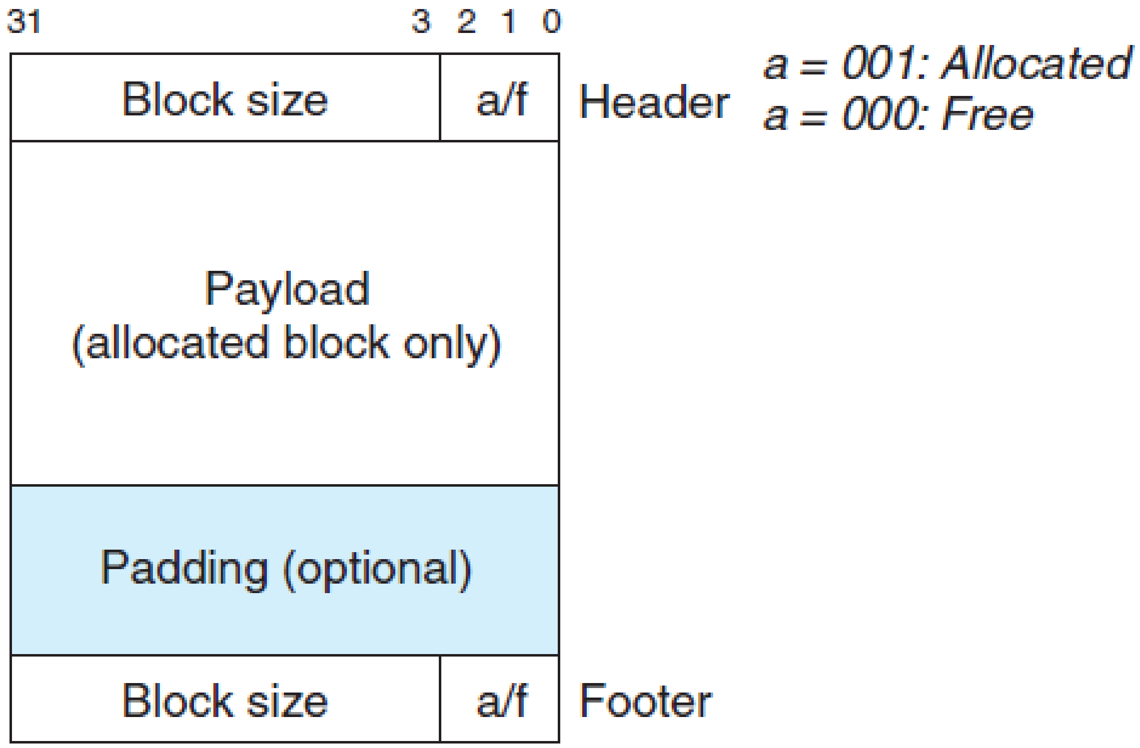


Coalescing

- Coalescing the next free block is straightforward and efficient
  - The header of the current block points to the header of the next block, which can be checked to determine if the next block is free
  - If so, its size is added to the size of the current header and the blocks are coalesced in constant time
- But for coalescing the previous block, the only option would be to search the entire list
  - With implicit free list, this means that each call to free would require time linear in the size of the heap

Coalescing with Footer

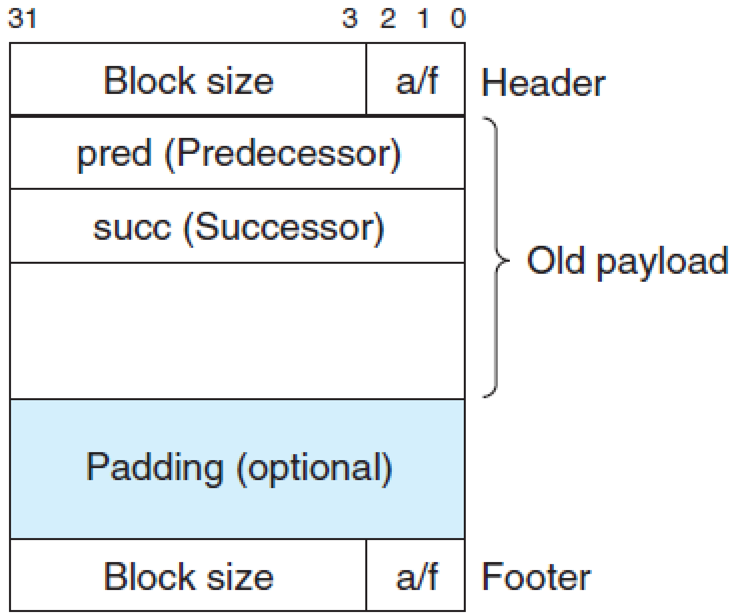
- Footers, allow for constant-time coalescing of the previous block
- The idea, is to add a footer at the end of each block, where the footer is a replica of the header
- The footer of the previous block is one word away from the start of the current block
- So the allocator can determine the starting location and status of the previous block by inspecting its footer



- If we were to store the allocated/free bit of the previous block in one of the excess low order bits of the current block, then allocated blocks would not need footers

Explicit Free Lists

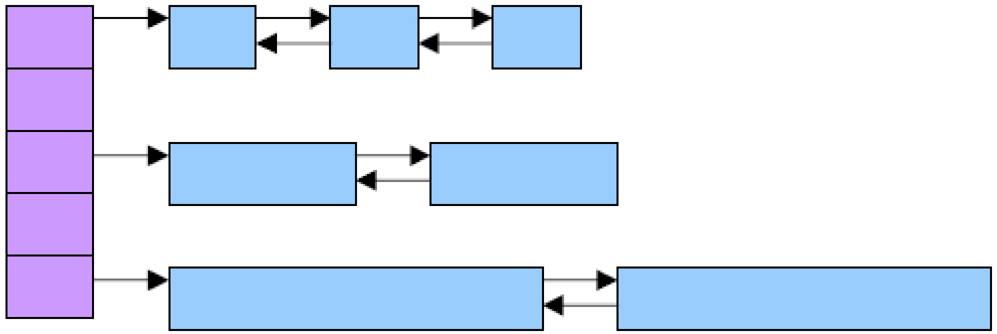
- With implicit free list, block allocation time is **linear** in the total number of heap blocks
- A better approach is to organize the free blocks into an explicit doubly linked list
- The pointers that implement the list can be stored within the bodies of the **free blocks**
- Reduces the first fit allocation time to linear in the number of free blocks



- Can be maintained in last-in first-out (LIFO) order by inserting newly freed blocks at the beginning of the list
  - freeing a block can be performed in constant time
  - If boundary tags are used, then coalescing can also be performed in constant time

Multiple Free Lists

- A single linked list of free blocks requires time linear in the number of free blocks to allocate a block
- Can maintain **multiple free lists**, where each list holds blocks that are roughly the same size
- the free list for each size class contains same-sized blocks, free blocks are never split
- To free a block, the allocator inserts the block at the front of the list, the list need only be singly linked
- Allocating and freeing blocks are both fast constant-time operations
- No splitting, and no coalescing means there is little per-block overhead
- Since there is no coalescing, allocated blocks require no headers
- However may cause internal and external fragmentation



Problems with Manual Management

- **Memory leaks** where resources are never freed

```
void leak1() {Object *x = new Object; return; }
void leak2()
    {Object *x = new Object; . . . x = new Object; . . .}
```

- **Double frees** where a resource is freed twice
  - May corrupt the heap or destroy a different object

```
p = malloc(10); ...  free(p);  q = malloc(10); free(p);
```

- **Use-after-free** (dangling pointer) where a deallocated resource is still used

```
for (p = head; p != NULL; p = p->next) { free(p); }
// correct:
for (p = head; p != NULL; p = q)
    {q = p->next; free(p); }
```

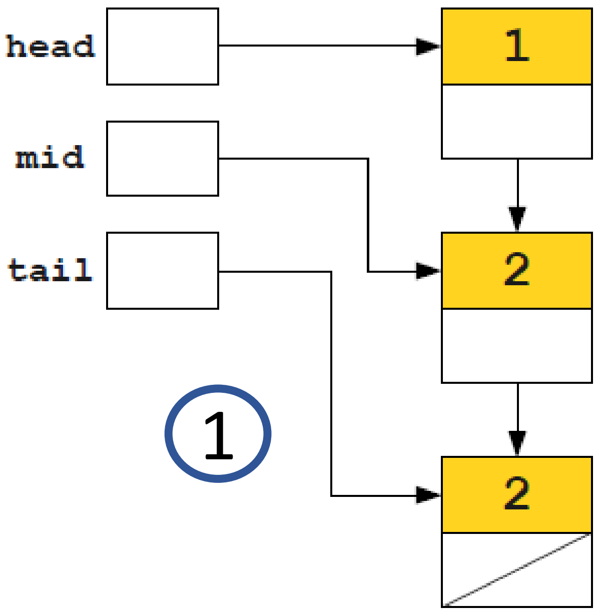
valgrind

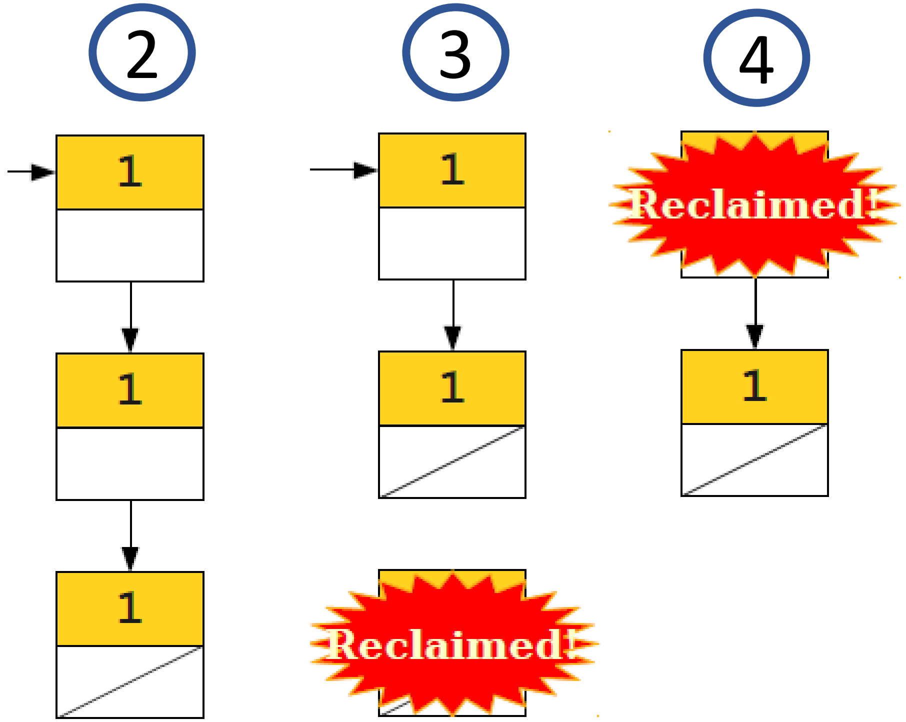
Automatic Management - Reference Counting

- Store with each object a reference count tracking how many references (pointers) exist to the object.
  - Creating a reference to an object increments its reference count
  - Removing a reference to an object decrements its reference count
- If its reference count reached zero, it is unreachable
  - Reclaim the memory for that object
  - Remove all outgoing references from that object
    - This might trigger other reclaims

Reference Counting Example

```
class LinkedList {
      LinkedList next;
}
void f() {
LinkedList *head = new LinkedList;
LinkedList *mid = new LinkedList;
LinkedList *tail = new LinkedList;
head->next = mid;
mid->next = tail; 1
mid = tail = null;  2
head->next->next = null; 3
head = null; 4  5 all reclaimed
}
```

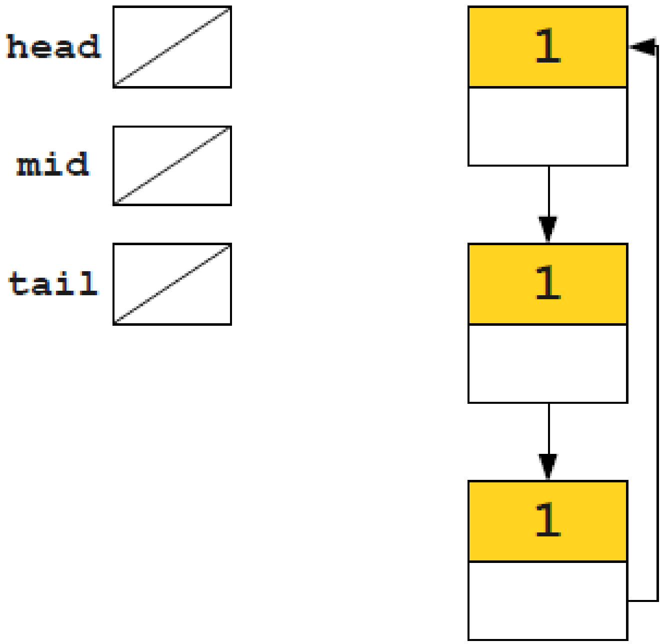




Reference Cycles

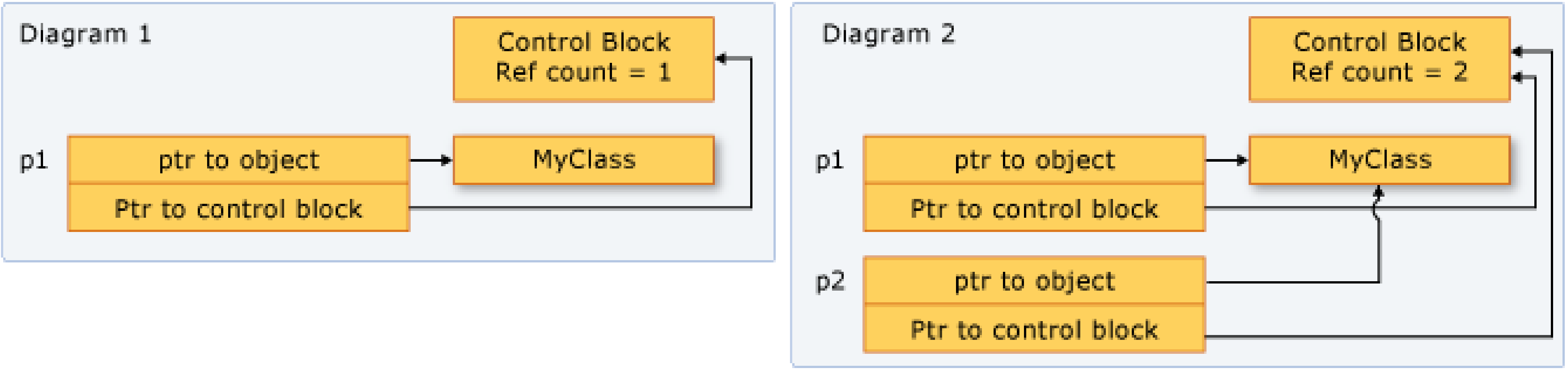
- A reference cycle is a set of objects that cyclically refer to one another
- Because all the objects are referenced, they may have nonzero reference counts but be unreachable

```
LinkedList *head = new LinkedList;
LinkedList *mid = new LinkedList;
LinkedList *tail = new LinkedList;
head->next = mid;
mid->next = tail;
tail->next = head;
head = null;
mid = null;
tail = null;
```



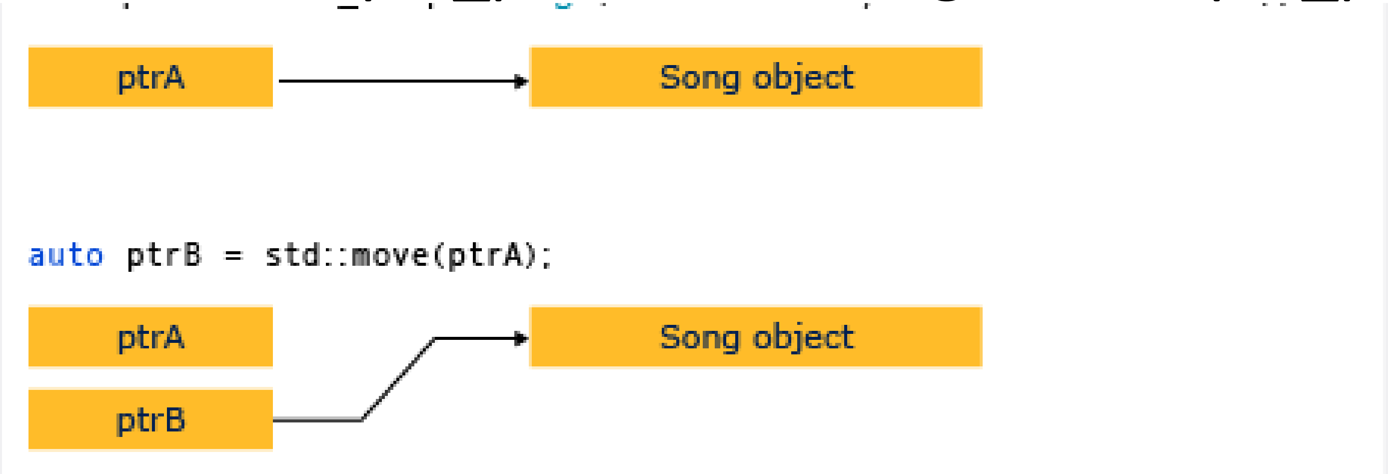
shared_ptr

- **shared_ptr** is a class with * and -> overloaded
- The class contains the actual raw pointer and a pointer to a reference count which keeps track of how many **shared_ptrs** point to it
- **shared_ptr** constructor increases the reference count
- **shared_ptr** destructor decrements the count and if it reached 0 deletes
- **shared_ptr** can be copied, passed by value, and assigned



unique_ptr

- **unique_ptr** is a small, fast smart pointer for managing resources with exclusive-ownership semantics
  - There is no reference counting
  - When you need a smart pointer for a plain C++ object, prefer unique_ptr
- Cannot be copied to another unique_ptr or passed by value to a function
  - If you could copy a **unique_ptr**, you'd end up with two **unique_ptrs** to the same resource
- A unique_ptr can be **moved**, the ownership of the memory resource is transferred to another unique_ptr and the original unique_ptr no longer owns it:



garbage collection

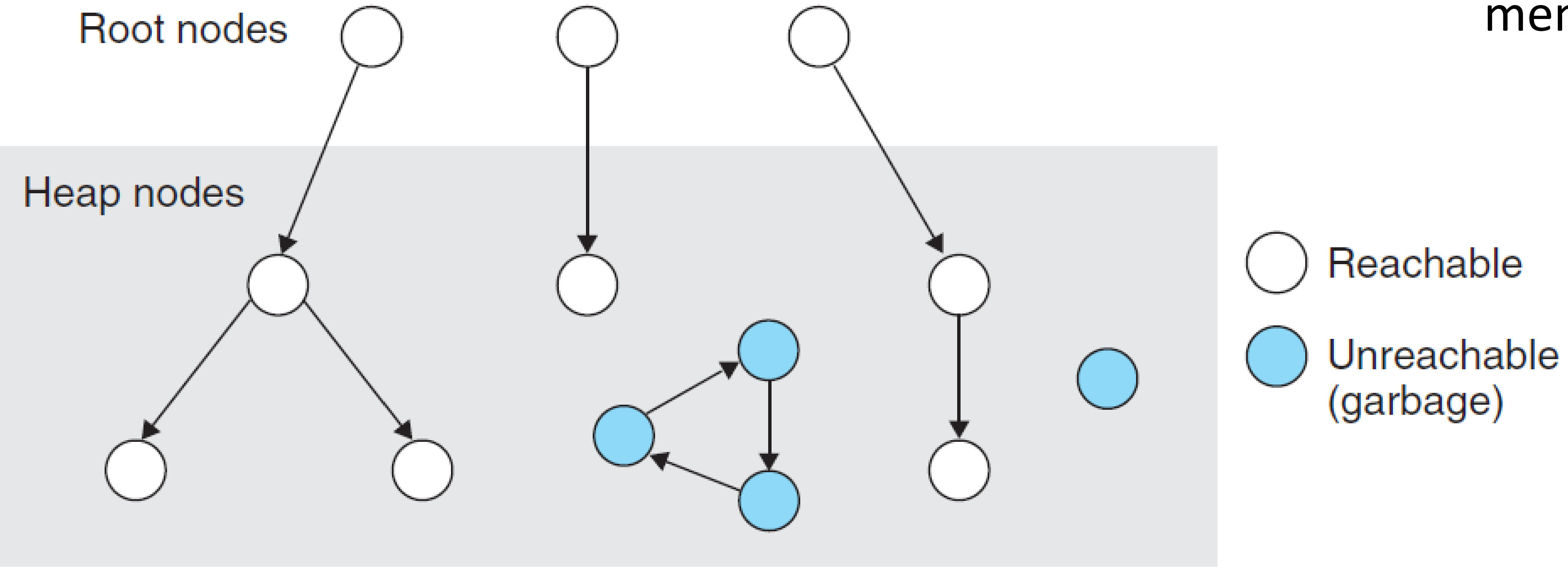
- An object is **reachable** if it can still be referenced by the program
- Some **reachable** objects are garbage – they may never be used again:

```
Object *x = new Object;
Object *y = new Object;
/* What is garbage ? */
x.doSomething();
/* What is garbage ? */}
```

- However, automatic memory management can only detect and reclaim **unreachable** objects
- Reclaiming **unreachable** objects automatically is called **garbage collection**

Garbage Collector

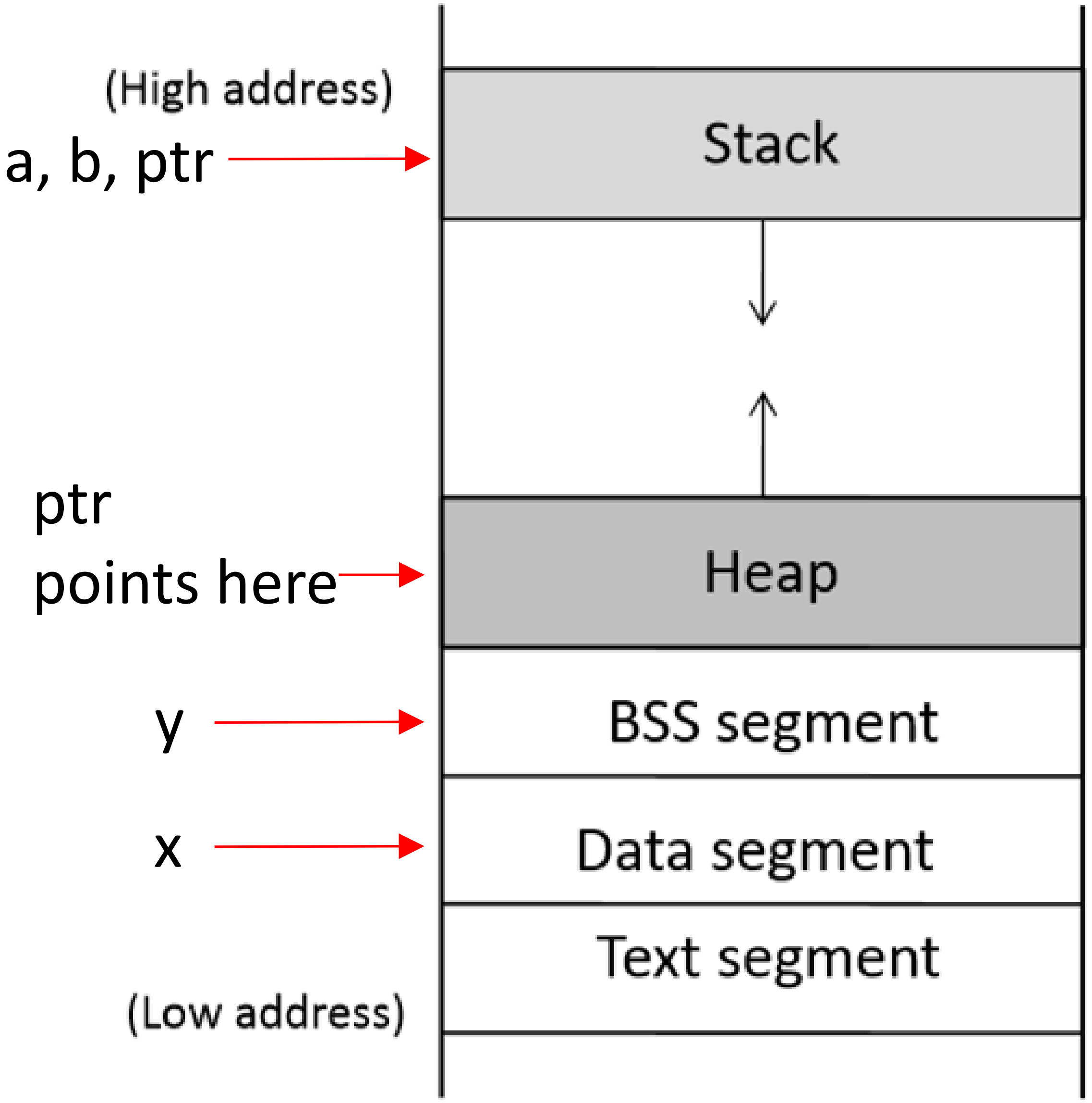
- A garbage collector views memory as a directed reachability graph
- The graph is partitioned into a **root set** and a **heap set**
- **Root set** correspond to locations not in the heap - variables on the **stack**, or **global** variables in the read-write data area
- **Heap set** correspond to **allocated blocks** in the heap



memory map

Program Memory

```
int x = 100;          //  Data segment
int main ( )
{
   int a = 2;         // Stack
   float b = 2.5;     // Stack
   static int y;      // BSS segment
   // allocate memory on Heap
   int *ptr = (int *) malloc(2*sizeof(int));
   // values 5 and 6 stored on heap
   ptr[0] = 5;
   ptr[1] = 6;
   // deallocate memory on heap
   free (ptr) ;
}
```



Garbage Collector

- block **q** is **reachable** if there exists a directed **path** from any **root** to **q**
- The path may include a directed edge **p → q** where some location in **block p** points to some location in **block q**
- **Unreachable** blocks correspond to **garbage,** they can never be used by the application
- A **Mark and Sweep** garbage collector consists of two phases:
  - A **mark phase**, which marks all reachable and allocated descendants of the root nodes
  - A **sweep phase**, which frees each unmarked allocated block
- Typically, one of the spare low-order bits in the **block header** is used to indicate whether a block is marked or not

Mark and Sweep

`typedef void *`**`ptr`**`;`

`ptr isPtr(ptr p):` `If p points to an` **`allocated`** `block, returns a pointer to that block`

```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}
```

```
void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
         unmarkBlock(b);
      else if (blockAllocated(b))
         free(b);
      b = nextBlock(b);
   }
   return;
}
```

Mark and Sweep

- The **mark function** is called for each root
- The **mark function** calls itself recursively on each word in block
- The **sweep function** is called once, it iterates over each block in the heap