Operating Systems

Lecture 10

17. Free-Space Management

What is a heap?

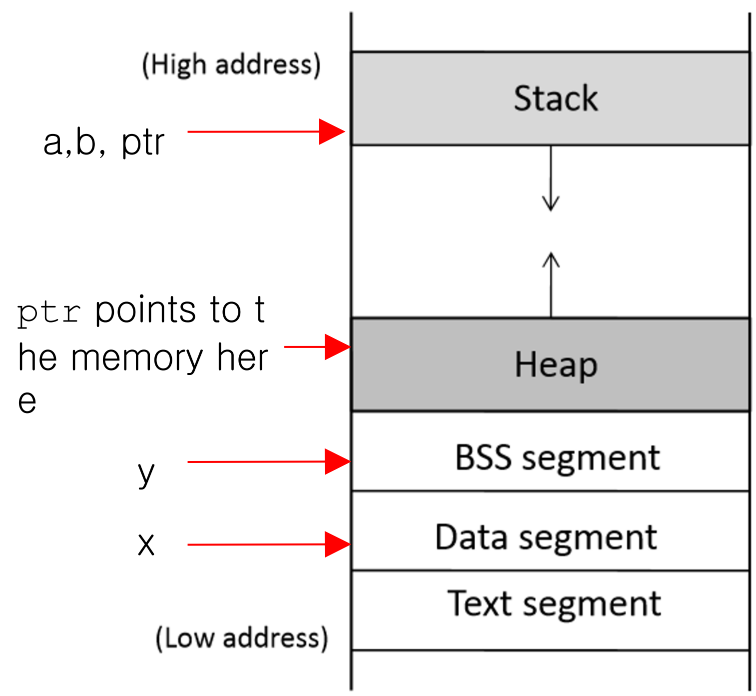
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // 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);

  return 1;
}
```



What is a heap?

- Heap is a collection of variable-size memory chunks allocated by the p rogram
  - e.g., `malloc()`, `free()` in C,

    creating a new object in Java

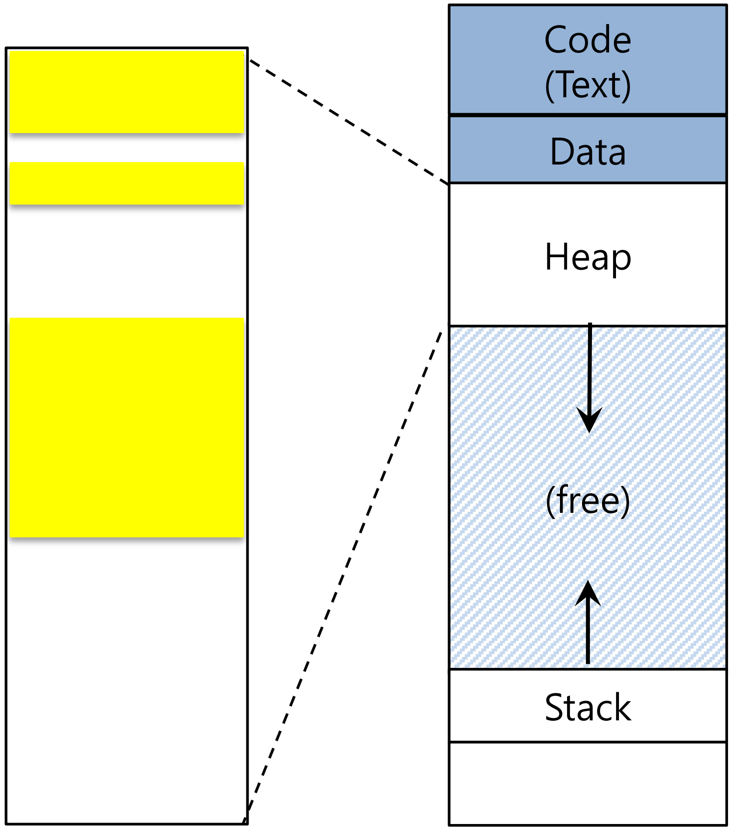
    creating a new object in Javascript

- Heap management system controls the allocation, de-allocation, and r eclamation of memory chunks.
  - To do this some meta data is necessary for book-keeping

Managing heap

```
malloc()
free()
```
in **`libc`**

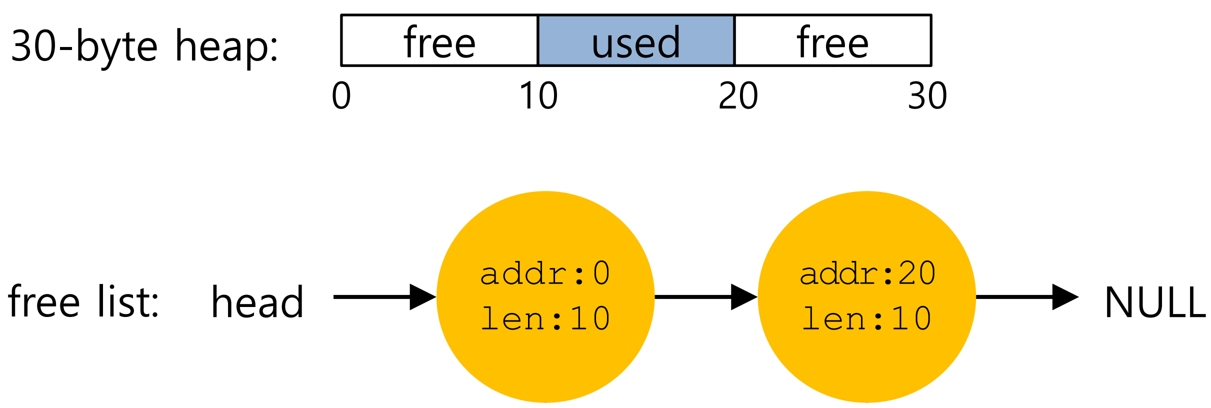
To manage the memory in a heap



Managing heap

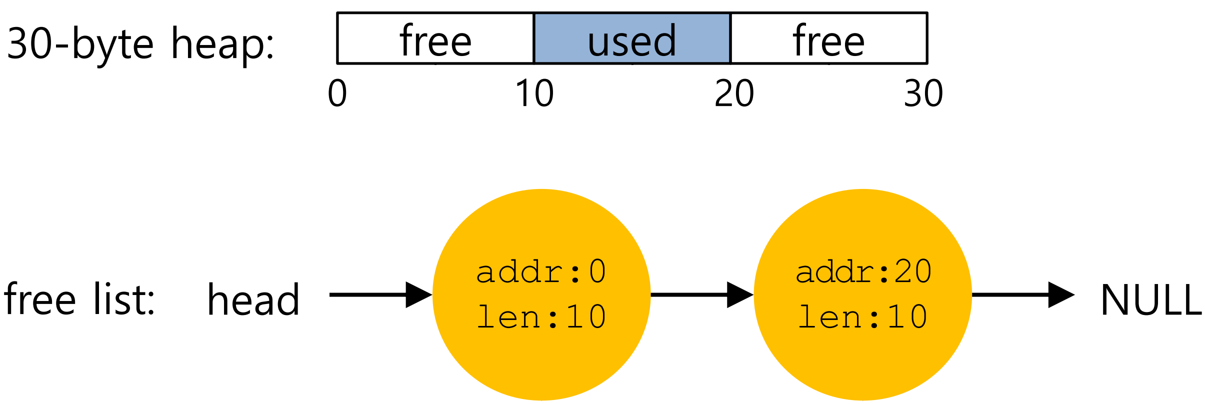
- Upon memory request by `malloc()`, finds a free chunk of memory that can satisfy the request from a free list.

```
malloc(5);
```



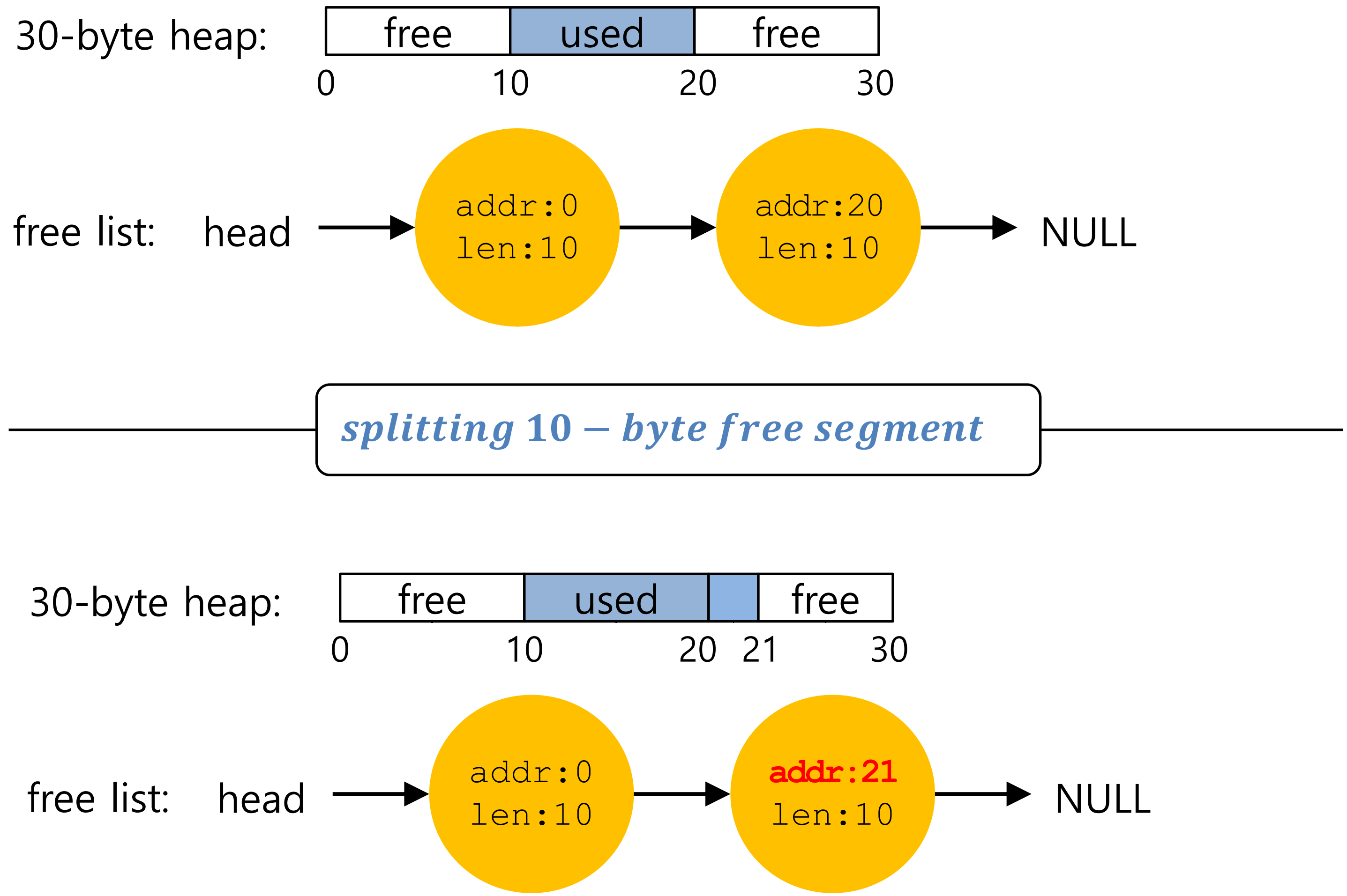
Splitting

- Finding a free chunk of memory that can satisfy the request and splitting it into two.
  - When request for memory allocation is **smaller** than the size of free chunks.



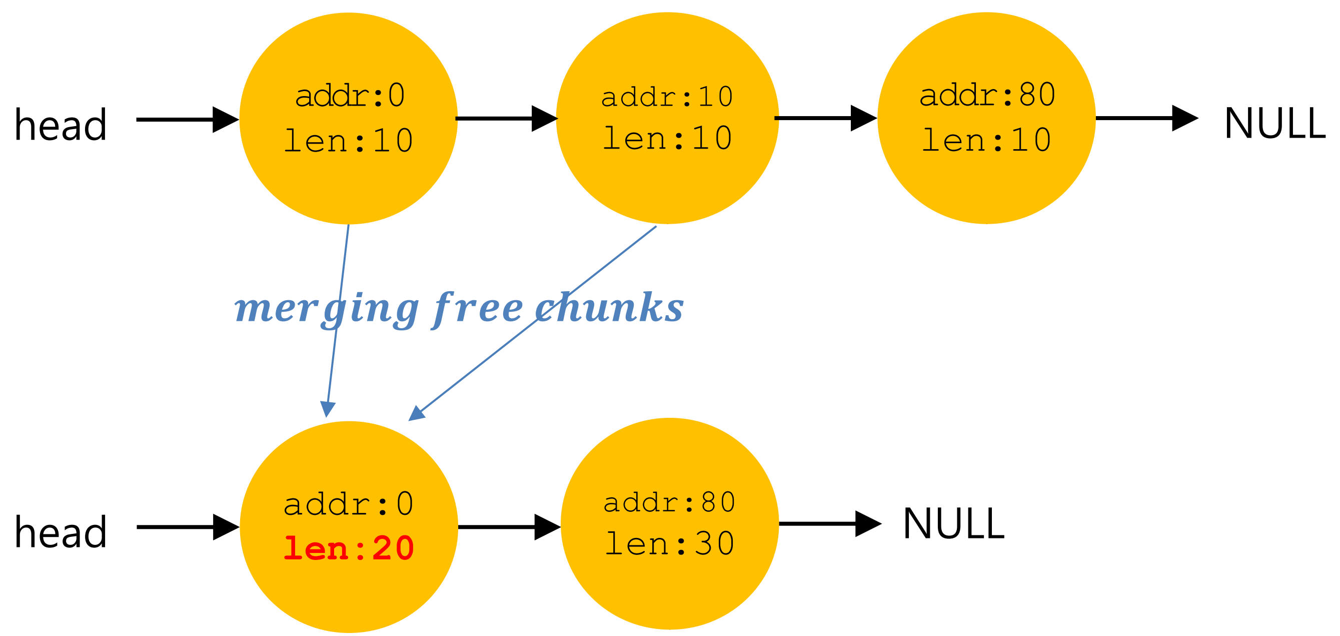
Splitting(Cont.)

- Two 10-bytes free segment with **1-byte request**



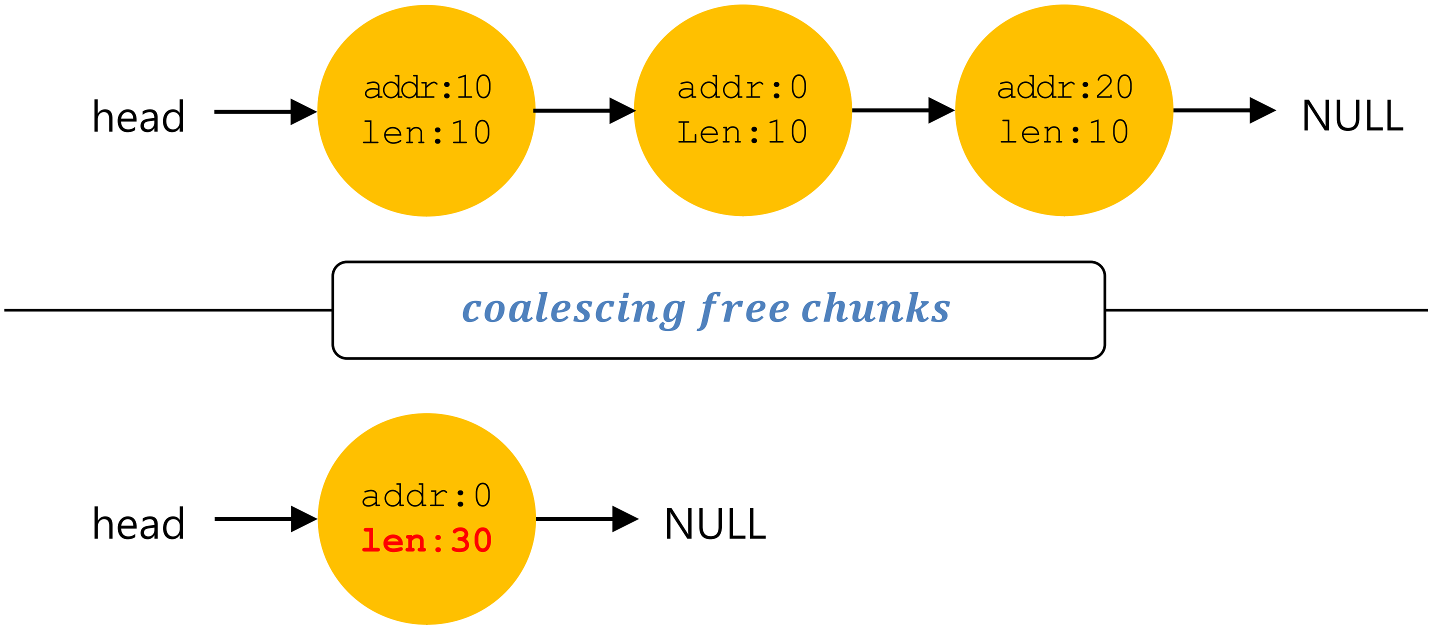
Managing heap

- `free()` will deallocate the used chunk and insert it to a free list to be used later.
- A free chunk may be merged with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



Coalescing

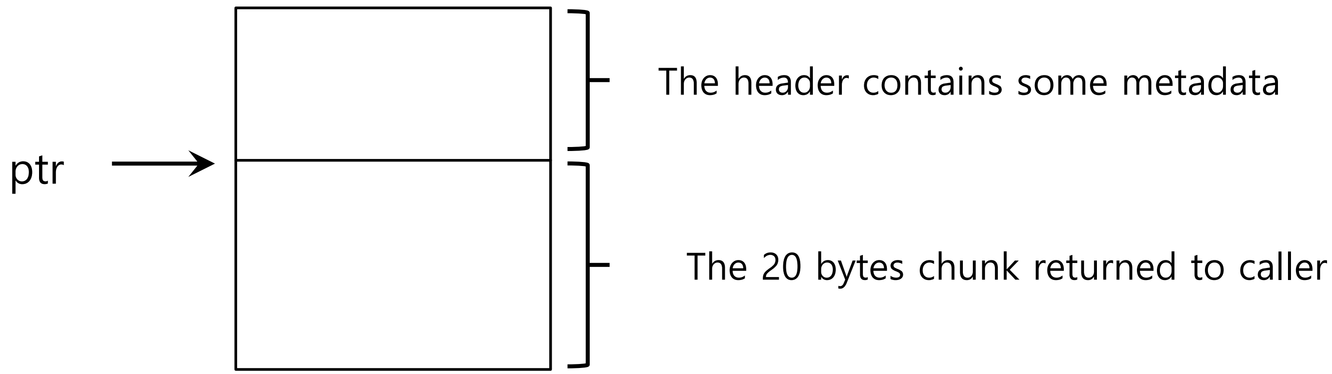
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



Managing heap

- Every chunk has a metadata in its header for the heap management

```
ptr = malloc(20);
```

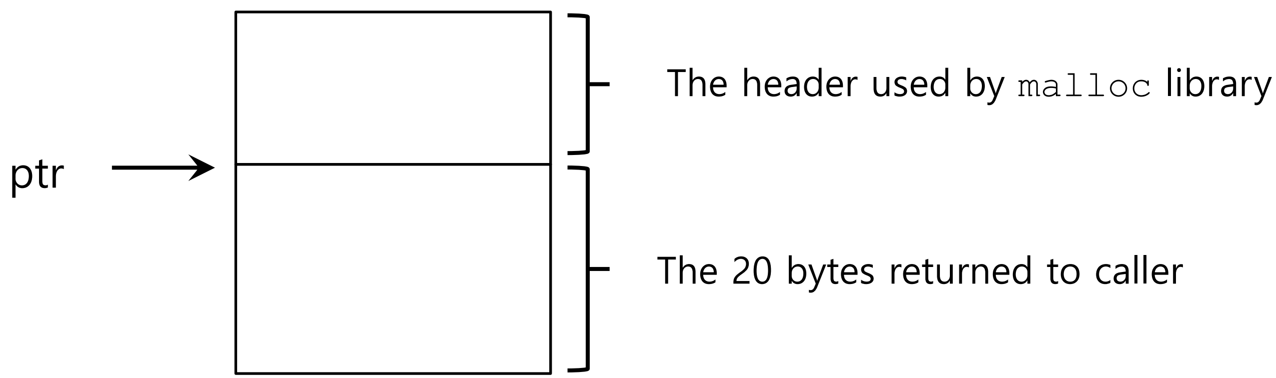


An allocated chunk plus Header

Tracking The Size of Allocated Regions

- The interface to `free(void *ptr)` does **not take** a **size parameter**.
  - How does the library **know the size** of memory region that will be back **into free list**?

```
ptr = malloc(20);
```

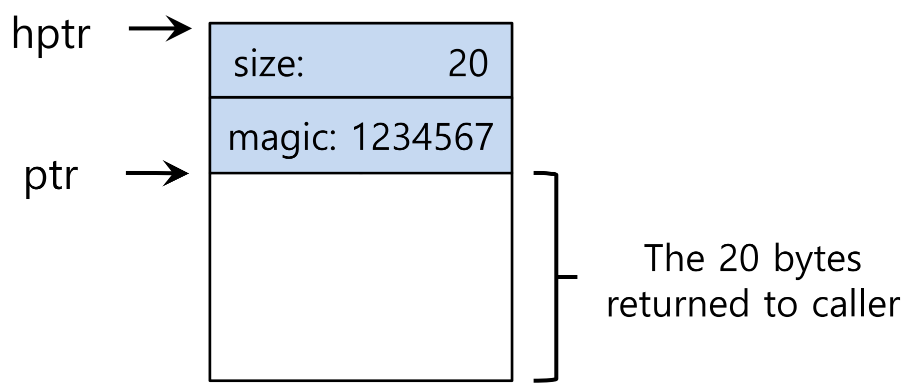


An Allocated Region Plus Header

The Header of Allocated Memory Chunk

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

Actual chuck size of malloc(N) = N + size of header
Here, 28 Byte



The Header of Allocated Memory Chunk(Cont.)

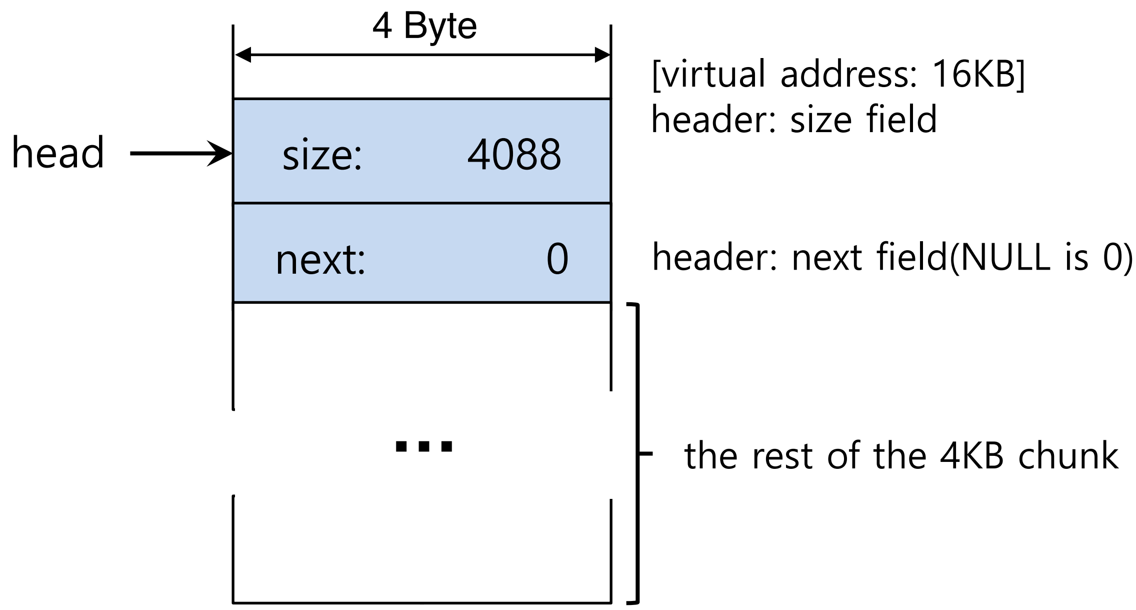
```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
    …
    assert(hptr->magic==1234567) ;
    …
}
```

Embedding A Free List

```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

Heap Initialization

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```



Embedding A Free List: Allocation

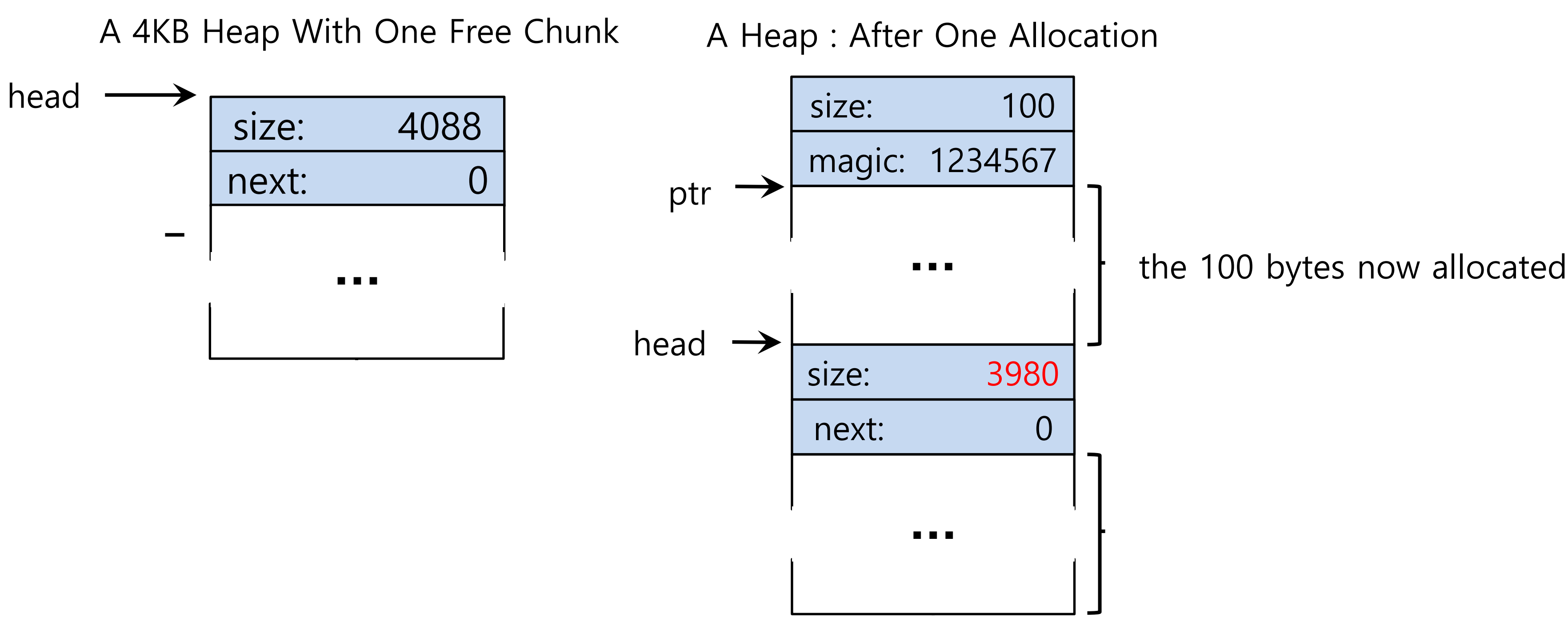
- If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request.

- The library will
  - **Split** the large free chunk into two.
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list.

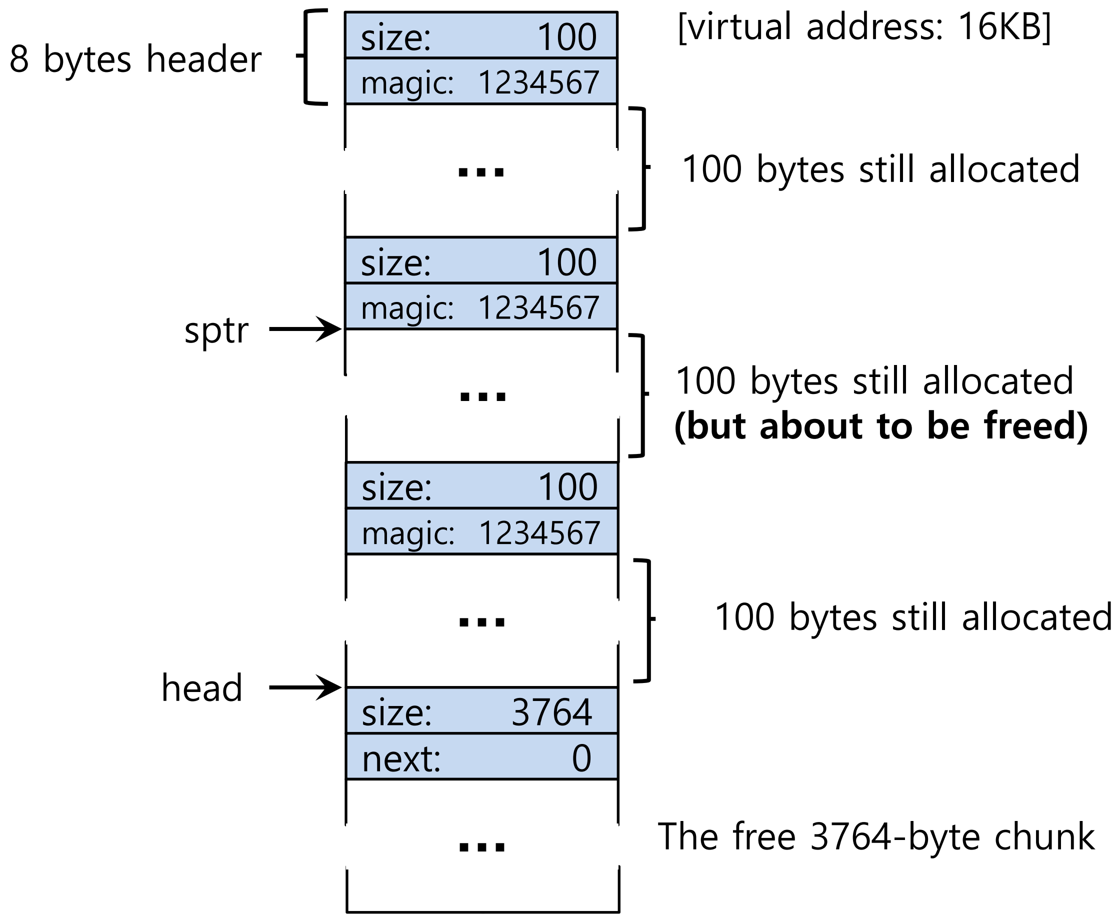
Embedding A Free List: Allocation(Cont.)

- Example: a request for 100 bytes by `ptr = malloc(100)`

  → 108 byte is returned.



Free Space With Chunks Allocated

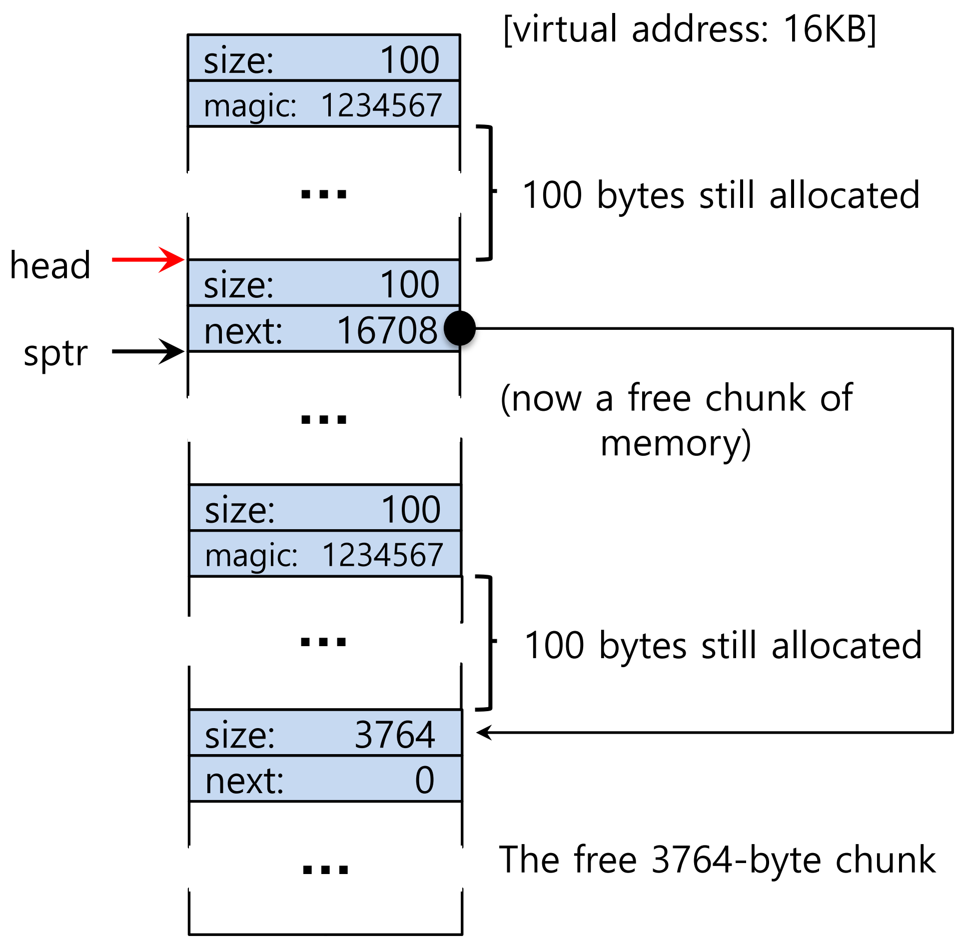


Free Space With Three Chunks Allocated

Free Space With `free()`

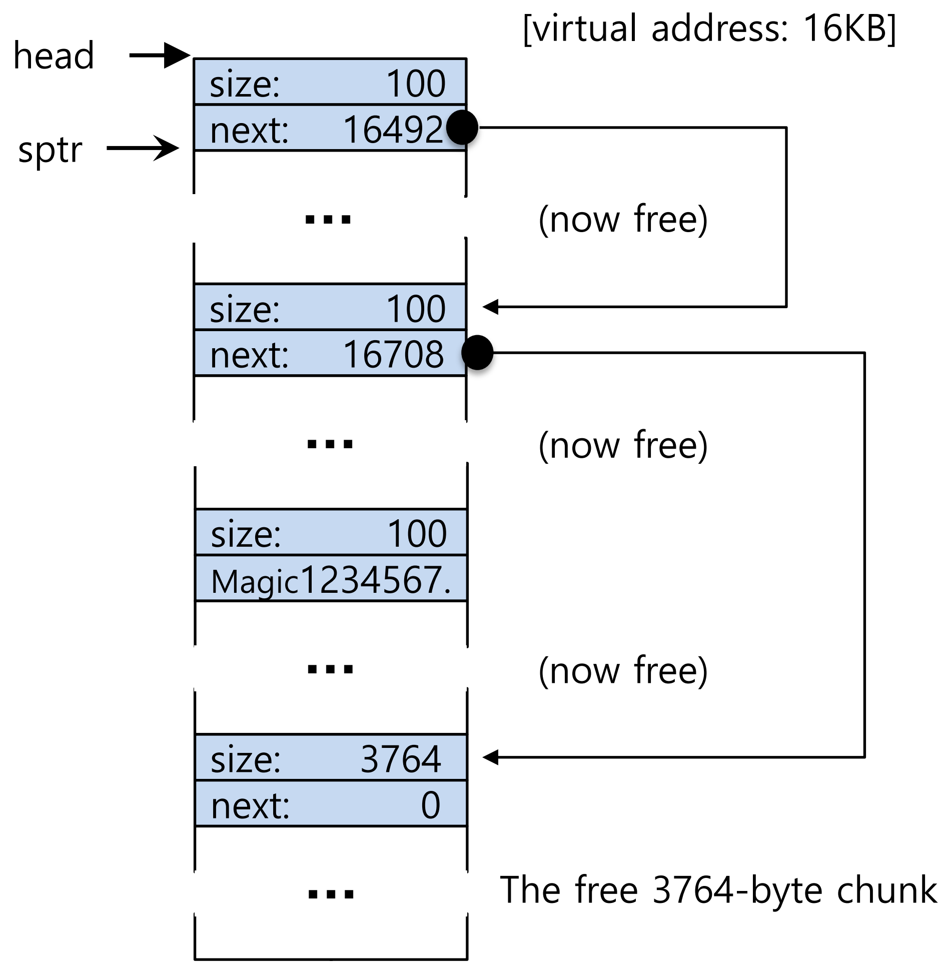
- `free(sptr)`

```
void* tmp = head ;
head = sptr ;
head->next = tmp ;
```



Free Space With Freed Chunks

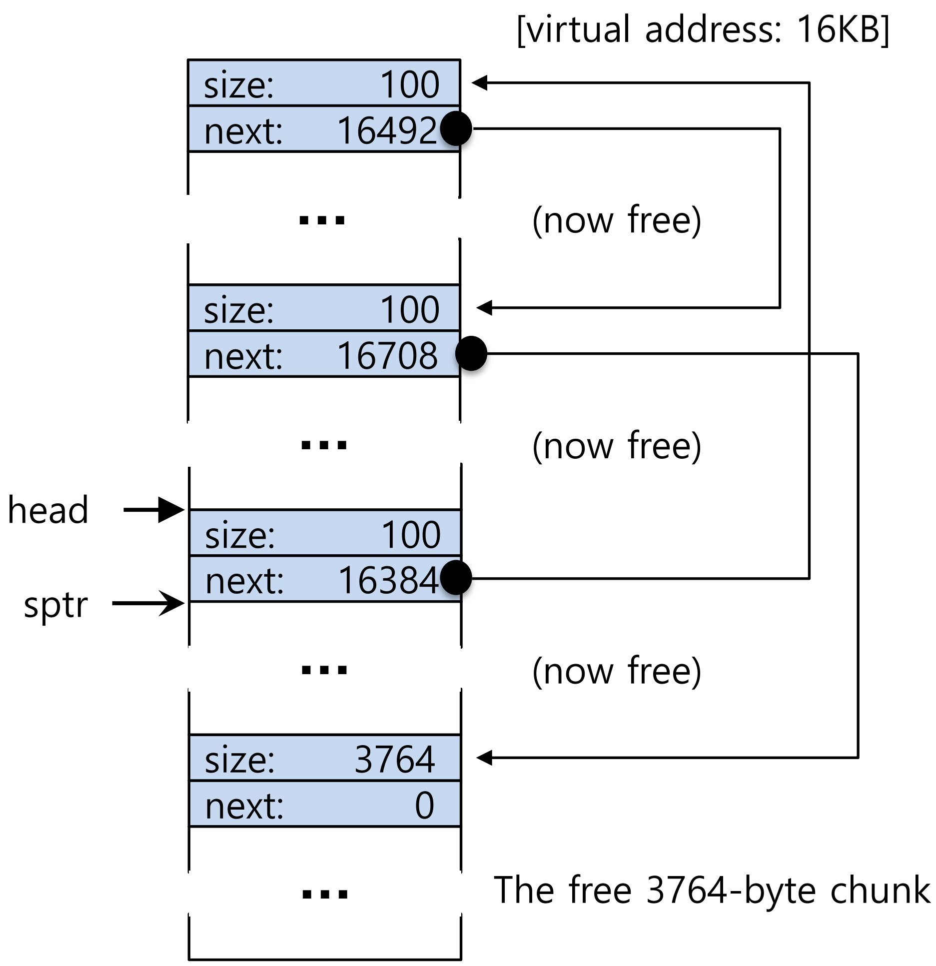
- `free(sptr)`



Free Space With Freed Chunks

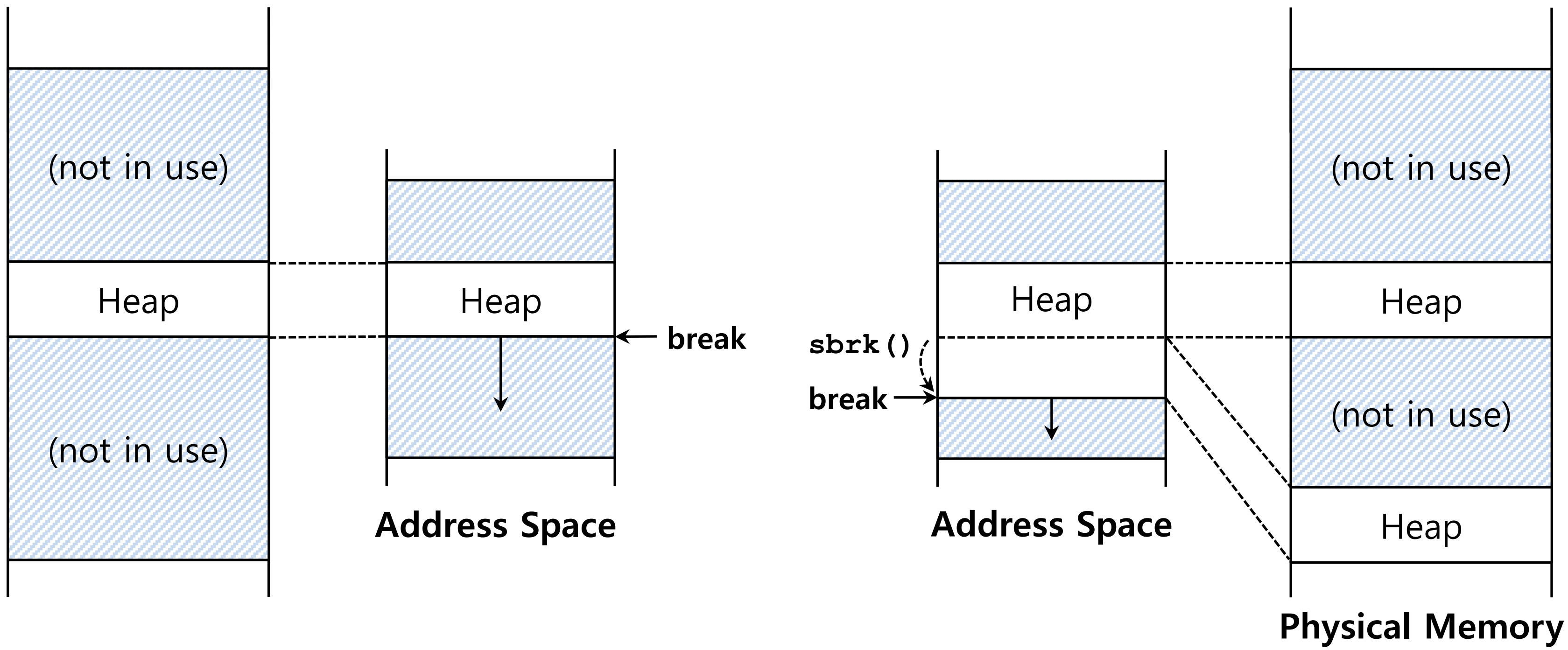
- free(sptr)

- **Coalescing** is needed in the list.



Growing The Heap

- Most allocators **start** with **a small-sized heap** and then **request more** memory from the OS when they run out.
  - e.g., `sbrk()`, `brk()` in most UNIX systems.



Managing Free Space: Basic Strategies

- Best Fit:
  - Finding free chunks that are **big or bigger than the request**
  - Returning the **one of smallest** in the chunks **in the group** of candidates

- Worst Fit:
  - Finding the **largest free chunk** and allocation the amount of the request
  - **Keeping the remaining chunk** on the free list.

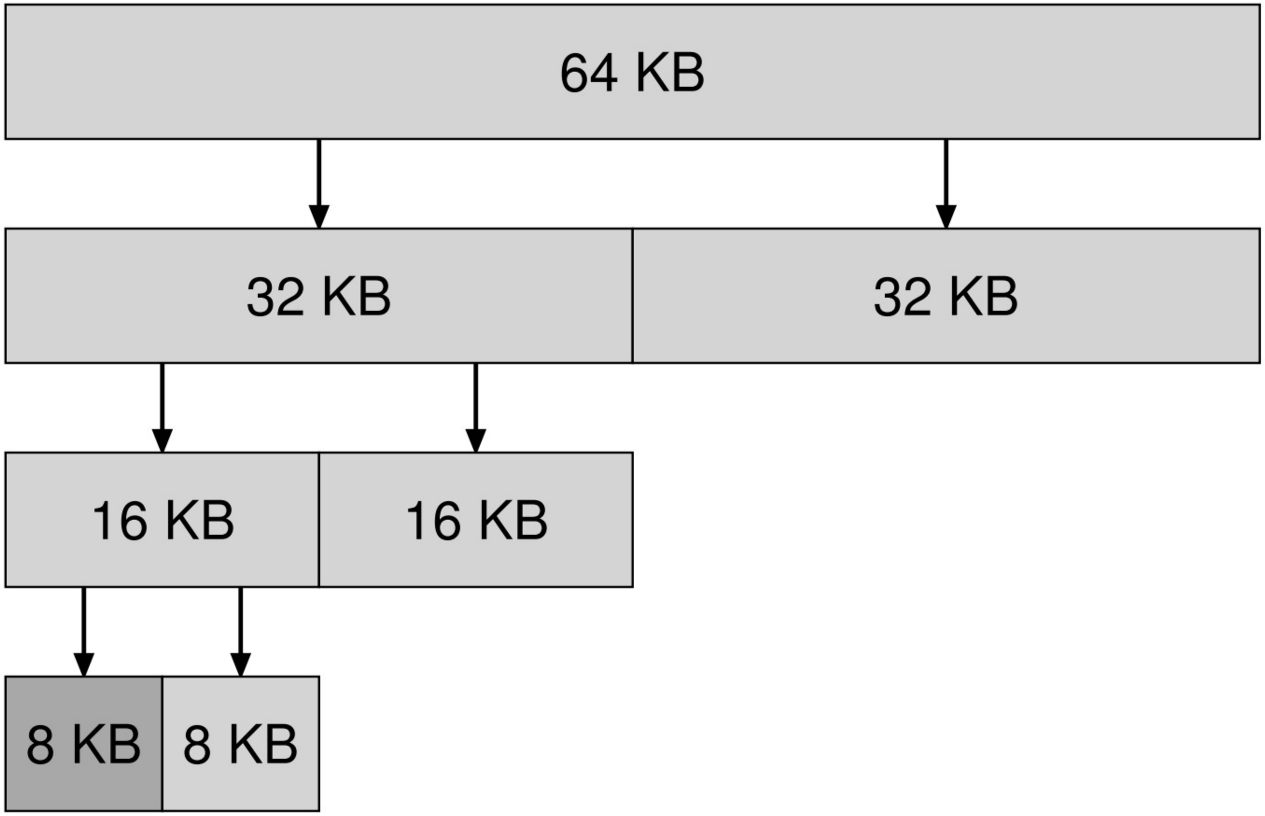
Managing Free Space: Basic Strategies(Cont.)

- First Fit:
  - Finding the **first chunk** that is **big enough** for the request
  - Returning the requested amount and remaining the rest of the chunk.

- Next Fit:
  - Finding the next chunk that is big enough for the request.
  - Searching at **where one was looking** at instead of the beginning of the list.

Buddy System

- Create the small buffers by repeatedly halving a large buffer and coalesce the adjacent free buffers.
- When a buffer is split, each half is called the buddy of the other.



Analysis

- Characteristic
  - Internal fragmentation by power-of-two allocation
  - Easy to find a buddy of a buffer by address and size
  - Use bitmap for coalescing.
- Advantage
  - Does a good job of coalescing adjacent free buffers.
  - Easy exchange of memory between the allocator and the paging system
- Disadvantage
  - Performance degrade: every time a buffer is released, the allocator tries to coalesce as much as possible.
  - Release routine needs both the address and size of the buffer.
  - Partial release is insufficient.

Heap overflow attacks

- What if heap memory is corrupted?
  - If a buffer is allocated on the heap and overflown, we could overwrite the heap meta data
  - This can allow us to modify any memory location with any value of our ch osen
  - This could lead to running arbitrary code

18. Paging: Introduction

Concept of Paging

- Paging **splits up** address space into **fixed-zed** unit called a **page**.
  - Segmentation: variable size of logical segments(code, stack, heap, etc.)

- With paging, **physical memory** is also **split** into some number of pages called a **page frame**.

- **Page table** per process is needed **to translate** the virtual address to physical address.

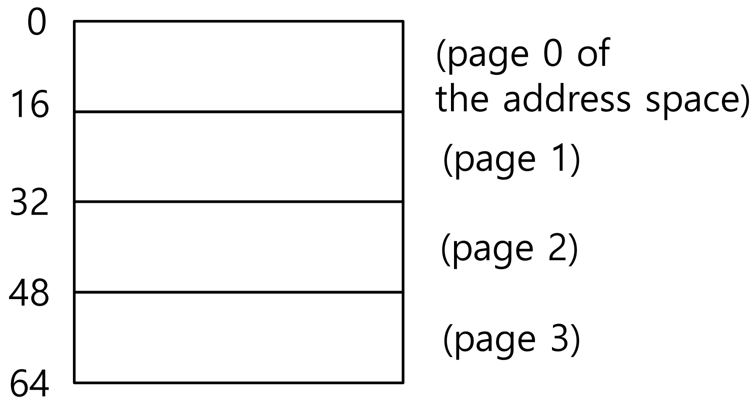
Advantages Of Paging

- **Flexibility:** Supporting the abstraction of address space effectively

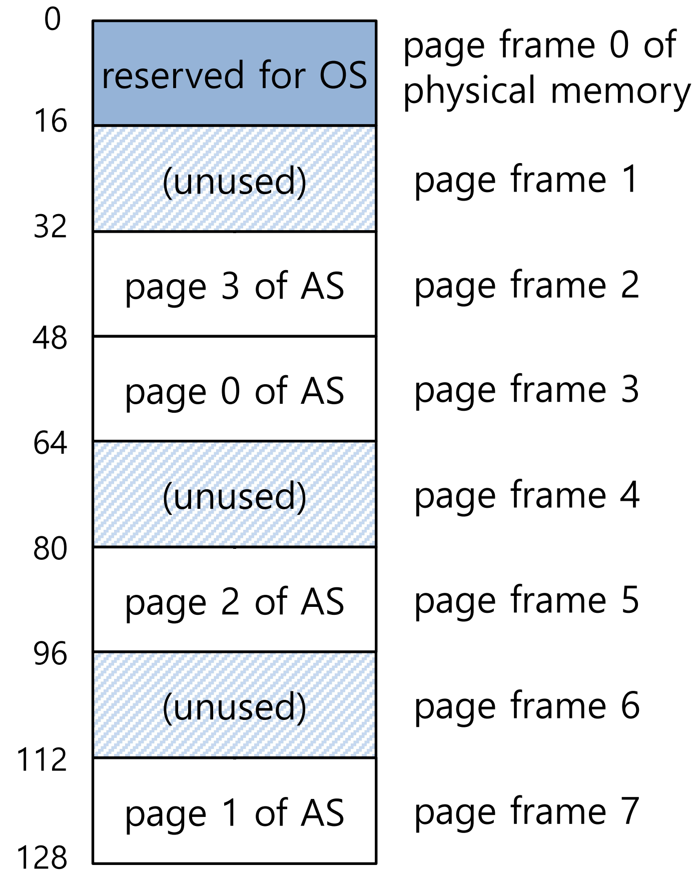
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages



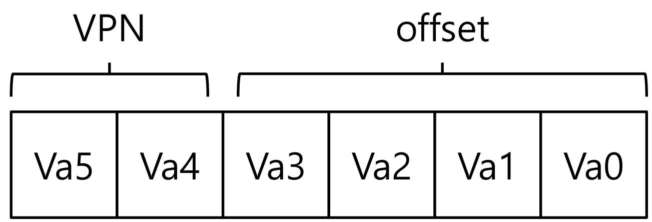
A Simple 64-byte Address Space



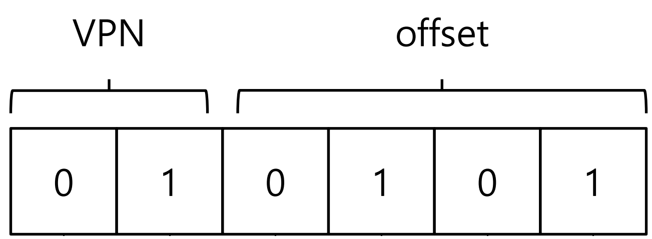
64-Byte Address Space Placed In Physical Memory

Address Translation

- Two components in the virtual address
  - VPN: virtual page number
  - Offset: offset within the page

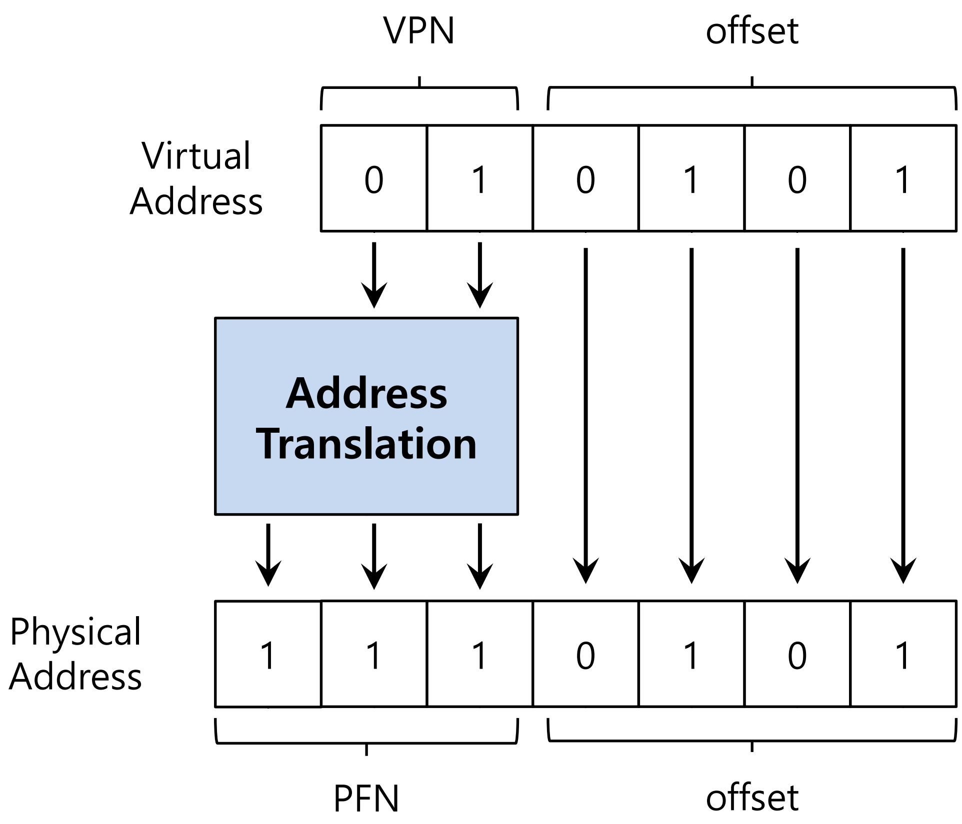


- Example: virtual address 21 in 64-byte address space



Example: Address Translation

- The virtual address 21 in 64-byte address space



Where Are Page Tables Stored?

- Page tables can get awfully large.
  - 32-bit address space with 4-KB pages, 20 bits for VPN
    - Page offset for 4 Kbyte page: 12 bit
    - $4MB = 2^{20}\ entries\ * 4\ Bytes\ per\ page\ table\ entry$

- Page tables for each process are stored in memory.

What Is In The Page Table?

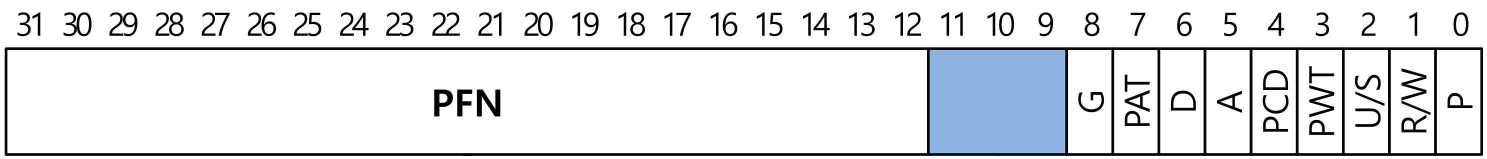
- The page table is a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

- The OS **indexes** the array by VPN, and looks up the page-table entry.

Common Flags Of Page Table Entry

- **Valid Bit**: Indicating whether the particular translation is valid.
- **Protection Bit**: Indicating whether the page could be read from, written to, or executed from
- **Present Bit**: Indicating whether this page is in physical memory or on disk(swapped out)
- **Dirty Bit**: Indicating whether the page has been modified since it was brought into memory
- **Reference Bit(Accessed Bit):** Indicating that a page has been accessed

Example: x86 Page Table Entry



An x86 Page Table Entry(PTE)

- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

Paging: Too Slow

- To find a location of the desired PTE, the **starting location** of the page table is **needed**.

- For every memory reference, paging requires to perform one or more **extra memory reference**.

Accessing Memory With Paging

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

```

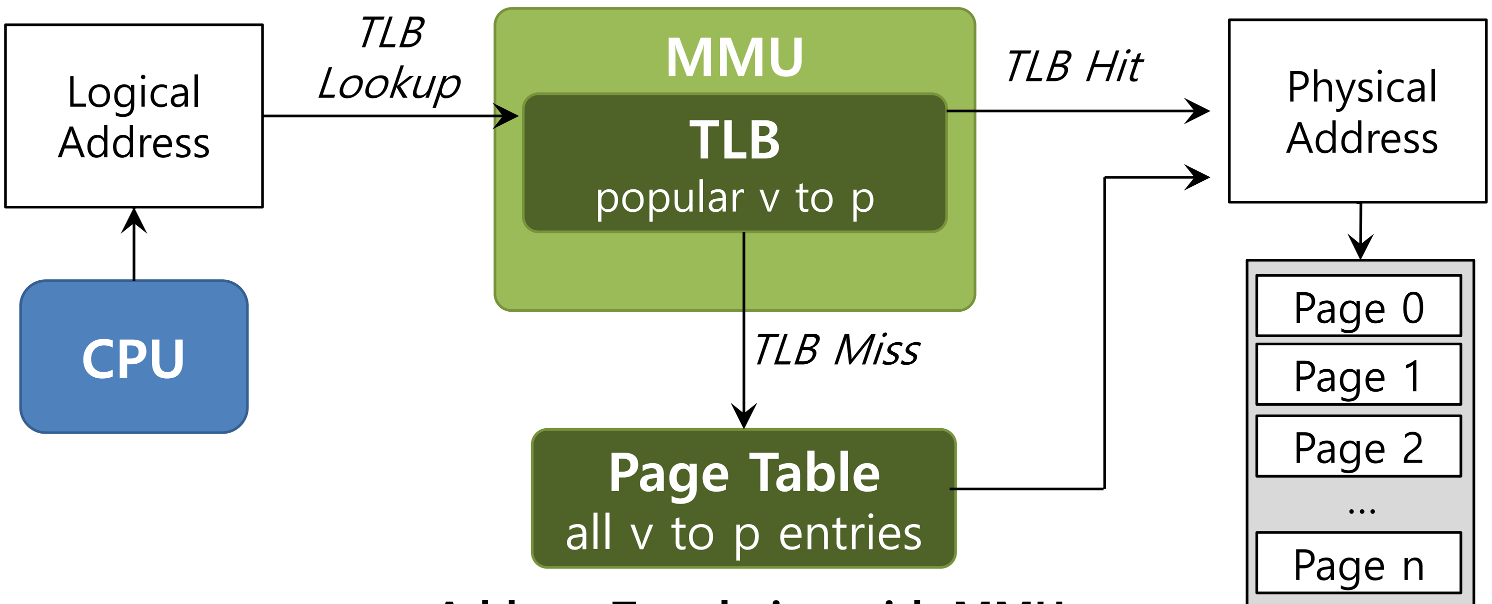
Accessing Memory With Paging

```
// Check if process can access the page
if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else
        // Access is OK: form physical address and fetch it
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```

19. Translation Lookaside Buffer

TLB

- Part of the chip's memory-management unit(MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

Physical Memory

TLB Basic Algorithms

```
VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
 (Success , TlbEntry) = TLB_Lookup(VPN)
    if(Success == TRUE){ // TLB Hit
        if(CanAccess(TlbEntry.ProtectBit) == True ){
            offset = VirtualAddress & OFFSET_MASK
            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
            AccessMemory( PhysAddr )
        }else RaiseException(PROTECTION_ERROR)
```

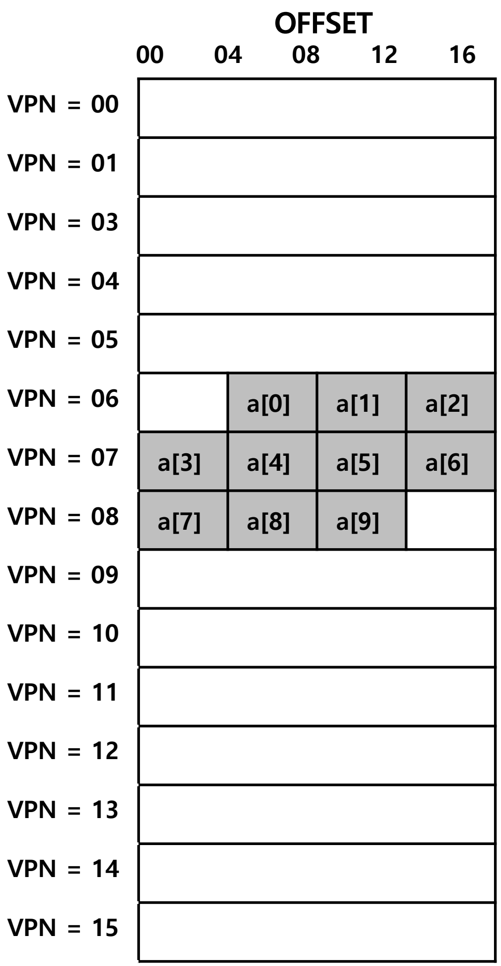
TLB Basic Algorithms (Cont.)

```
11:    }else{ //TLB Miss
12:        PTEAddr = PTBR + (VPN * sizeof(PTE))
13:        PTE = AccessMemory(PTEAddr)
14:        if(PTE.Valid == False)
              RaiseException SEGFAULT) ;
15:        else{
16:            TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:            RetryInstruction()
18:        }
19:    }
```

- (11-12 lines) The hardware accesses the page table to find the translation.
- (16 lines) updates the TLB with the translation.

Example: Accessing An Array

- How a TLB can improve its performance.




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

The TLB improves performance due to spatial and temporal locality

3 misses and 7 hits.
Thus TLB hit rate is 70%.

Who Handles The TLB Miss?

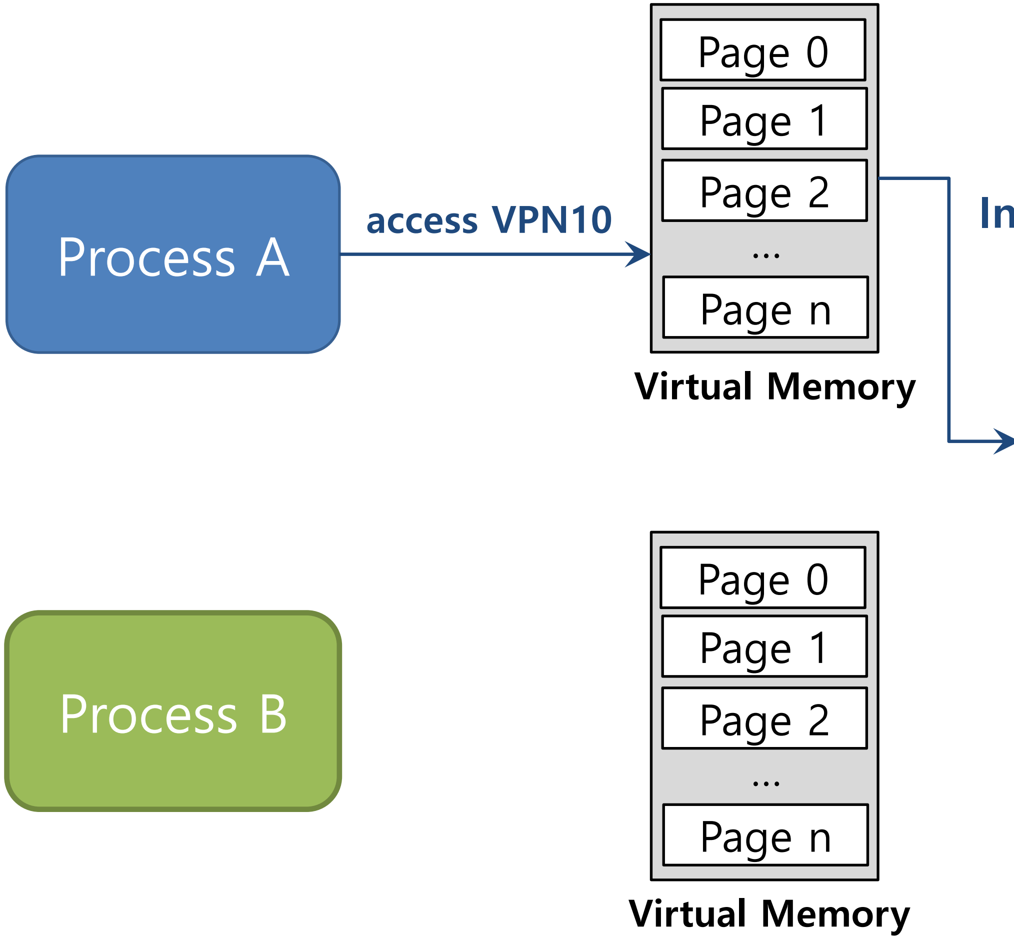
- Hardware handles the TLB miss entirely on CISC.
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**
  - **Intel x86**
- RISC has what is known as a **software-managed TLB.**
  - On a TLB miss, the hardware raises exception (trap handler).
    - **Trap handler is code** within the OS that is written with the express purpose of handling TLB miss.

TLB entry

- TLB is managed by **Full Associative** method.
  - A typical TLB has 32,64, or 128 entries.
  - Hardware searches the entire TLB in parallel to find the desired translation.
  - other bits: valid bits , protection bits, address-space identifier, dirty bit

VPN	PFN	other bits

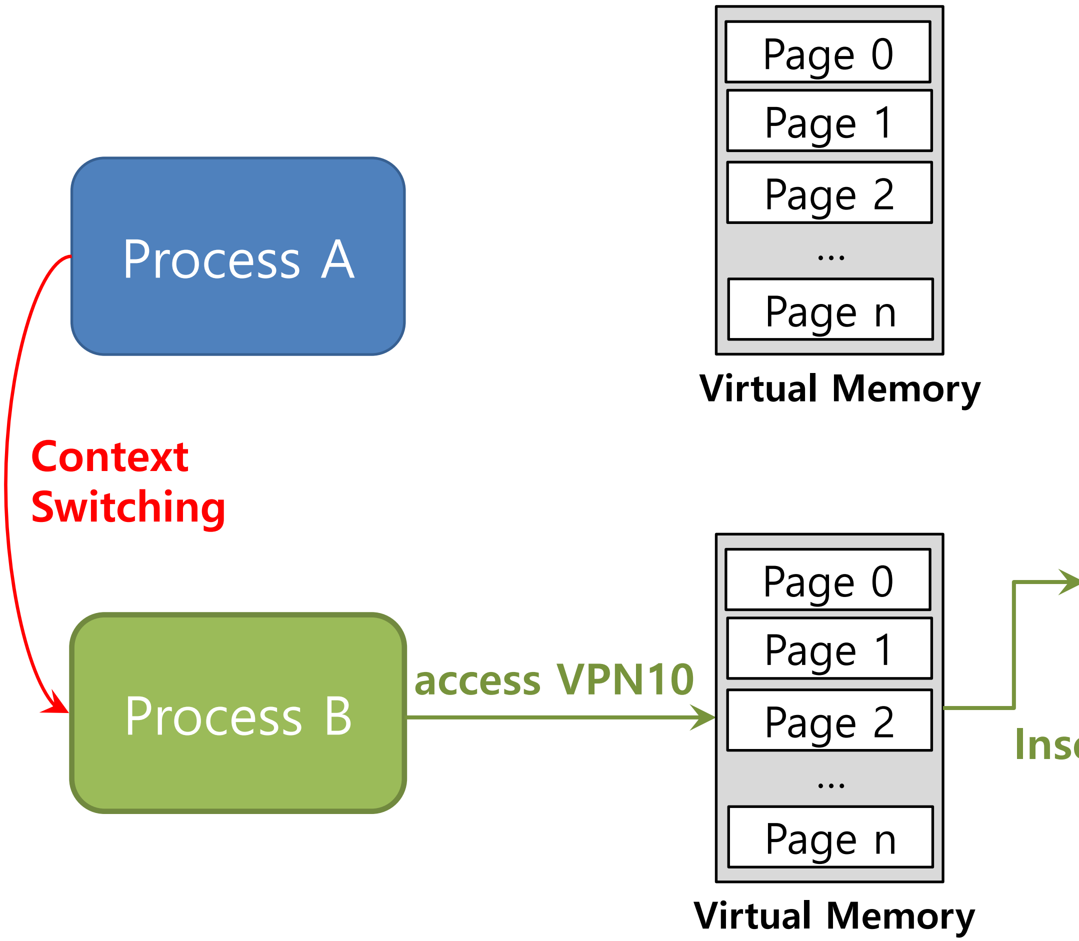
TLB Issue: Context Switching



TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
-	-	-	-
-	-	-	-

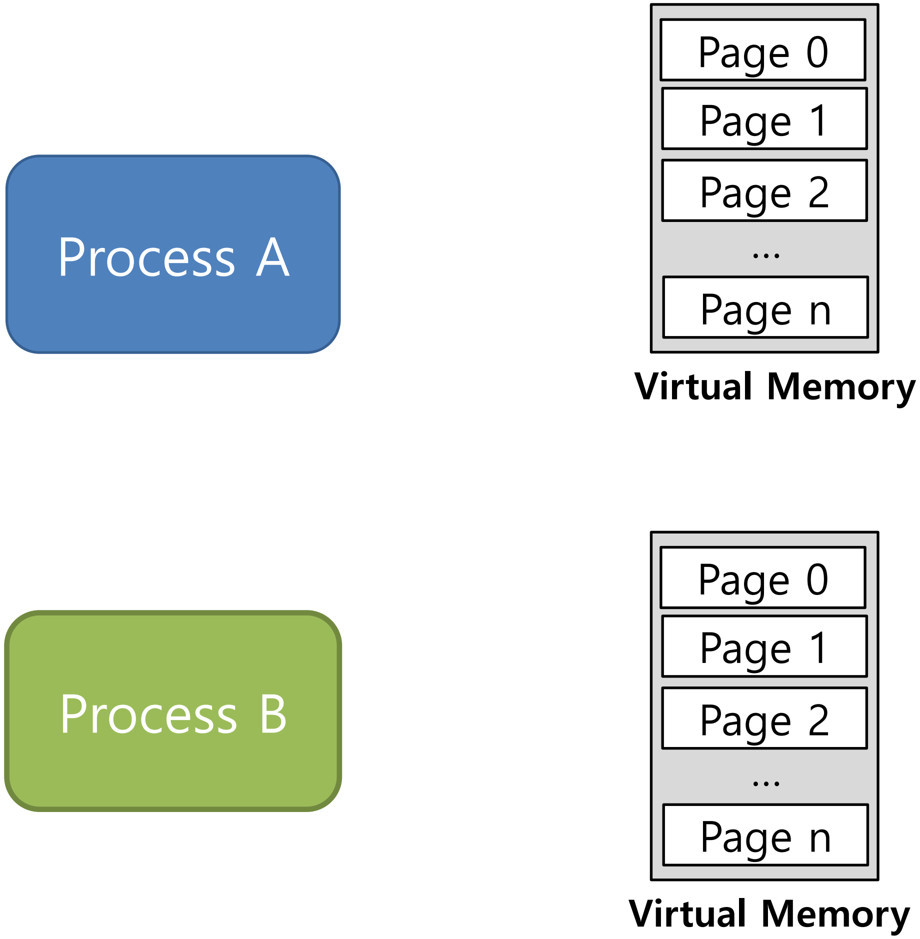
TLB Issue: Context Switching



TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

TLB Issue: Context Switching



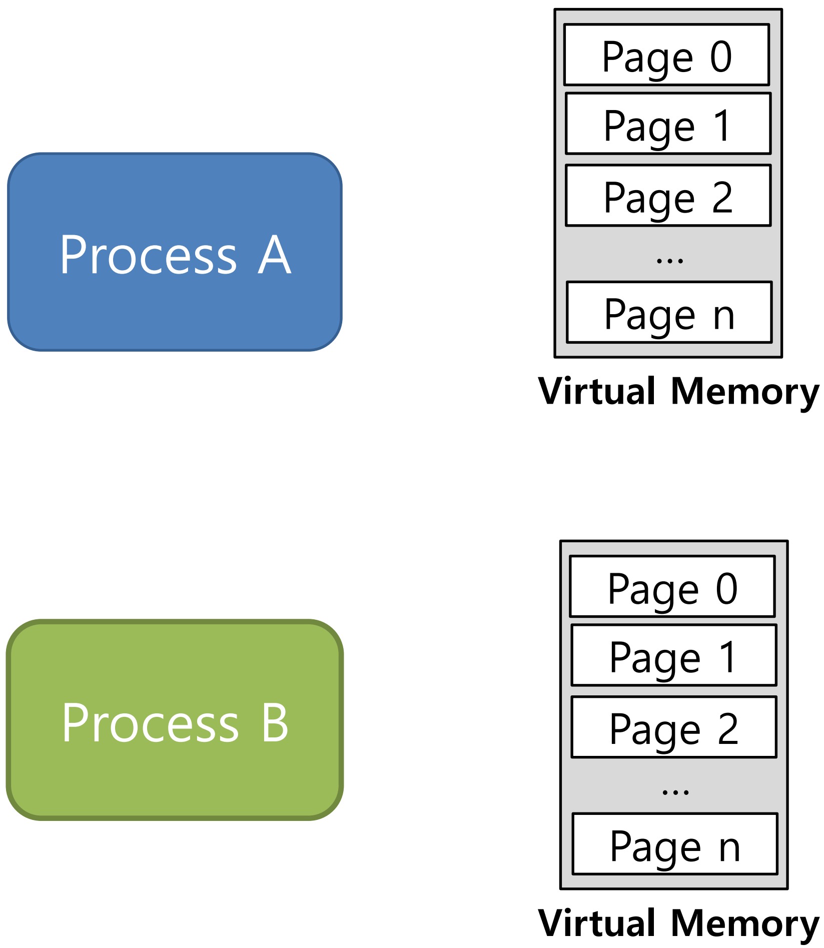
TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

To Solve Problem

- Provide an address space identifier(ASID) field in the TLB.

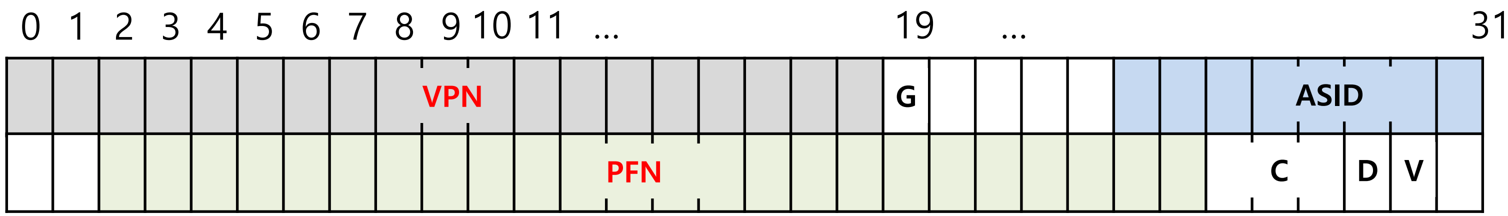


TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

A Real TLB Entry

All 64 bits of this TLB entry(example of MIPS R4000)



Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory(pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

The END