CSC 252: Computer Organization
Spring 2020: Lecture 23

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Lab5: https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment5.html

19	20	21 **Today**	22	23	24	25
26	27	28 **Last Lecture**	29	30	May 1	2
3	4 **Due**	5	6 **Final**	7	8	9

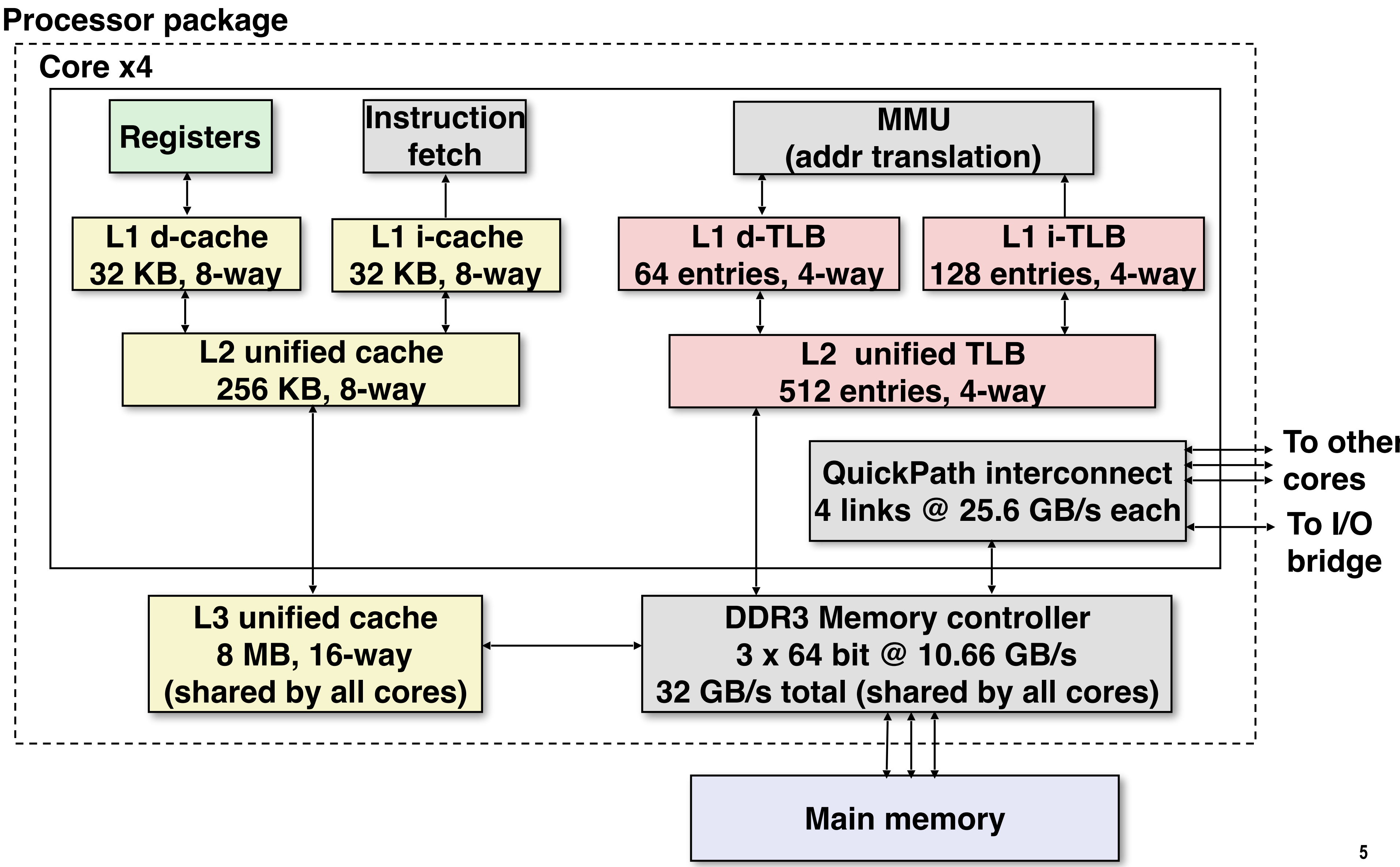
Announcements

- Virtual memory problem set and solution: https://www.cs.rochester.edu/courses/252/spring2020/handouts.html
- Final exam: May. 6, 19:15 EST — 22:15 EST
- Let me know if you can't make this time.
- Exam will be electronic on blackboard, but we will send you an PDF version so that you can work offline in case
  - 1) you don't have Internet access at the exam time or
  - 2) you lose Internet access.
  - Write down the answers on a scratch paper, take pictures, and send us the pictures
- Same rule as before: anything on paper is fine, nothing electronic other than using the computer to take the exam
- Will do a dry run on Apr. 28 during the class

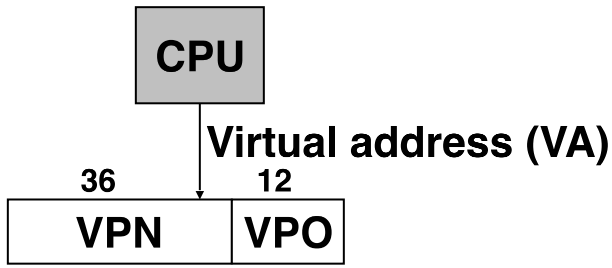
Today

- Case study: Core i7/Linux memory system
- Memory mapping
- Dynamic memory allocation

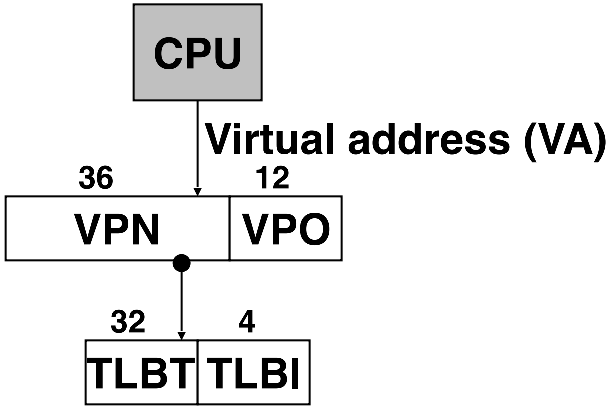
Intel Core i7 Memory System



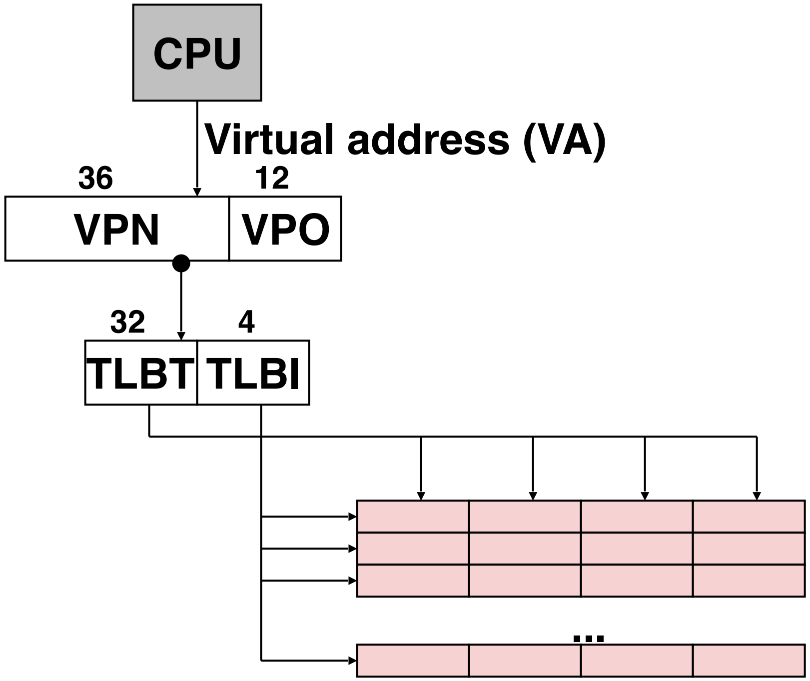
End-to-End Core i7 Address Translation



End-to-End Core i7 Address Translation

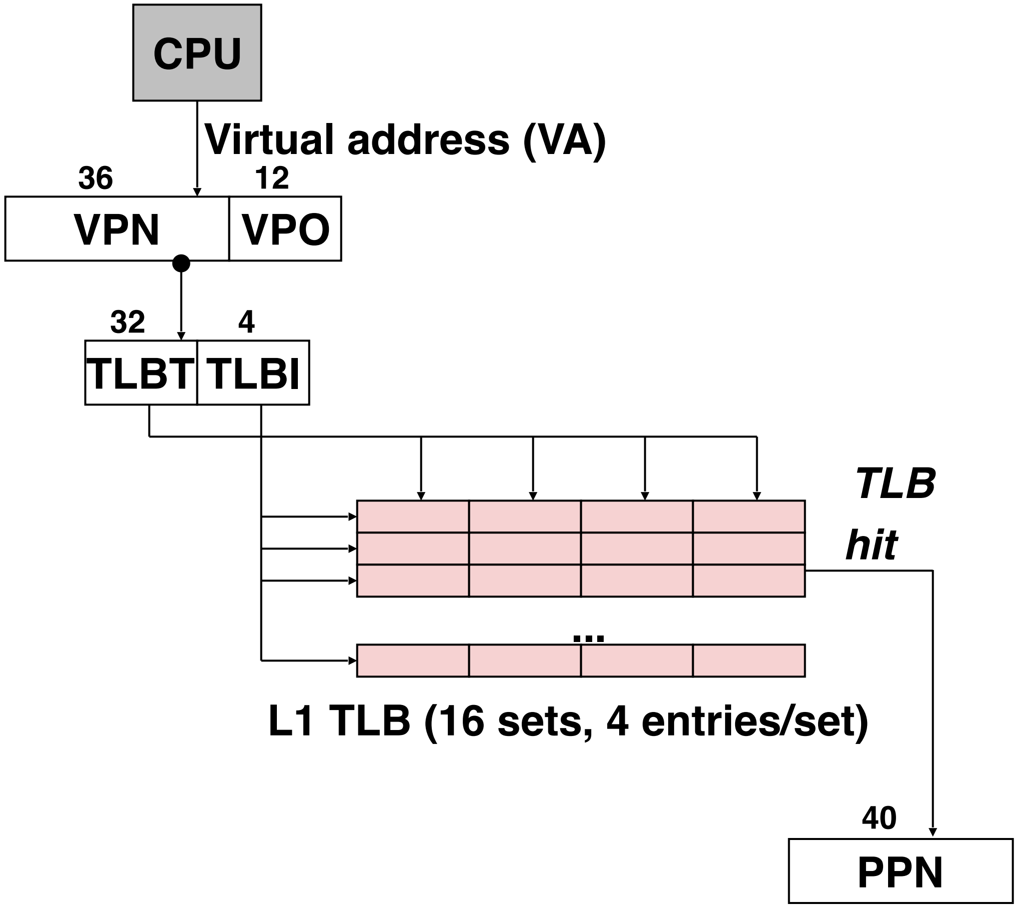


End-to-End Core i7 Address Translation

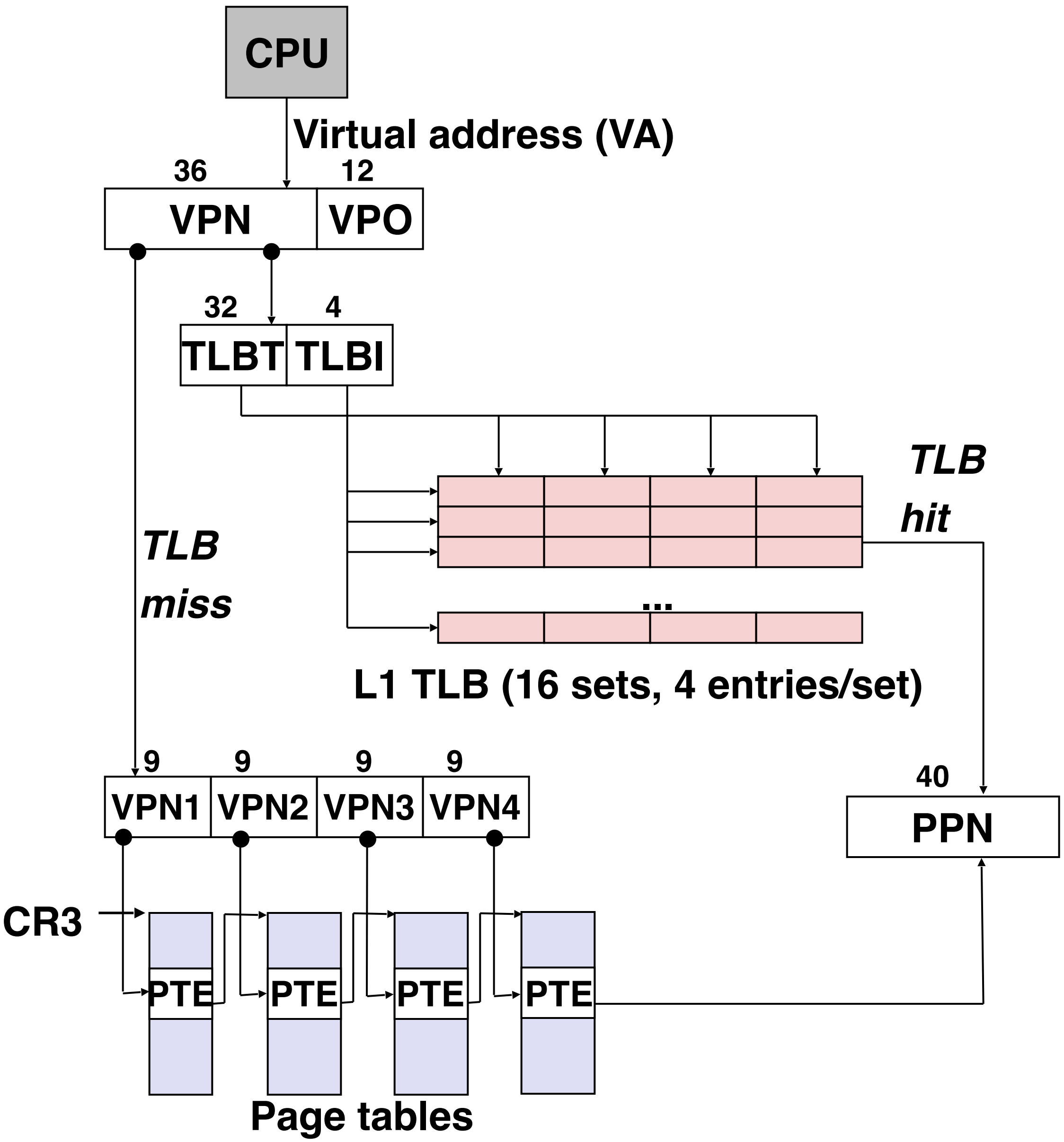


L1 TLB (16 sets, 4 entries/set)

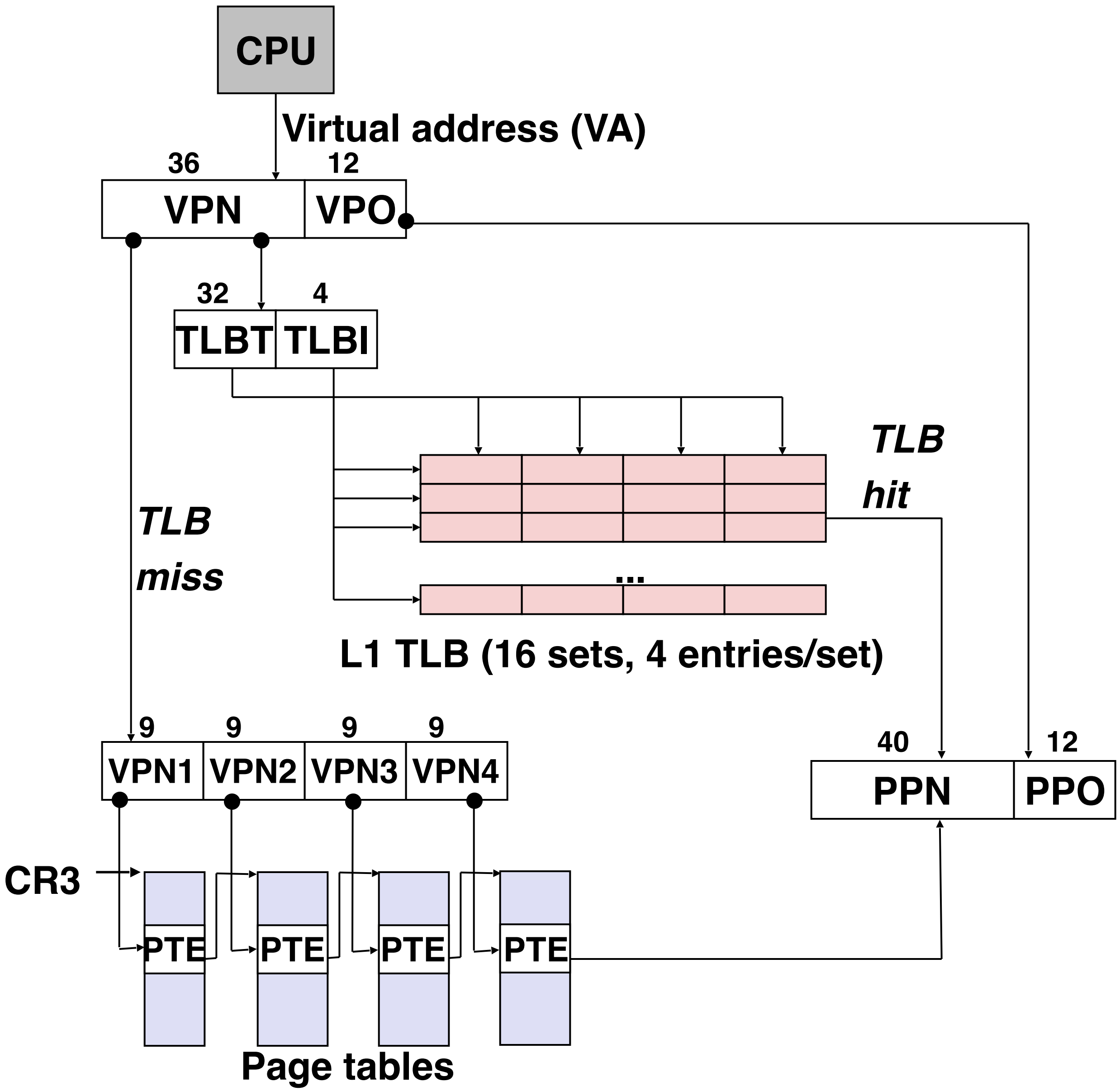
End-to-End Core i7 Address Translation



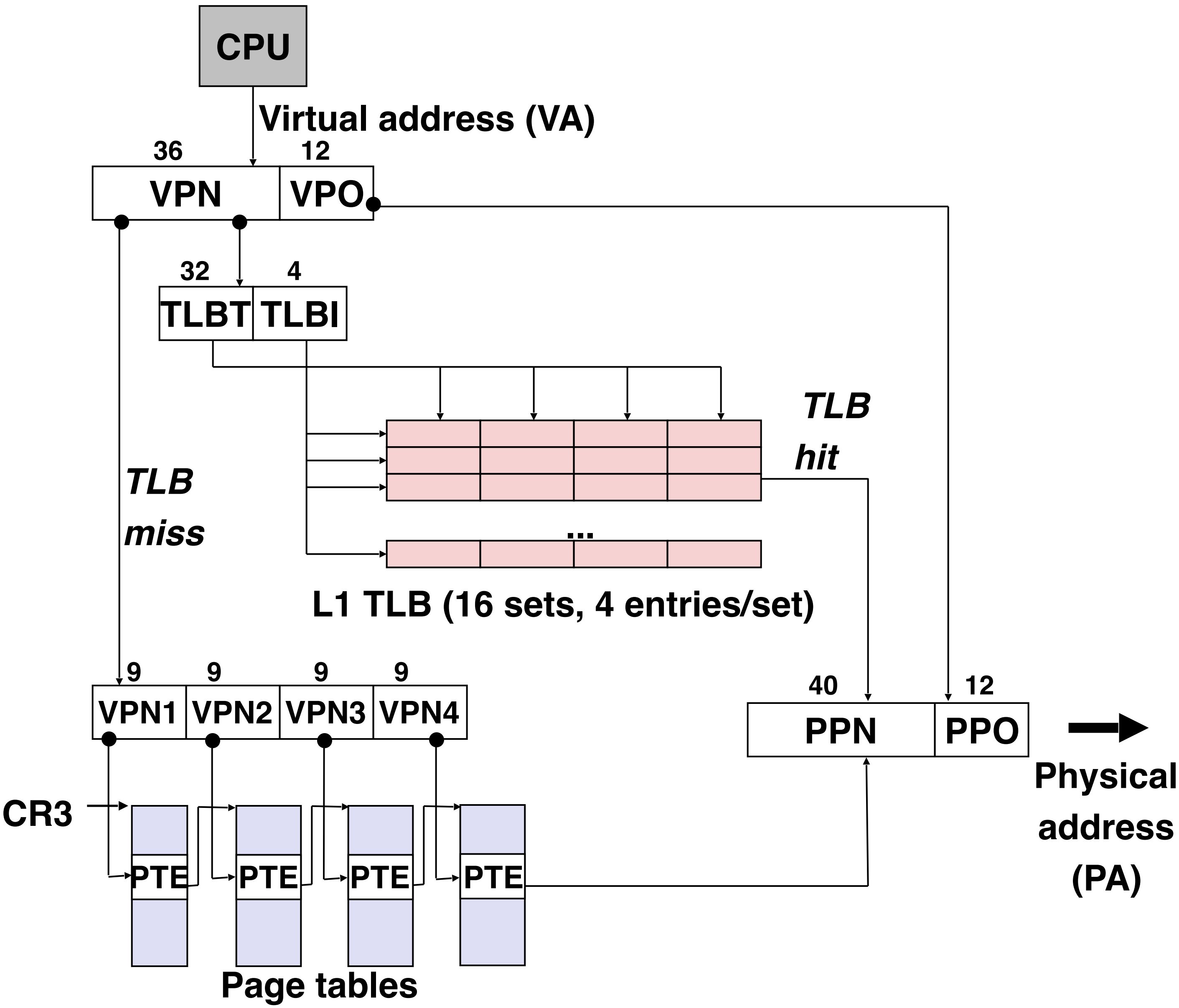
End-to-End Core i7 Address Translation



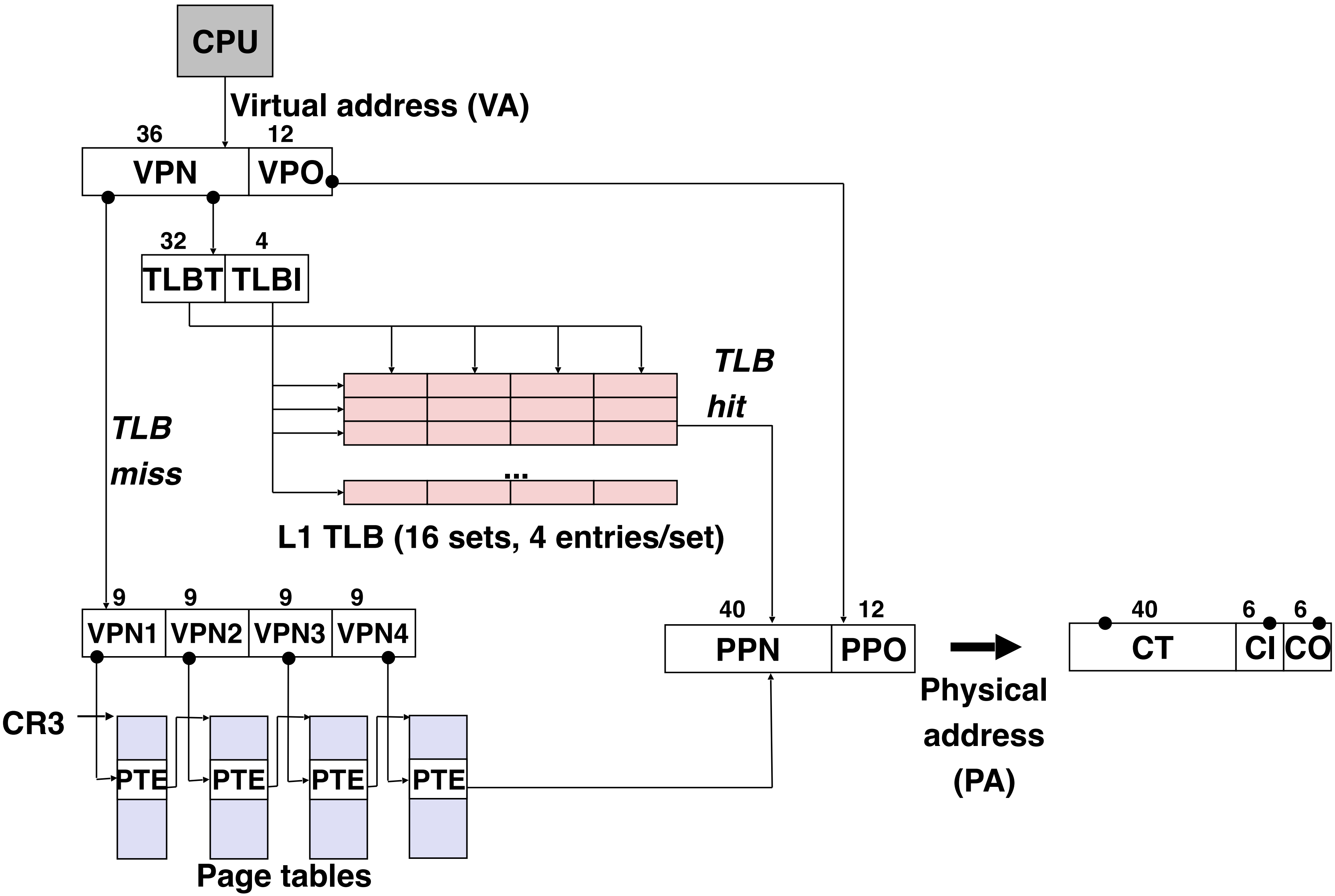
End-to-End Core i7 Address Translation



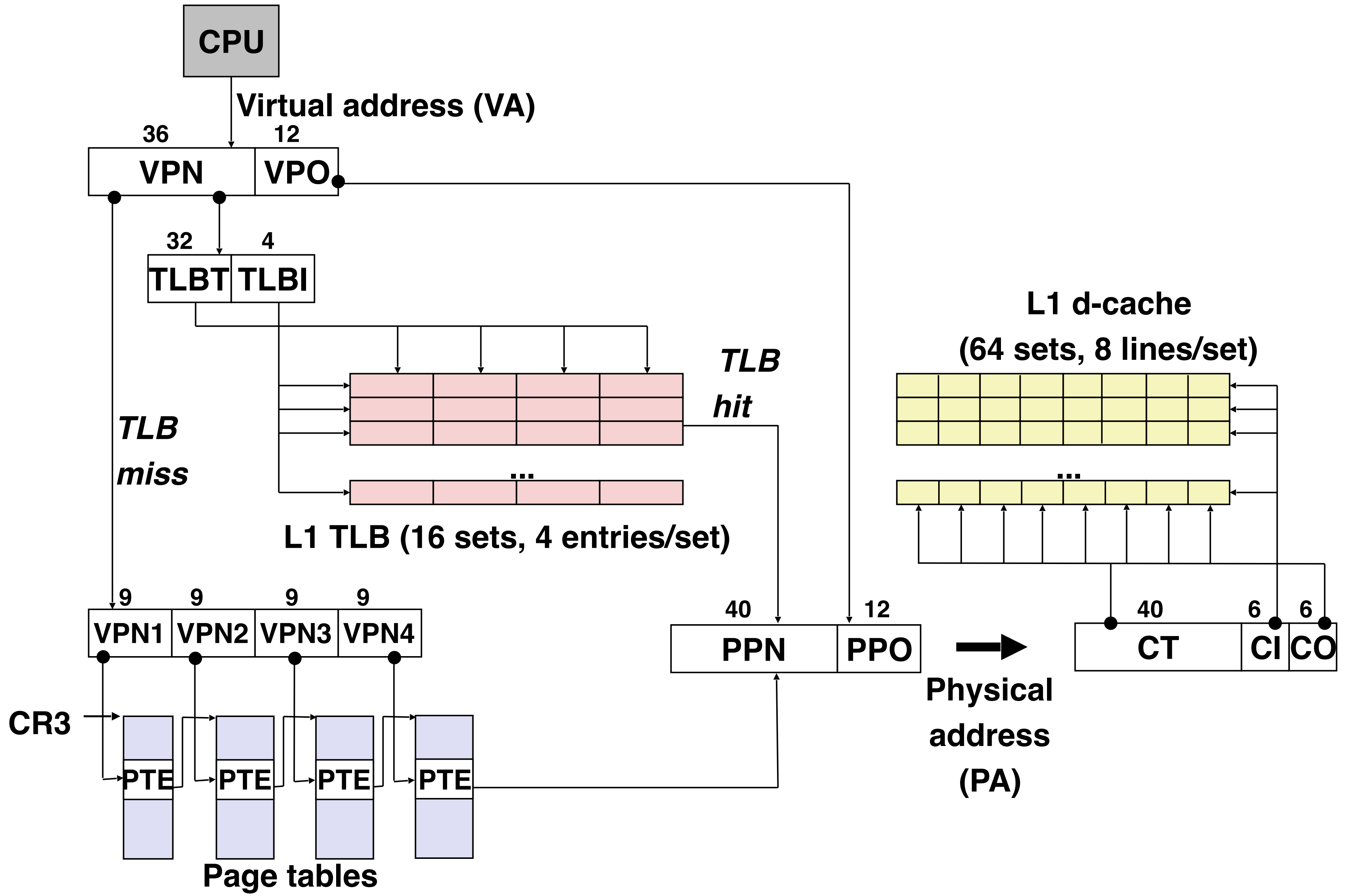
End-to-End Core i7 Address Translation



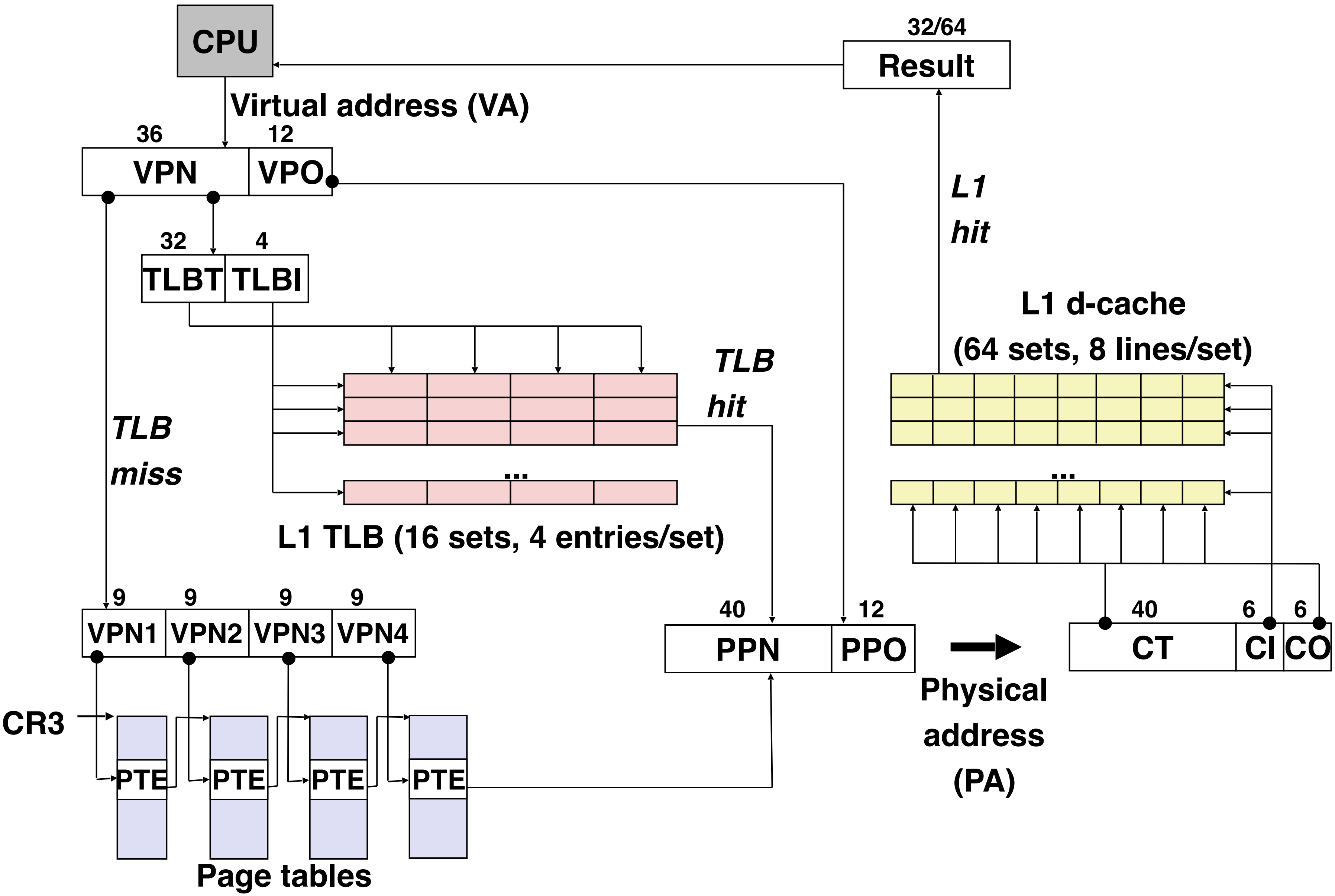
End-to-End Core i7 Address Translation



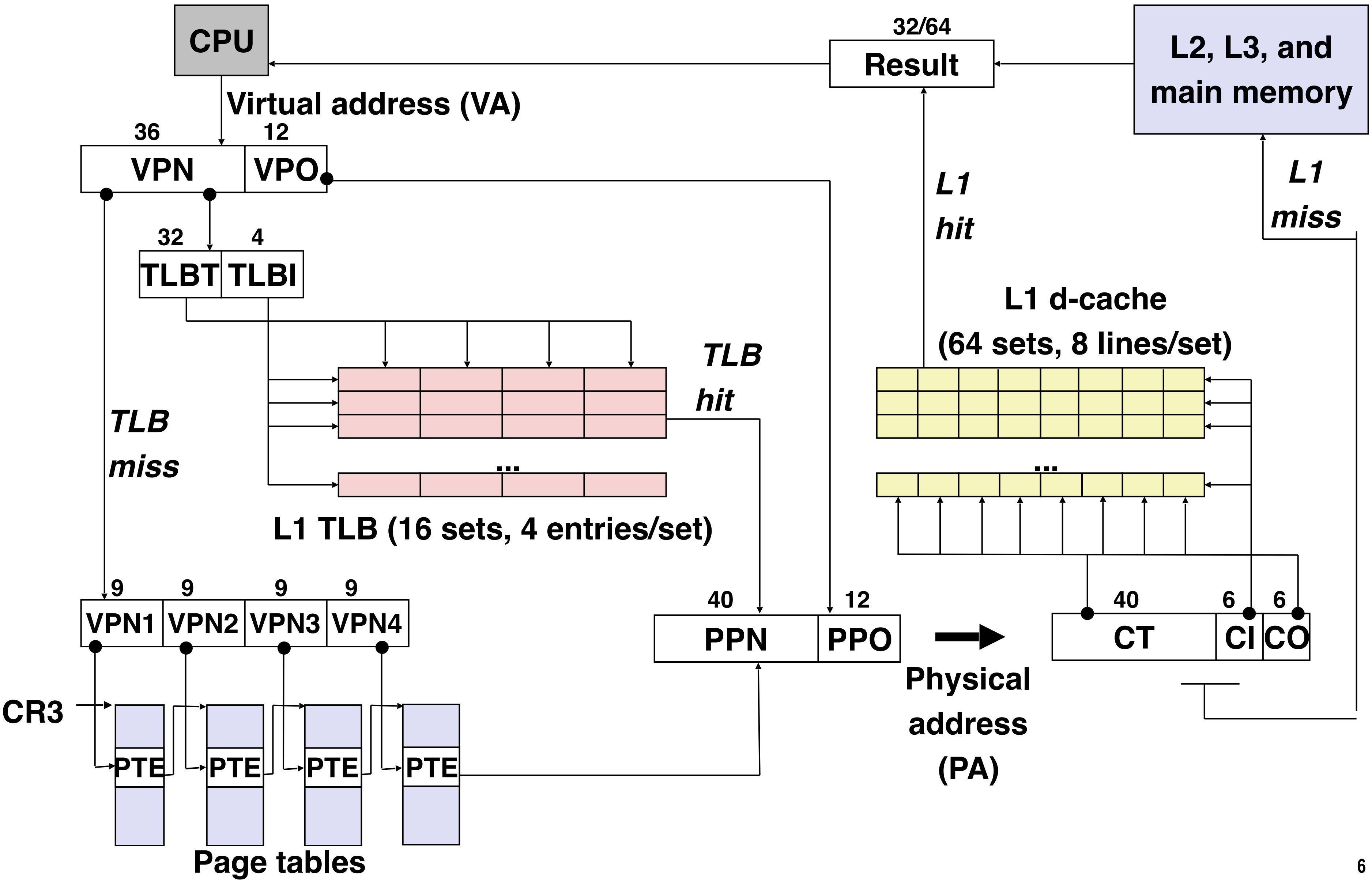
End-to-End Core i7 Address Translation



End-to-End Core i7 Address Translation



End-to-End Core i7 Address Translation



Core i7 Level 4 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1

Available for OS (page location on disk)	P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

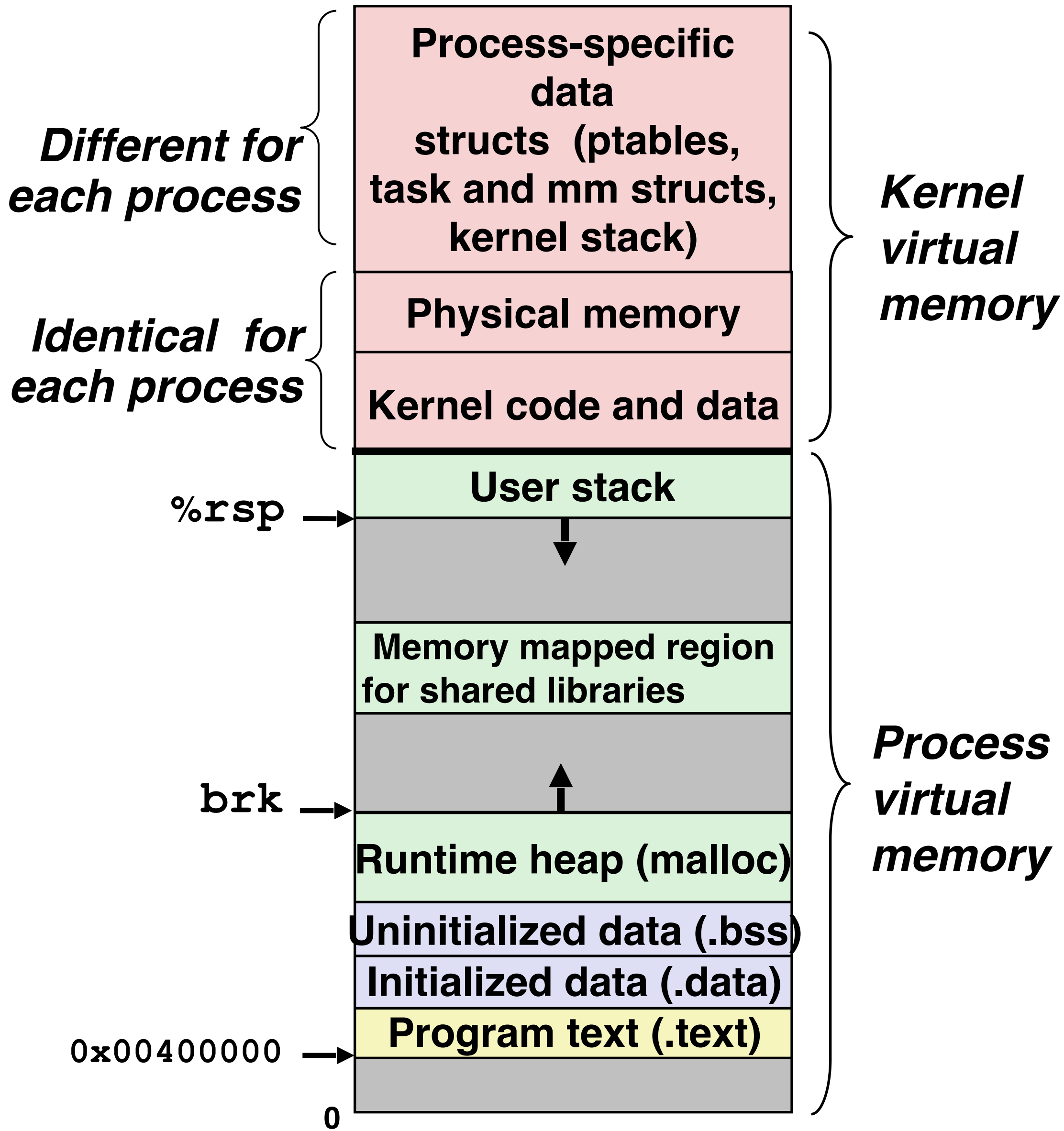
A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Virtual Address Space of a Linux Process



Today

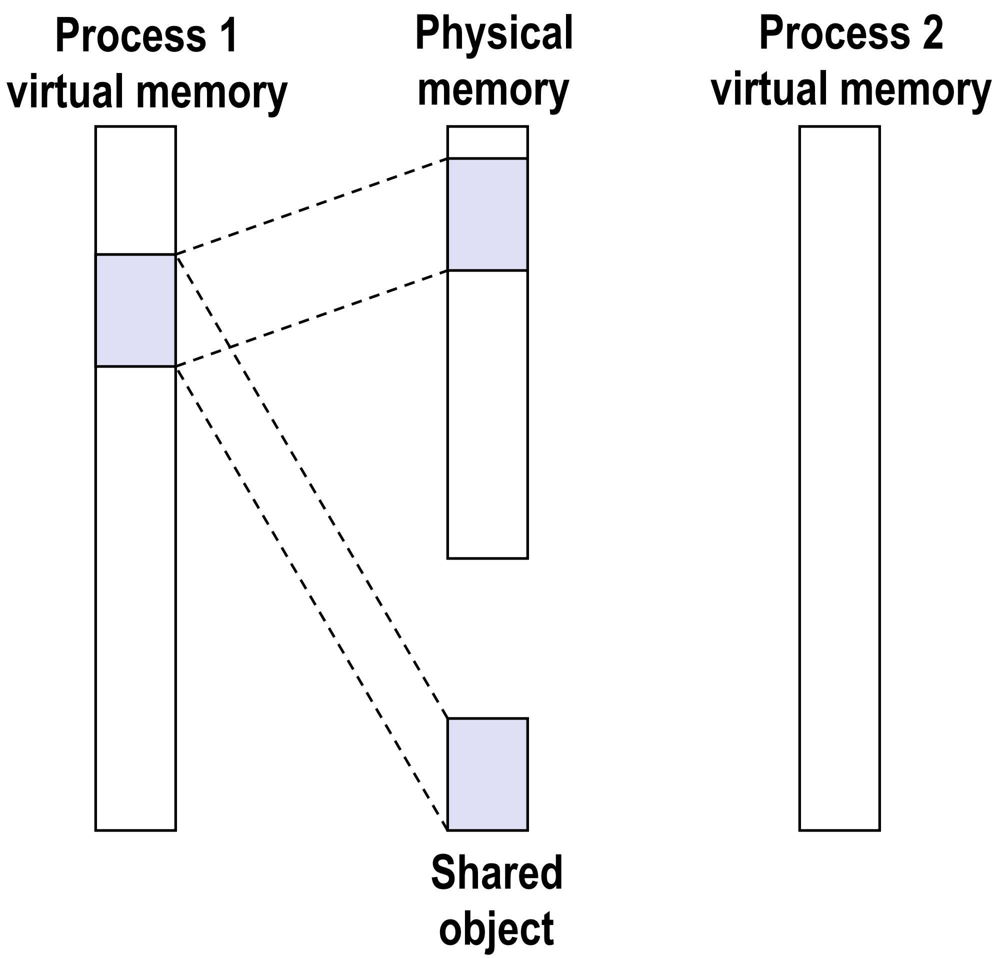
- Case study: Core i7/Linux memory system
- Memory mapping
- Dynamic memory allocation

Memory Mapping For Sharing

- Multiple processes often share data
  - Different processes that run the same code (e.g., shell)
  - Different processes linked to the same standard libraries
  - Different processes share the same file
- It is wasteful to create exact copies of the share object
- Memory mapping allow us to easily share objects
  - Different VM pages point to the same physical page/object

Sharing Revisited: Shared Objects

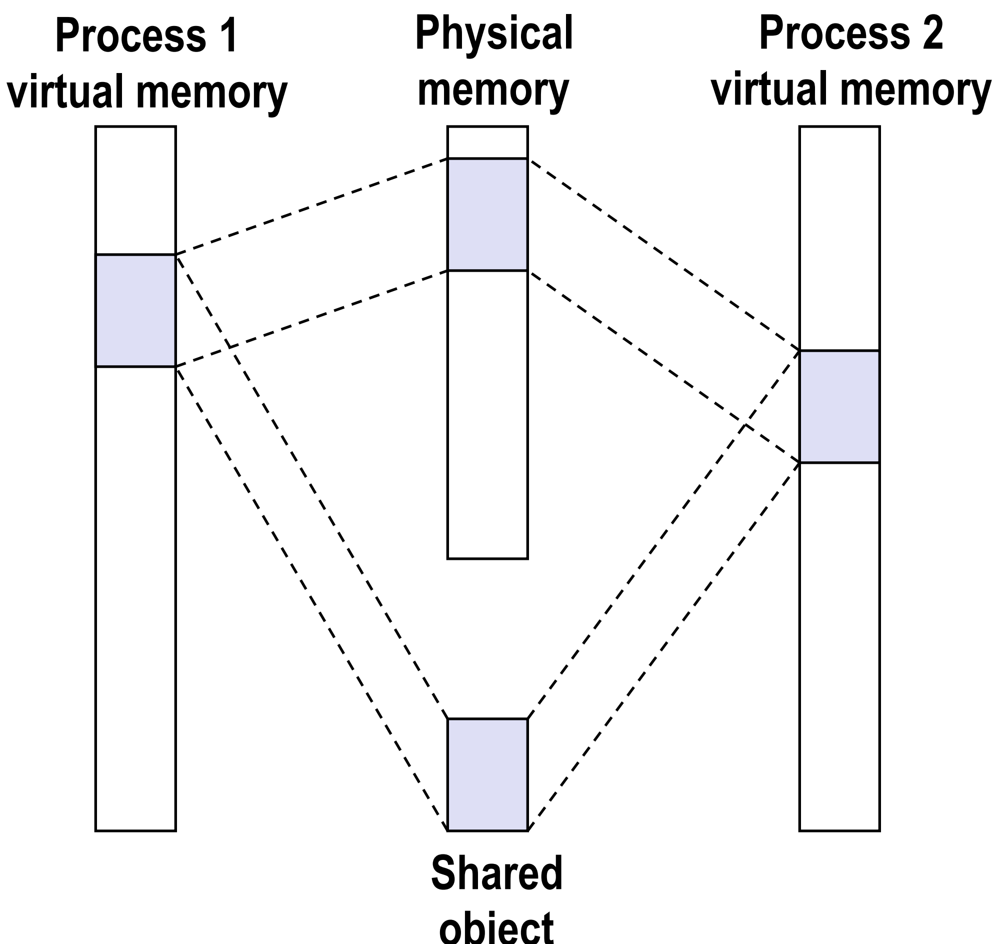
- Process 1 maps the shared object.



- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.

Sharing Revisited: Shared Objects

- Process 2 maps the shared object.



- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.
- Now when Proc. 2 wants to access the same object, the kernel can simply point the PTEs of Proc. 2 to the already-mapped physical pages.

The Problem…

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2

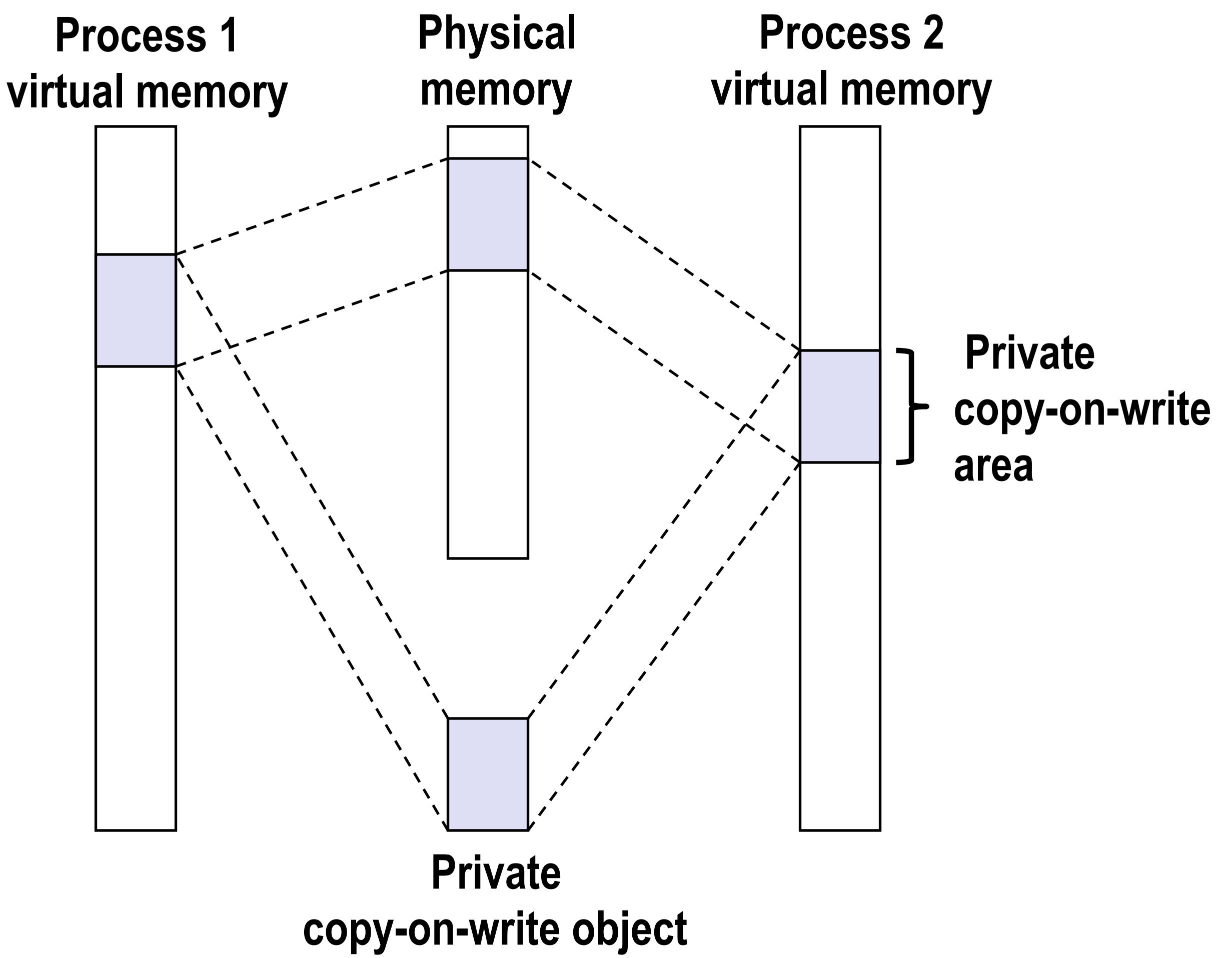
The Problem…

- What if Proc. 1 now wants to modify the shared object, but doesn't want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.

The Problem…

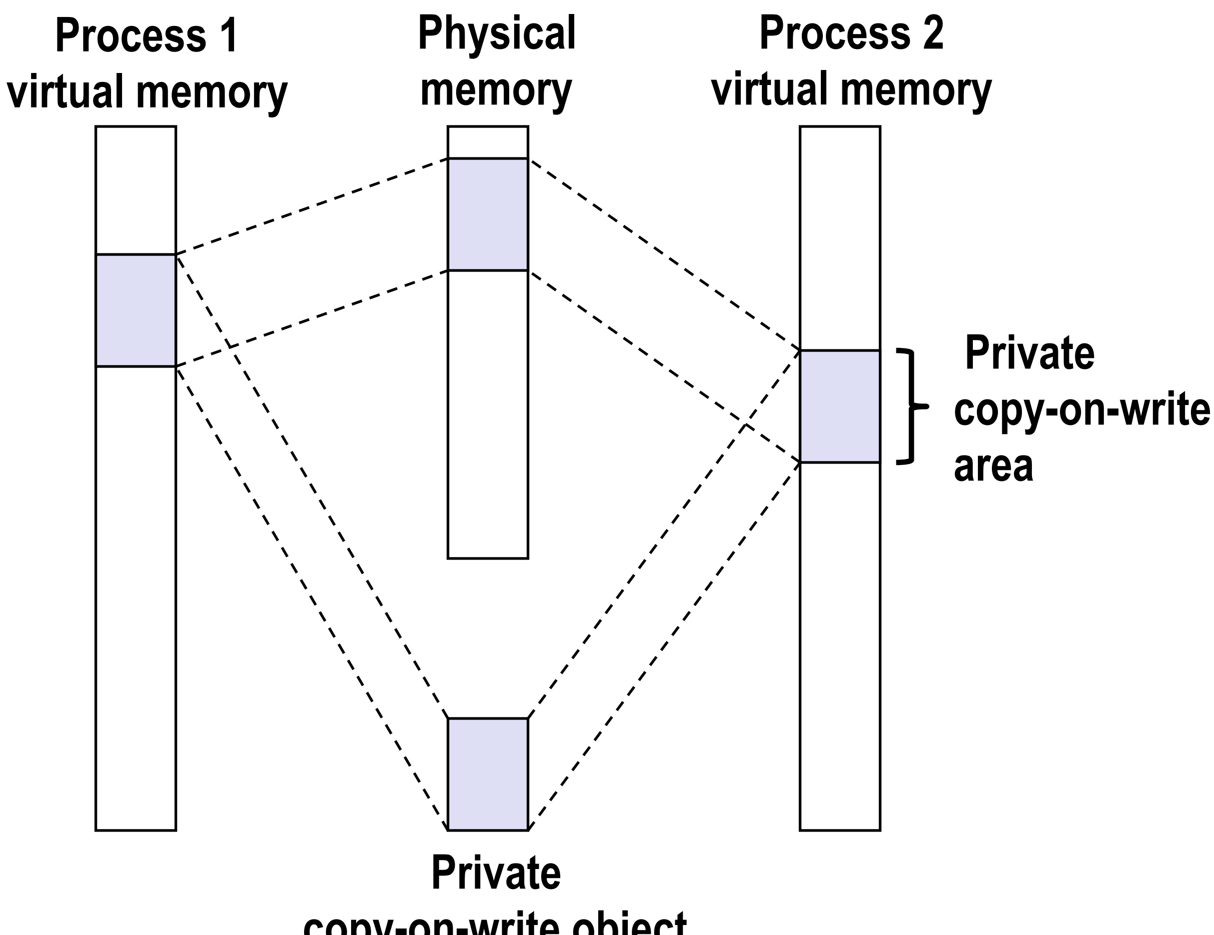
- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.
- Idea: Copy-on-write (COW)
    - First pretend that both processes will share the objects without modifying them. If modification happens, create separate copies.

Private Copy-on-write (COW) Objects



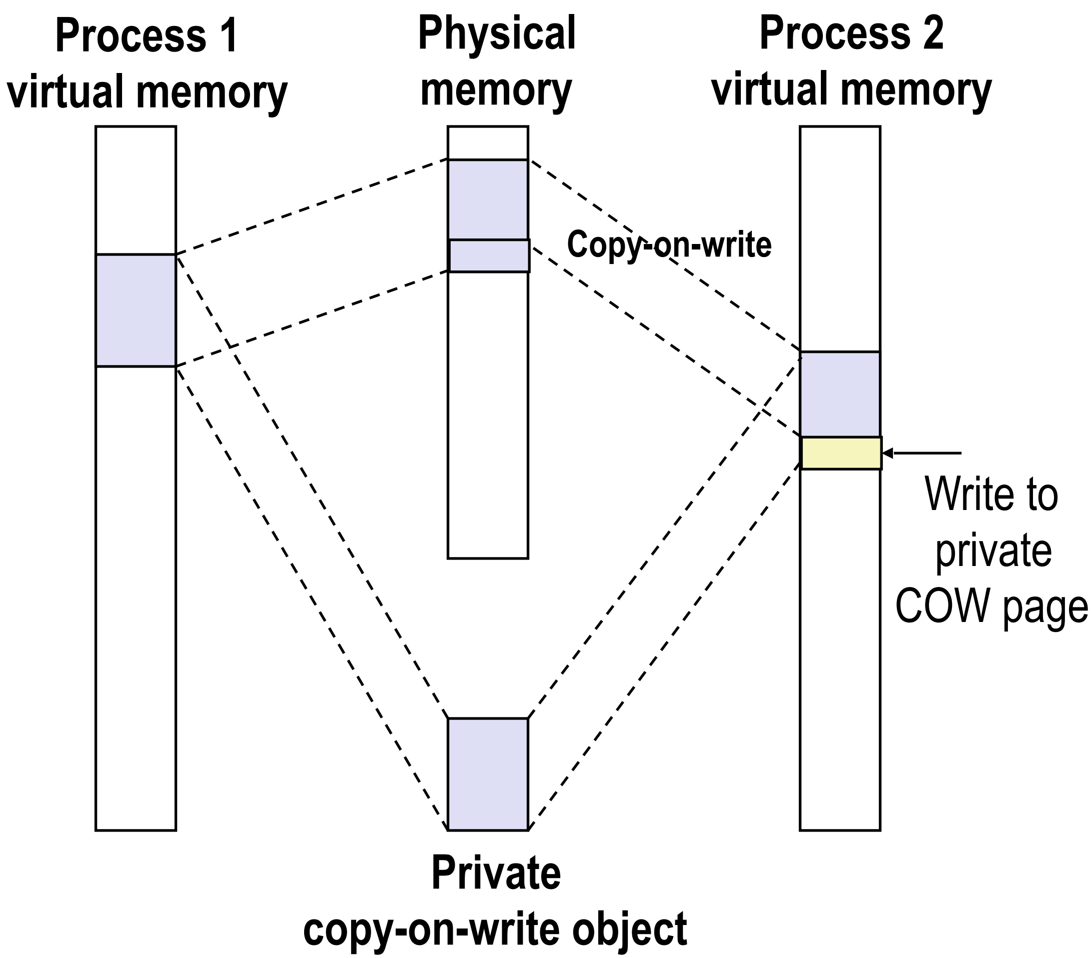
- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write (COW)
- PTEs in private areas are flagged as read-only

Private Copy-on-write (COW) Objects



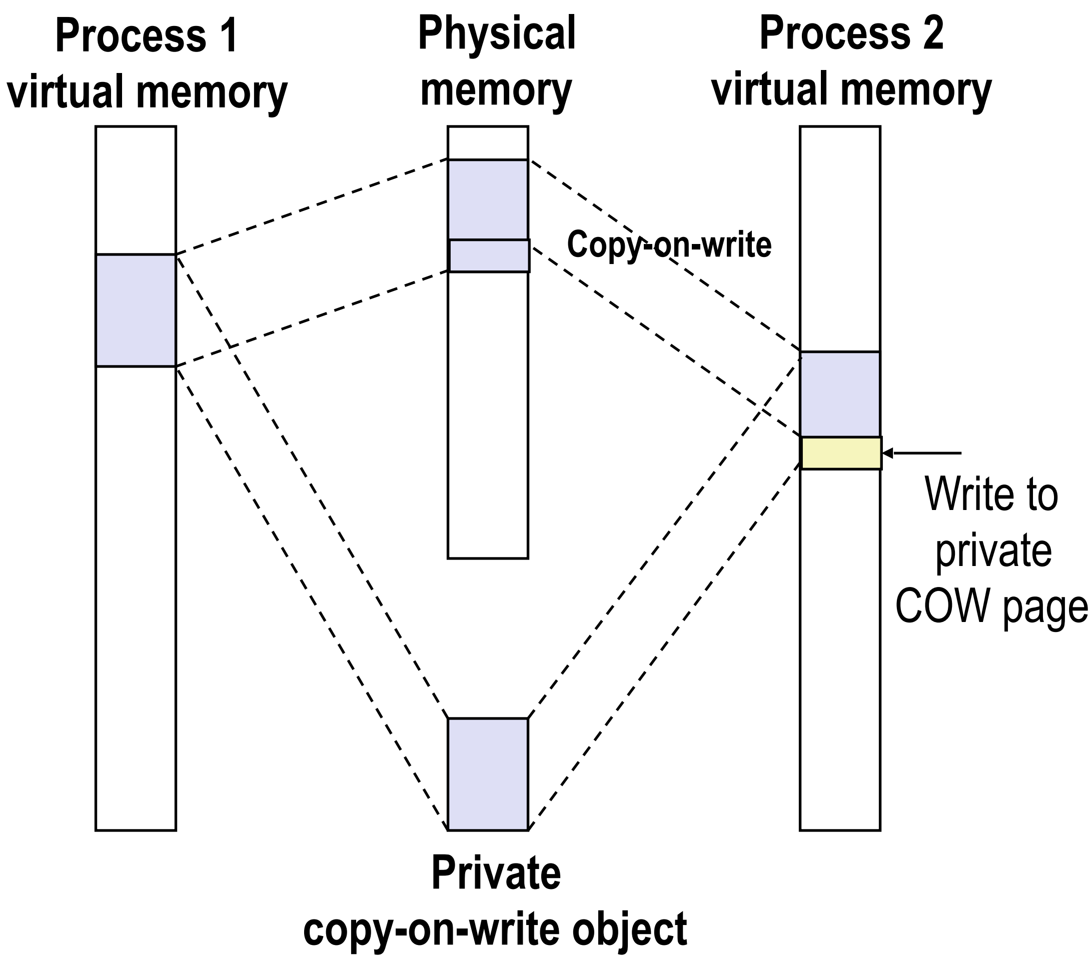
- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write (COW)
- PTEs in private areas are flagged as read-only

Private Copy-on-write (COW) Objects



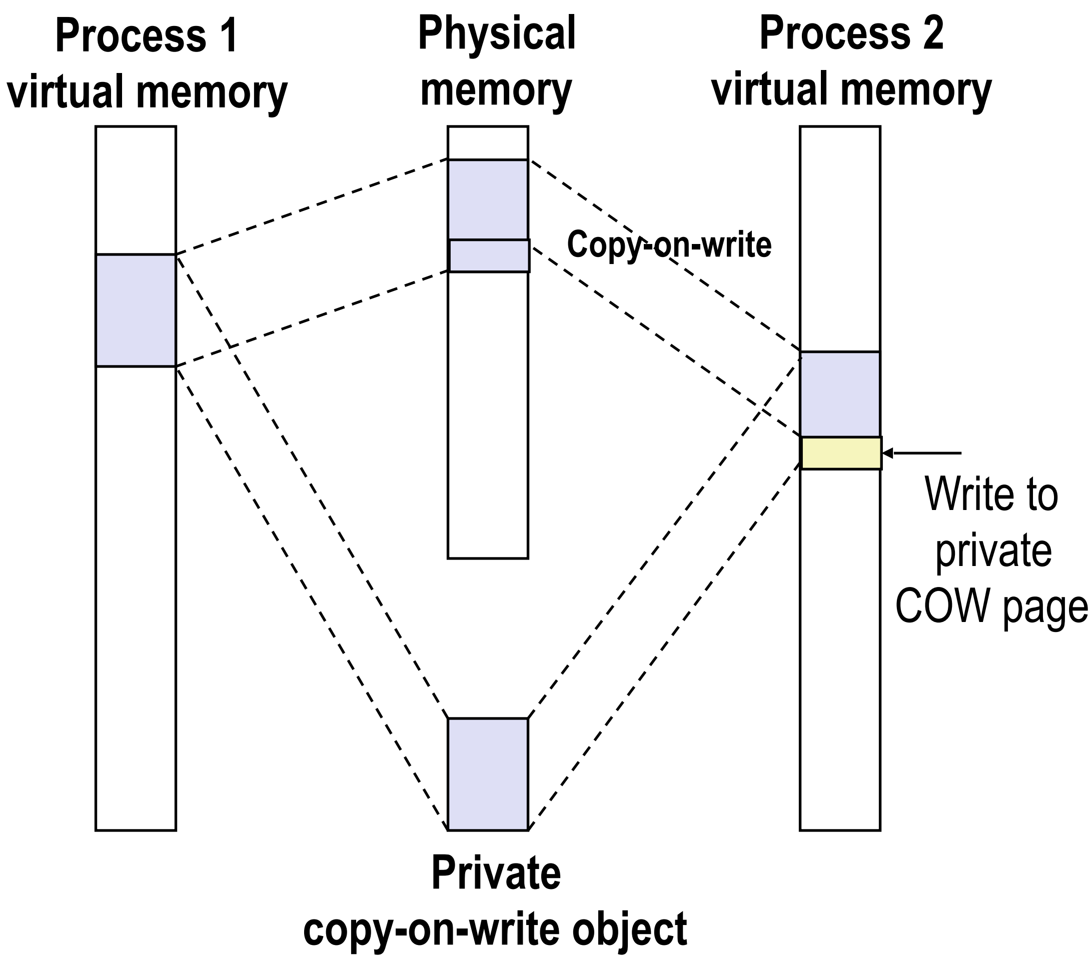
- Instruction writing to private page triggers page (protection) fault.

Private Copy-on-write (COW) Objects



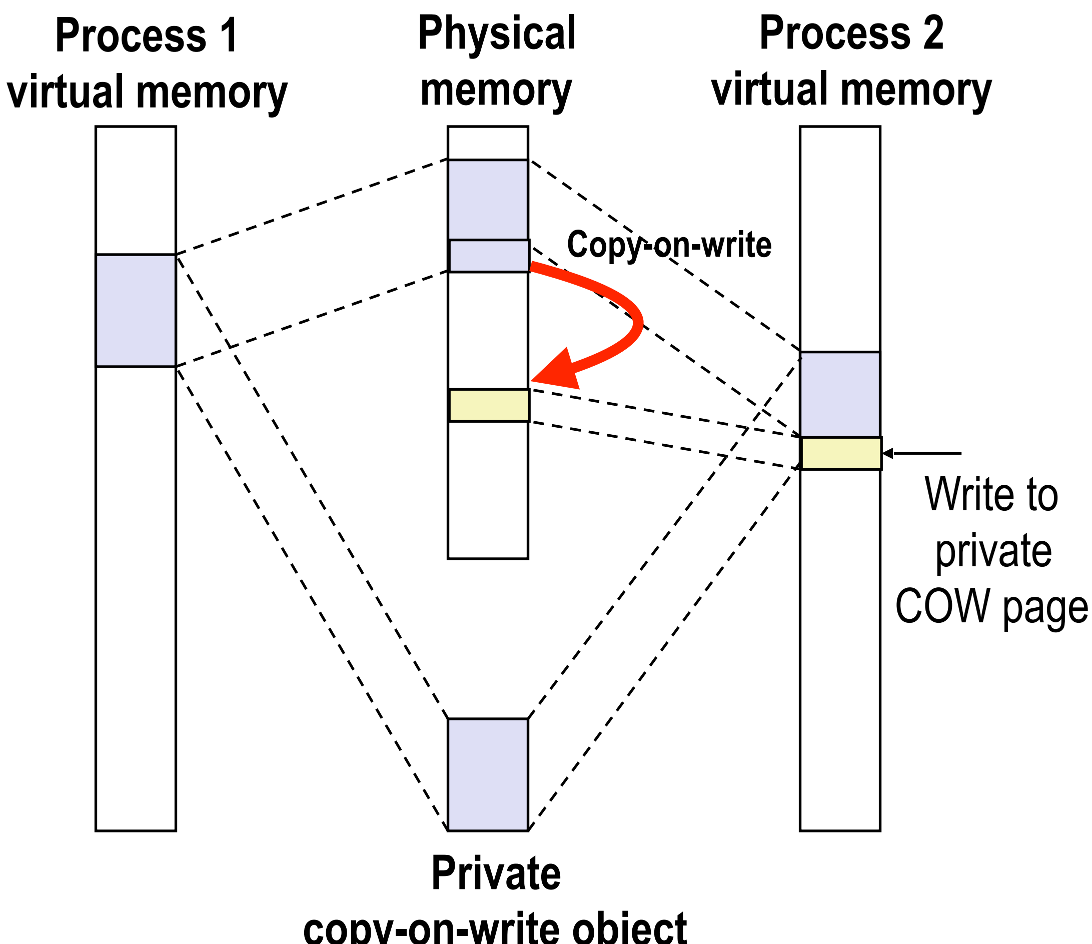
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object

Private Copy-on-write (COW) Objects



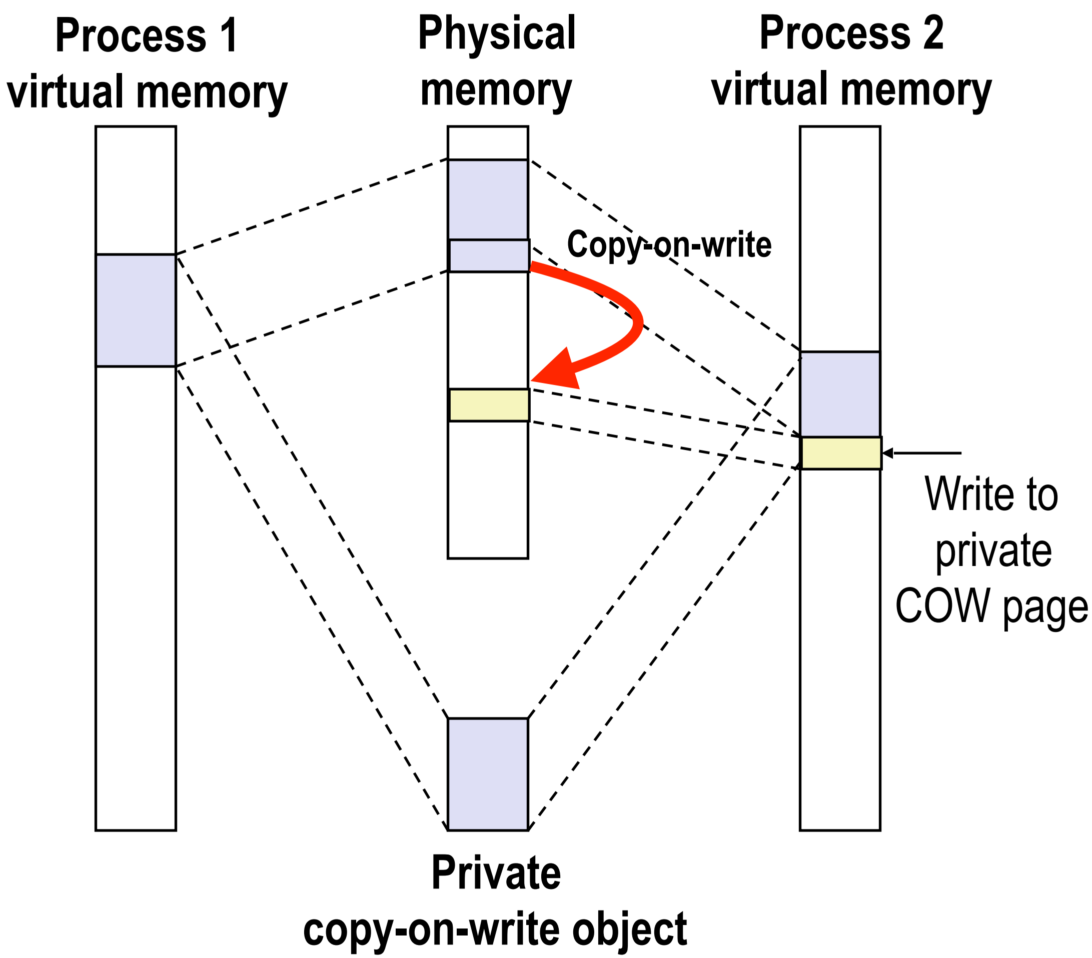
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.

Private Copy-on-write (COW) Objects



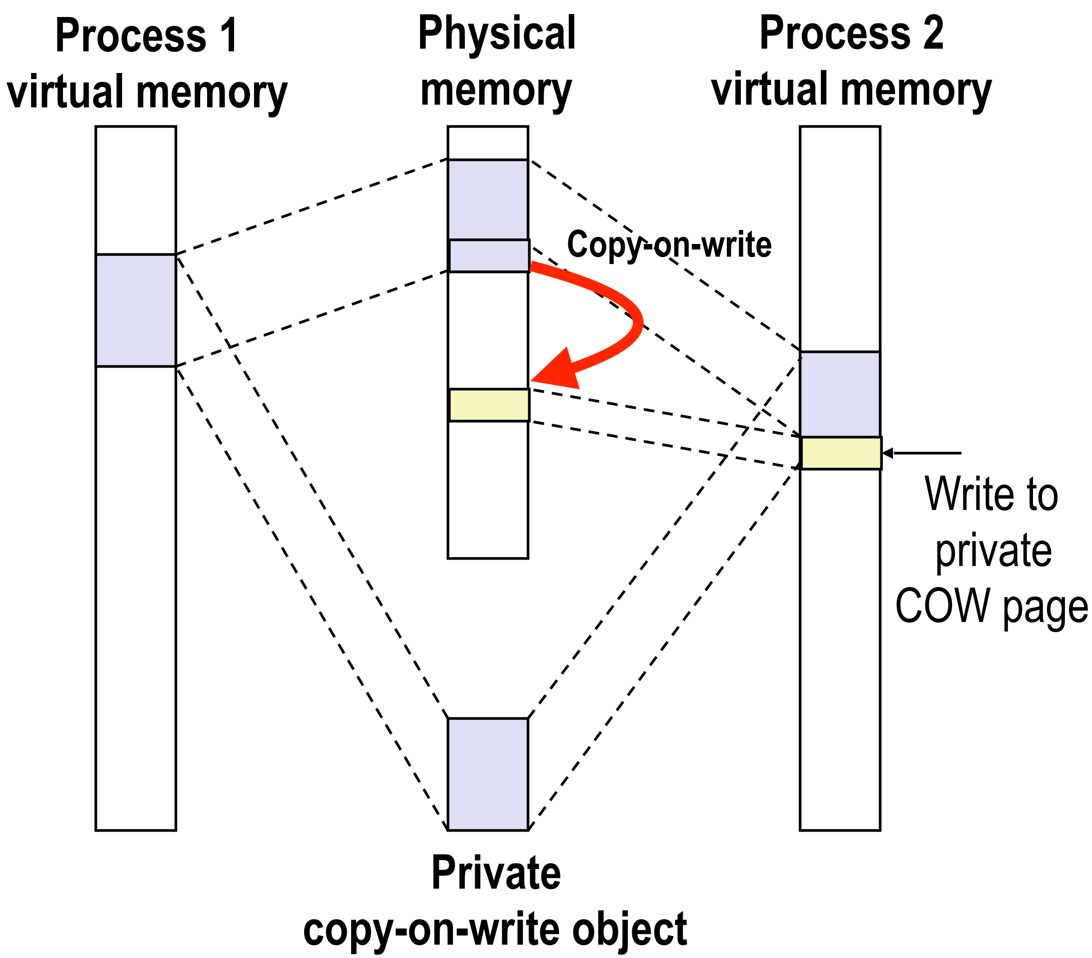
- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it's a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

User-Level Memory Mapping

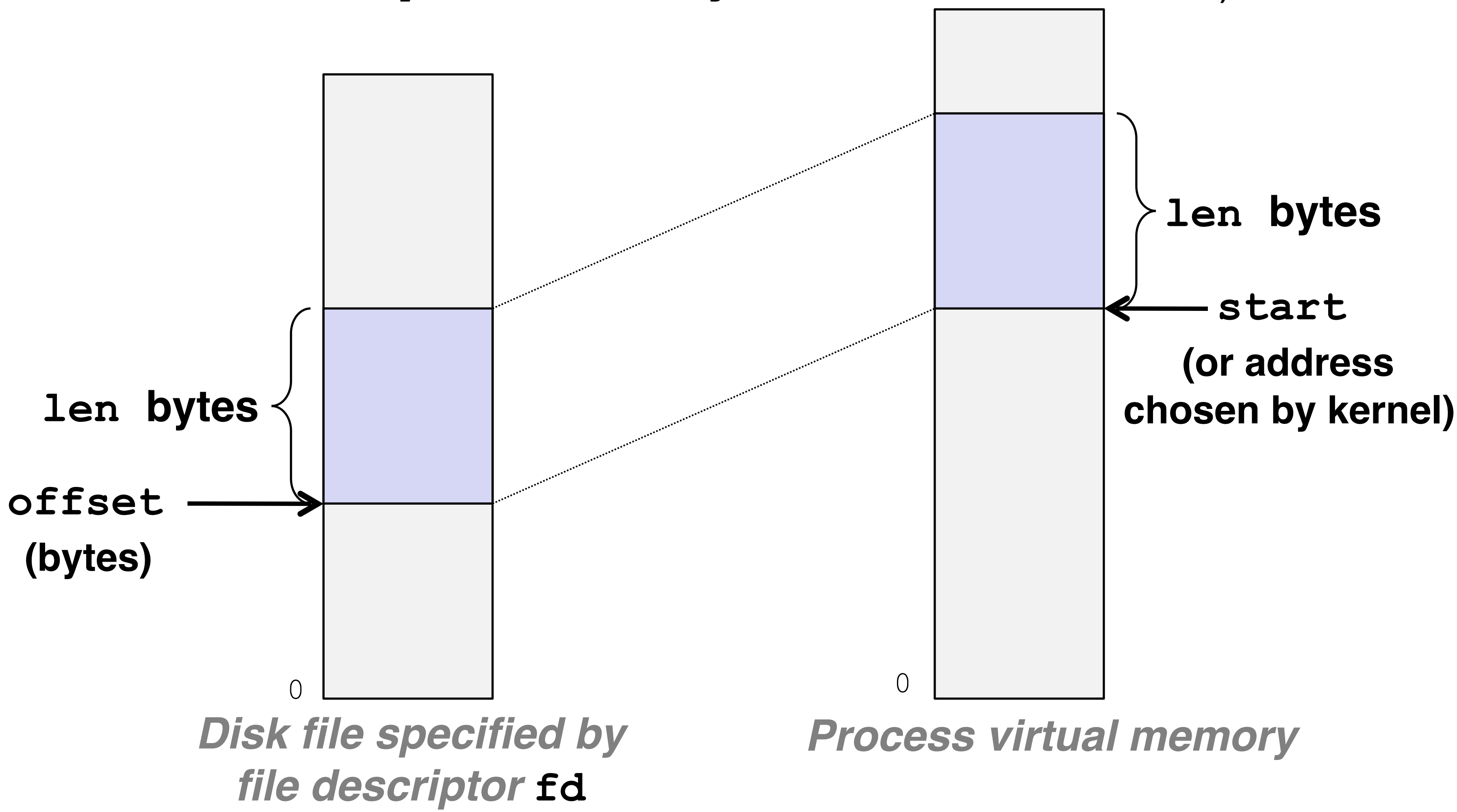
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
    - `start:` may be NULL for "pick an address"
    - `prot`: PROT_READ, PROT_WRITE, ...
    - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be `start`)

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Example: Using `mmap` to Copy Files

- Copying a file to stdout without transferring data to user space
  - i.e., no file data is copied to user stack

```c
#include "csapp.h"

void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size,
                PROT_READ,
                MAP_PRIVATE,
                fd, 0);
    Write(1, bufp, size);
    return;
}
```

mmapcopy.c

```c
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
               argv[0]);
        exit(0);
    }

    /* Copy input file to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

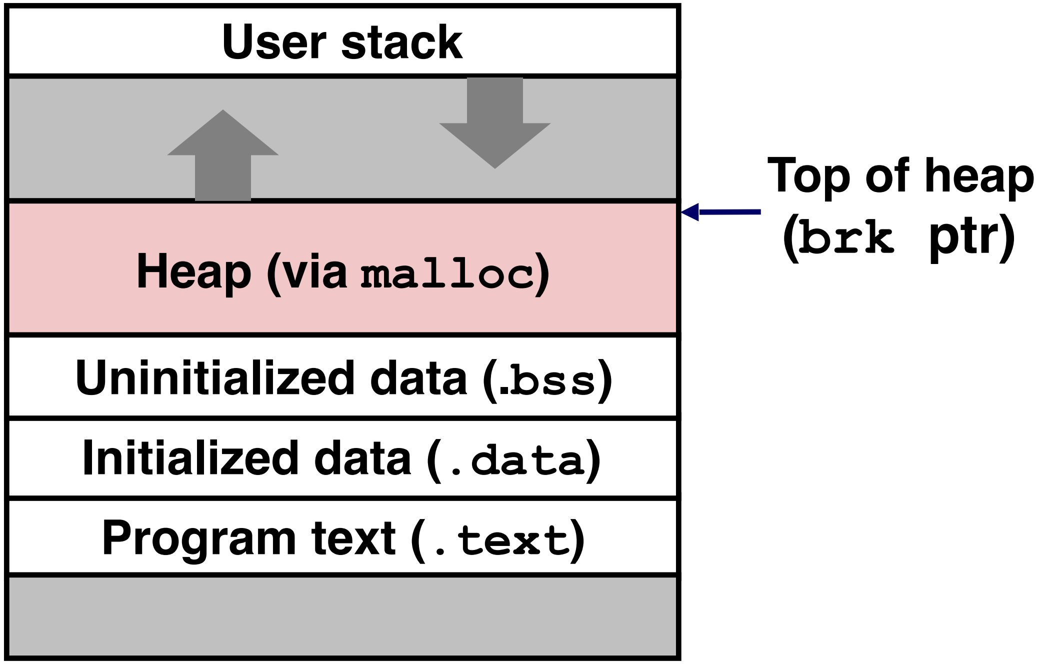
mmapcopy.c

Today

- Case study: Core i7/Linux memory system
- Memory mapping
- Dynamic memory allocation
    - Basic concepts
    - Implicit free lists

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



The `malloc/free` Functions

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

The `malloc/free` Functions

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- Successful:
    - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`

The `malloc/free` Functions

```
#include <stdlib.h>
```

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

`void free(void *p)`

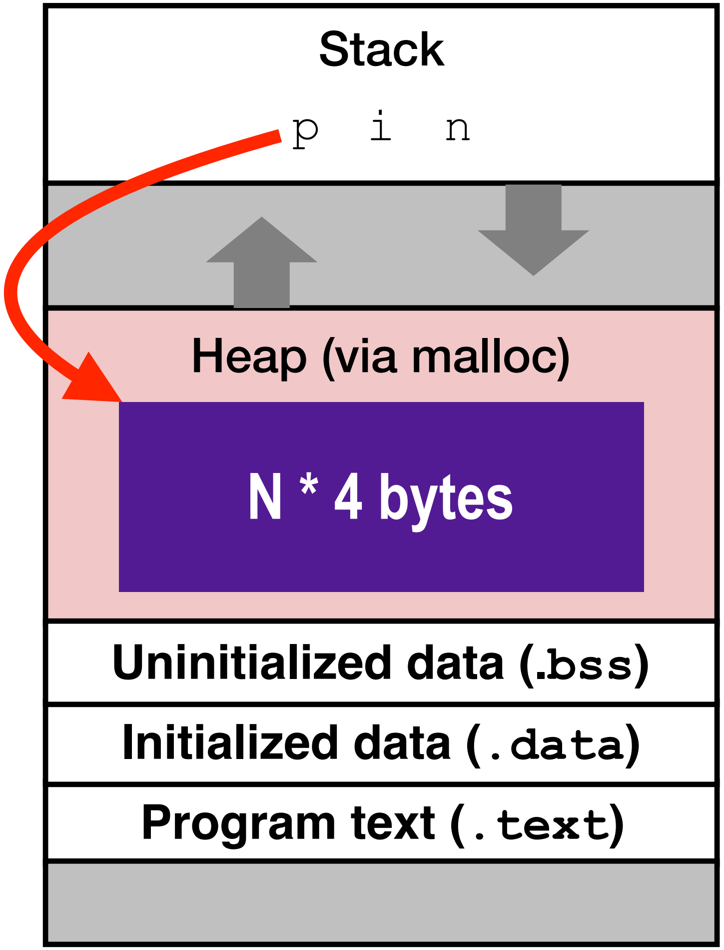
- Returns the block pointed at by **`p`** to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`

Other functions

- `calloc`: Version of `malloc` that initializes allocated block to zero.
- `realloc:` Changes the size of a previously allocated block.
- `sbrk`: Used internally by allocators to grow or shrink the heap

malloc Example

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

Why Do We Need Dynamic Allocation?

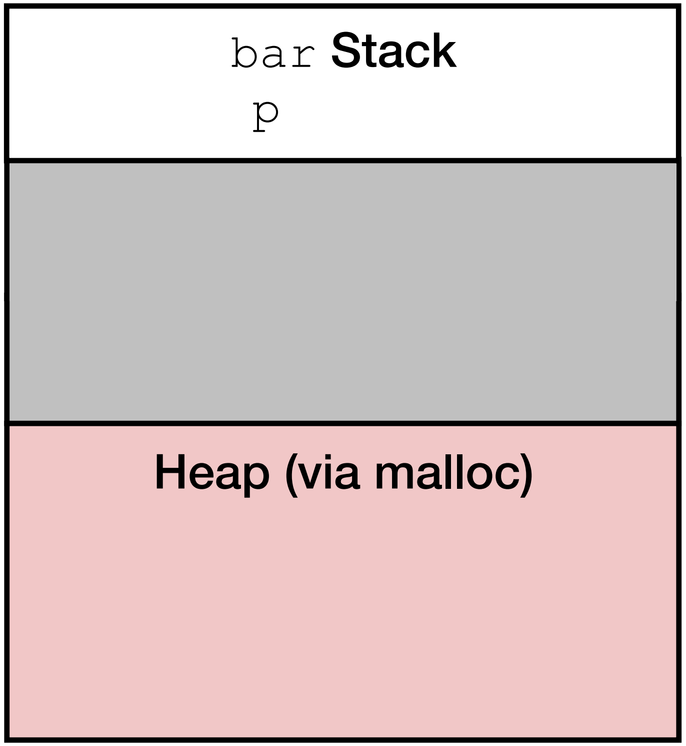
- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```

Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

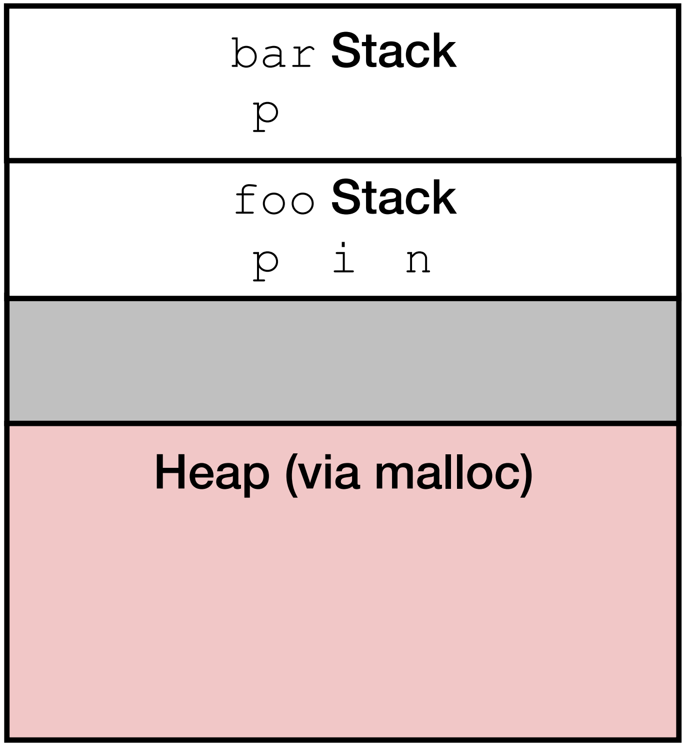
```c
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

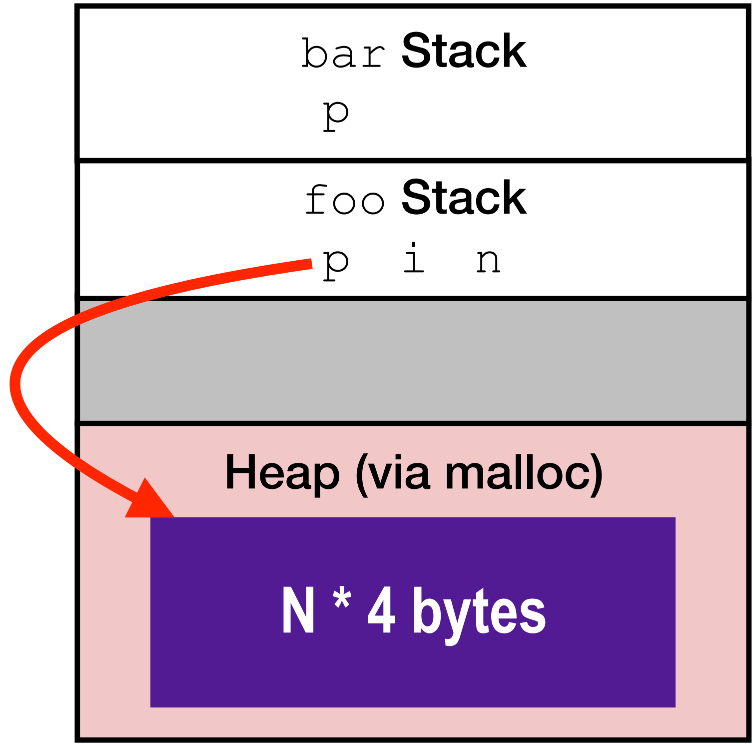
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf(“%d\n”, p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

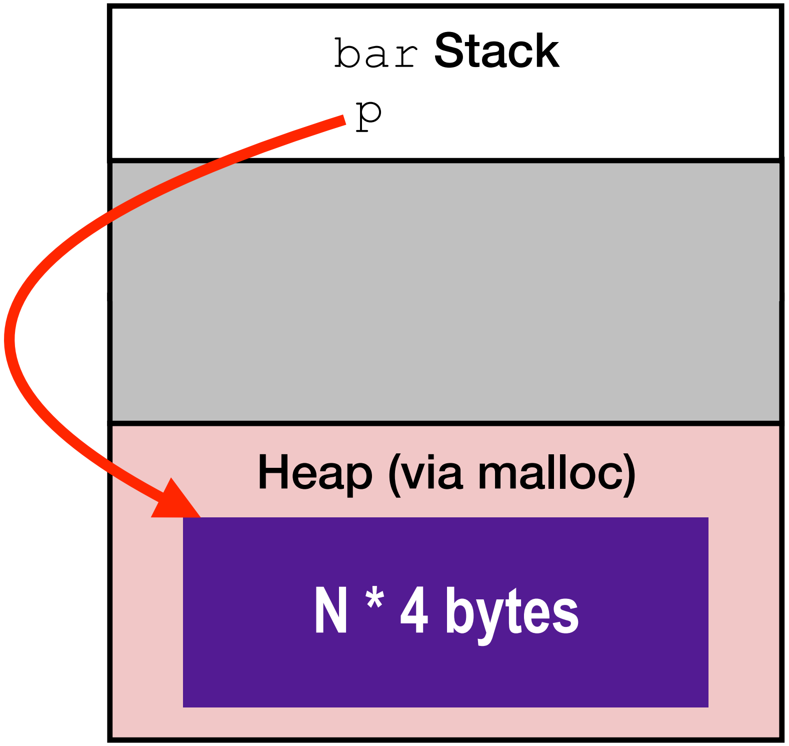
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

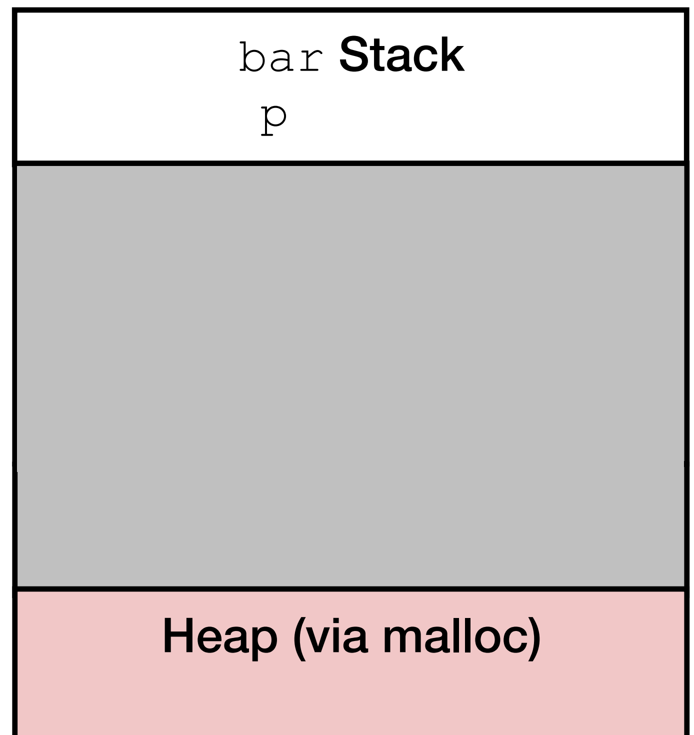
```
int* foo(int n) {
  int i, *p;

  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) exit(0);

  for (i=0; i<n; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo(5);

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

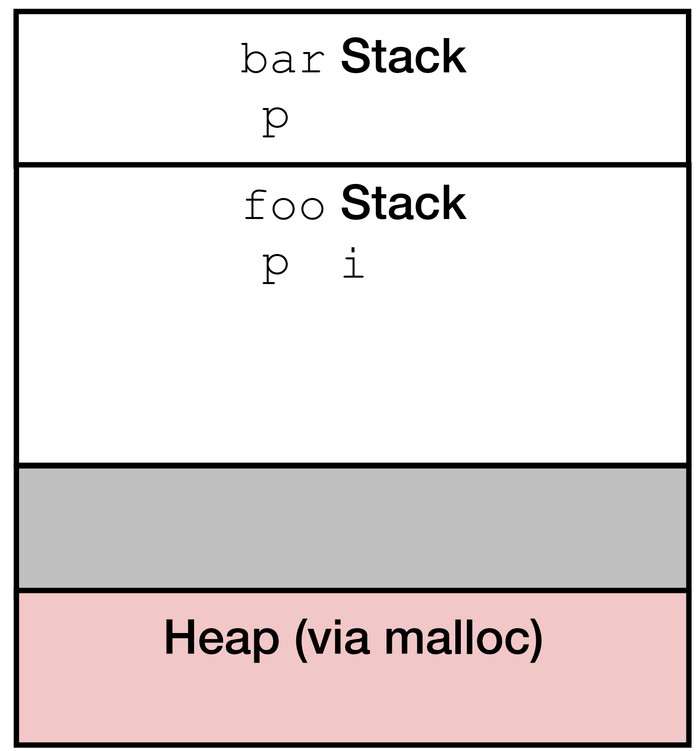
```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

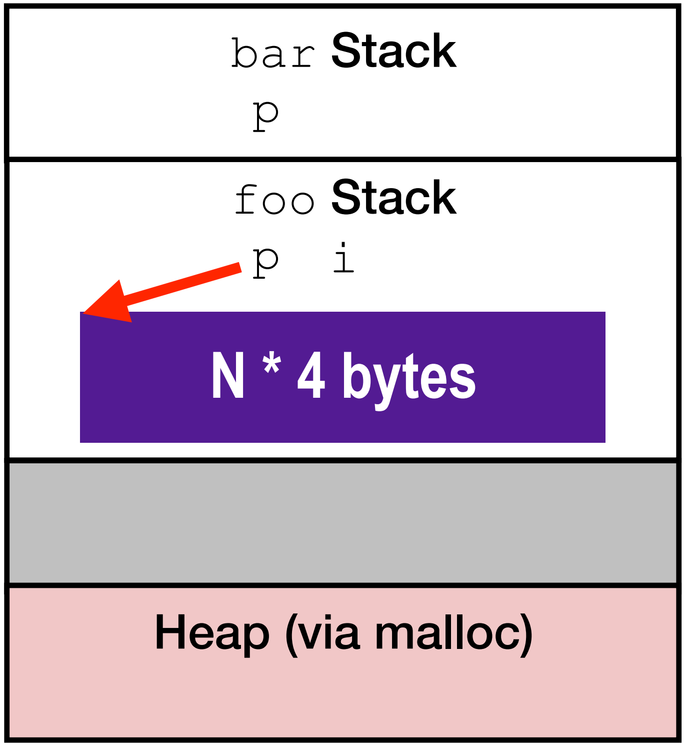
```c
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

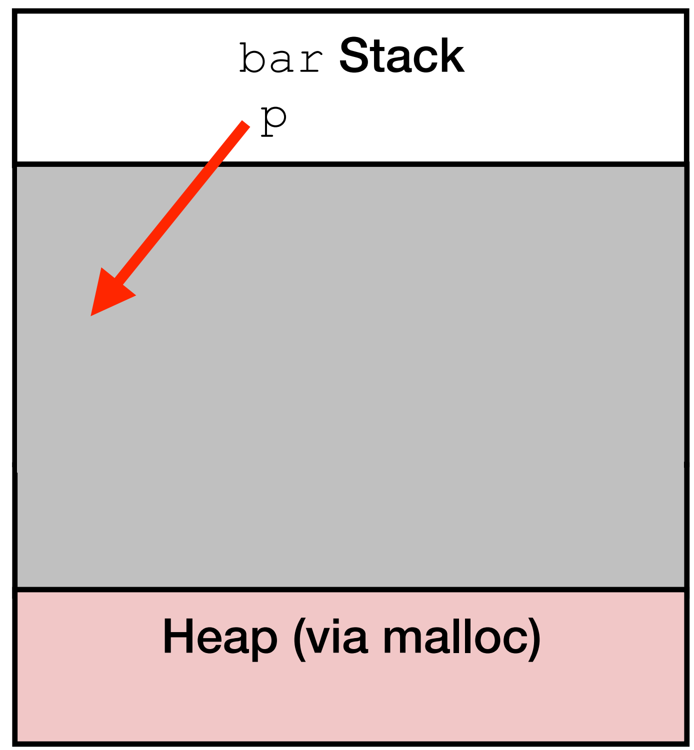
```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Why Do We Need Dynamic Allocation?

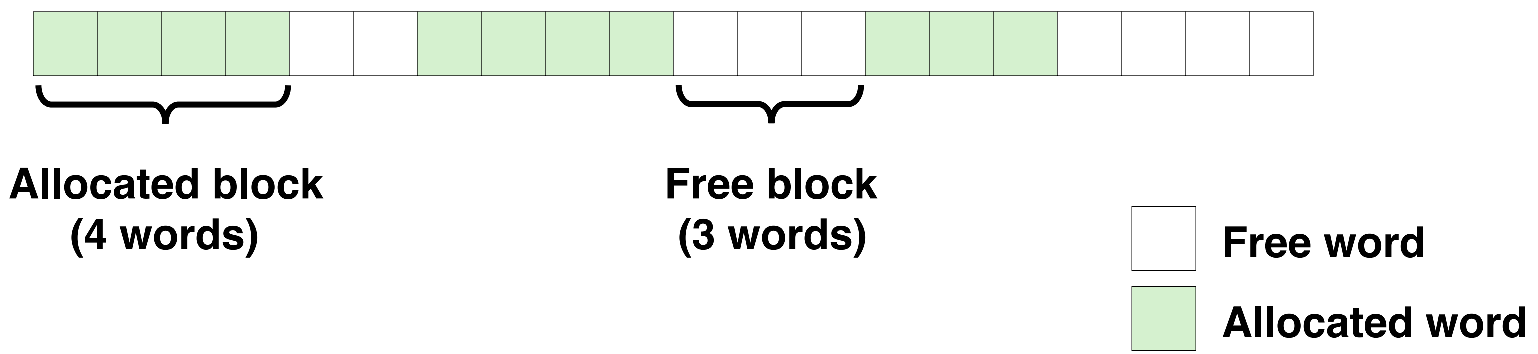
- Some data structures' size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```
int* foo() {
  int i;
  int p[5];

  for (i=0; i<5; i++)
    p[i] = i;

  return p;
}

void bar() {
  int *p = foo();

  printf("%d\n", p[0]);
}
```



Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Blocks that are no longer used should be free-ed to save space



- Assumptions Made in This Lecture
  - Memory is word addressed
  - Words are int-sized

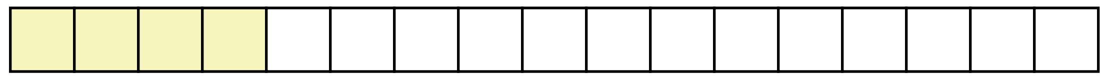
Dynamic Memory Allocation

- Types of allocators
  - *Explicit allocator*: application (i.e., programmer) allocates and frees space
    - E.g., `malloc` and `free` in C
  - *Implicit allocator:* application allocates, but does not free space
    - E.g. garbage collection in Java, JavaScript, Python, etc…

- Will discuss simple explicit memory allocation today

Allocation Example

`p1 = malloc(4)`



`p2 = malloc(5)`

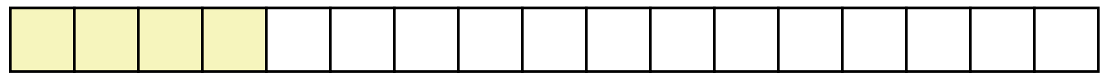
`p3 = malloc(6)`

`free(p2)`

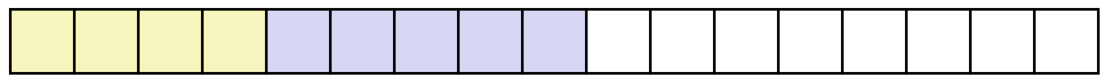
`p4 = malloc(2)`

Allocation Example

`p1 = malloc(4)`



`p2 = malloc(5)`



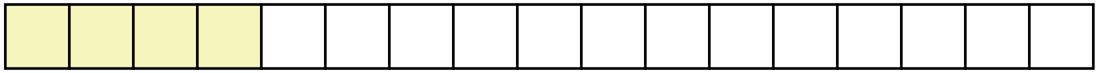
`p3 = malloc(6)`

`free(p2)`

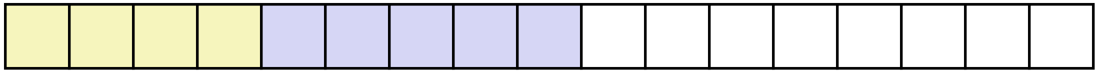
`p4 = malloc(2)`

Allocation Example

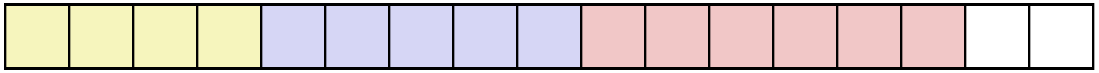
`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`

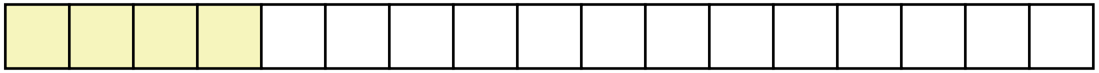


`free(p2)`

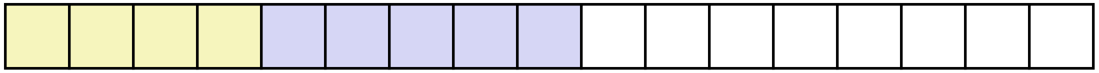
`p4 = malloc(2)`

Allocation Example

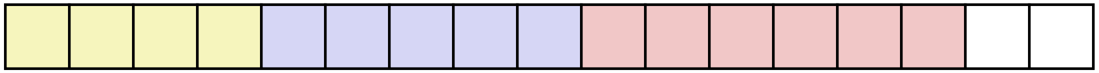
`p1 = malloc(4)`



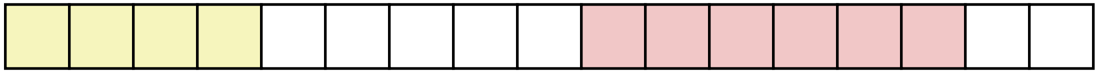
`p2 = malloc(5)`



`p3 = malloc(6)`



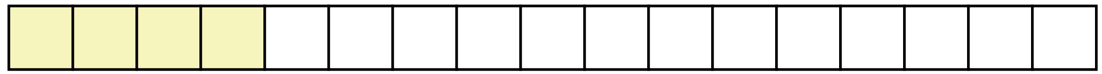
`free(p2)`



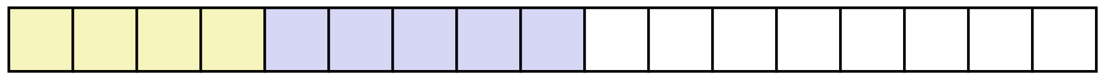
`p4 = malloc(2)`

Allocation Example

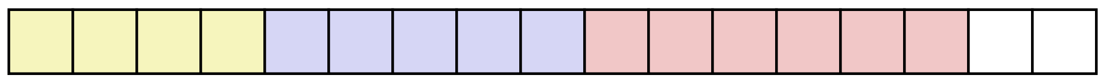
`p1 = malloc(4)`



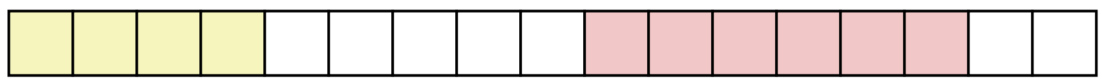
`p2 = malloc(5)`



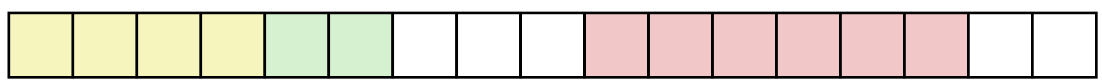
`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



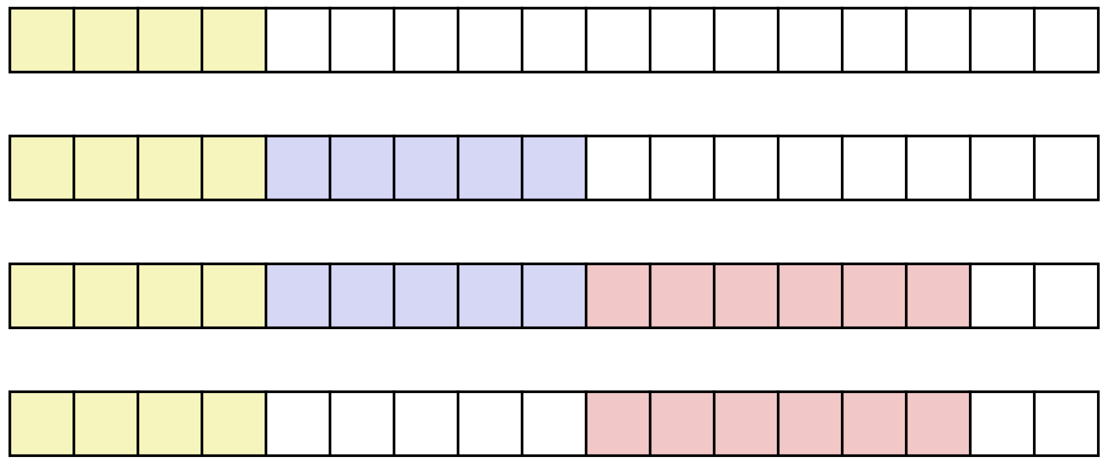
Constraints

- Applications
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - `free` request must be to a `malloc`'d block
- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can place allocated blocks only in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)
```



External Fragmentation

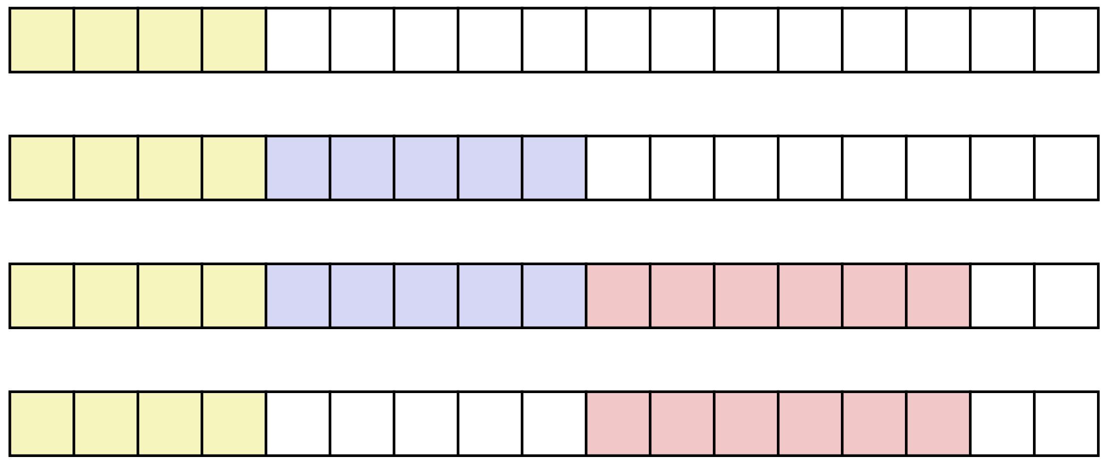
- Occurs when there is enough aggregate heap memory, but no single free block is large enough

`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`



`p4 = malloc(6)`

Oops! (what would happen now?)

External Fragmentation

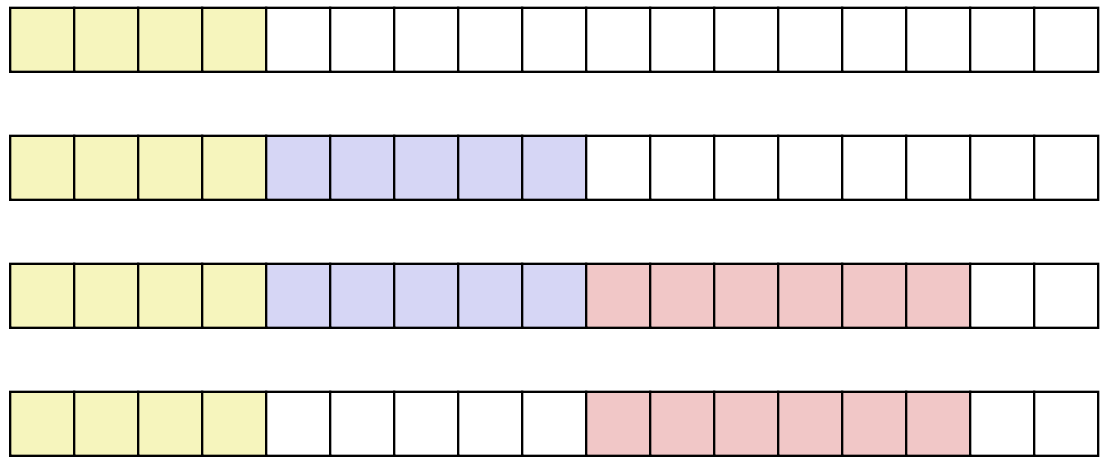
- Occurs when there is enough aggregate heap memory, but no single free block is large enough

`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`



`p4 = malloc(6)`

Oops! (what would happen now?)

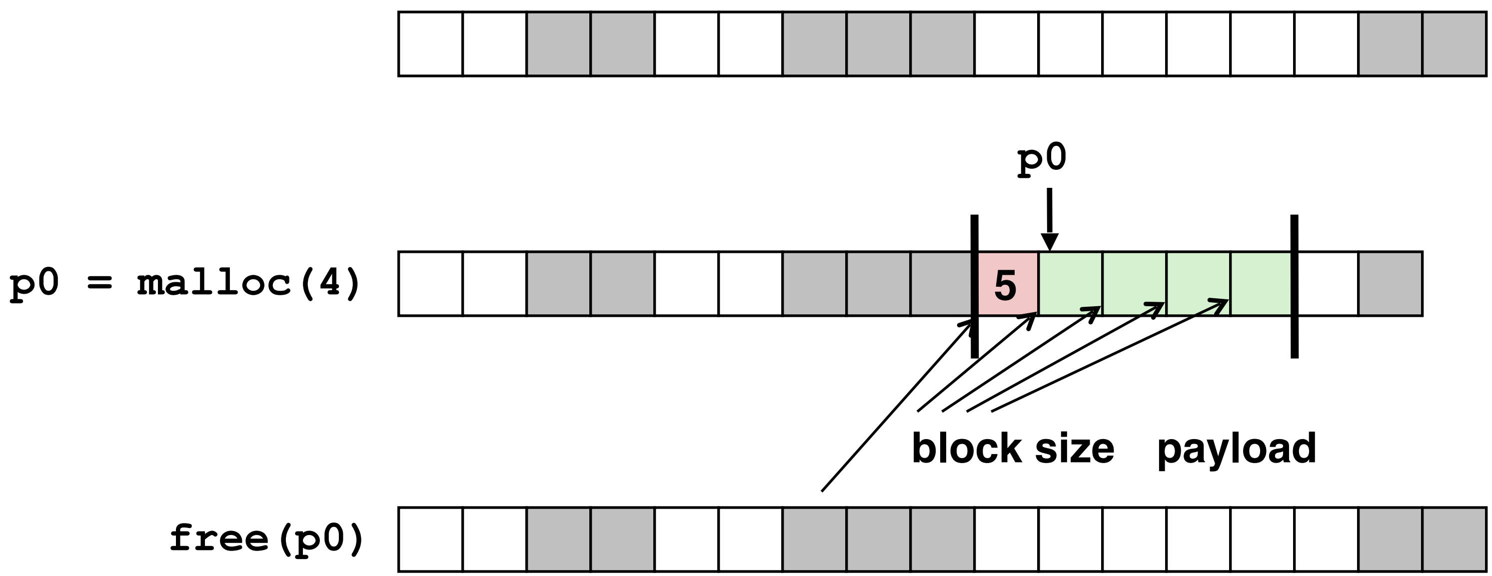
- Depends on the pattern of future requests

Key Issues in Dynamic Memory Allocation

- Free:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we reinsert freed block?
- Allocation:
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
  - How do we pick a block to use for allocation -- many might fit?

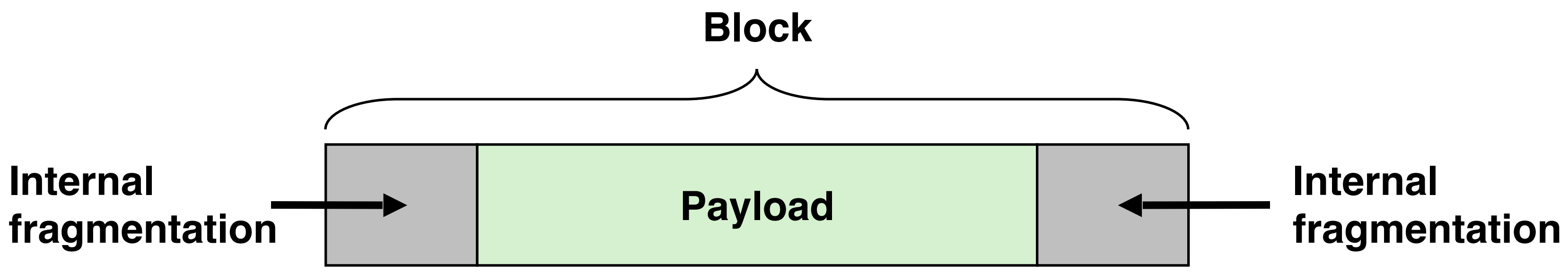
Knowing How Much to Free

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



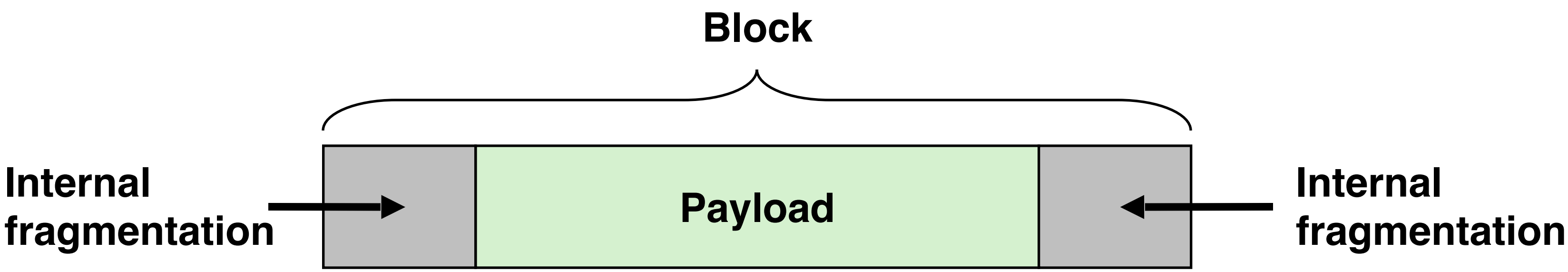
Internal Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than block size



Internal Fragmentation

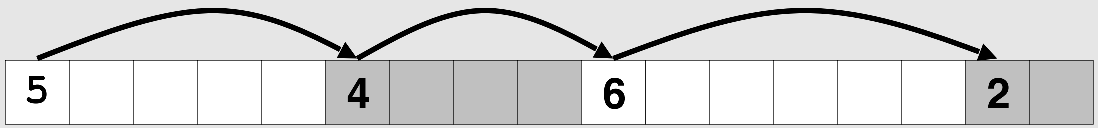
- For a given block, *internal fragmentation* occurs if payload is smaller than block size



- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions (e.g., to return a big block to satisfy a small request)

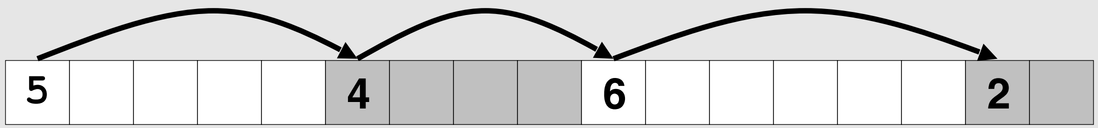
Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

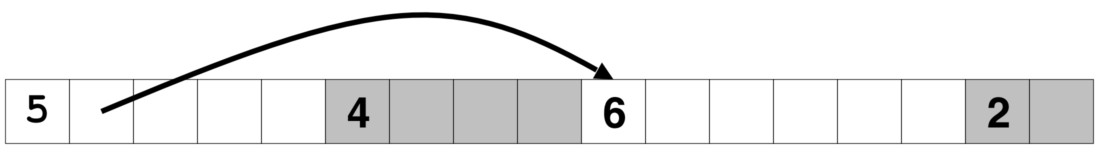


Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

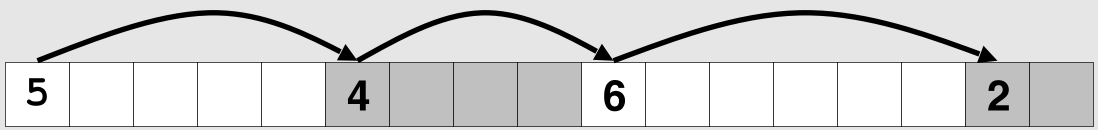


- Method 2: *Explicit list* among the free blocks using pointers

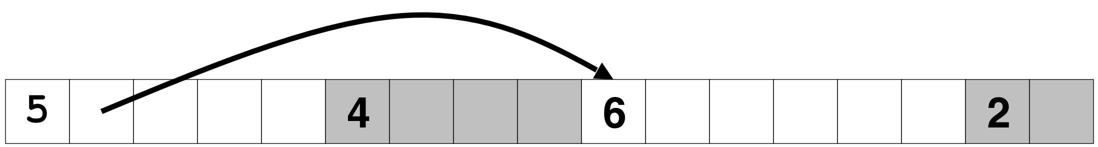


Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



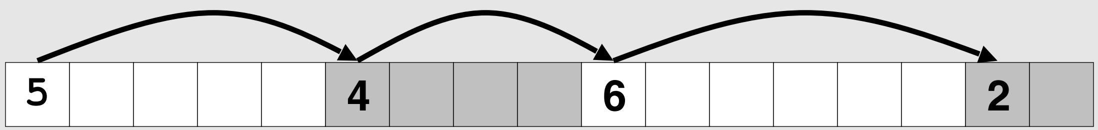
- Method 2: *Explicit list* among the free blocks using pointers



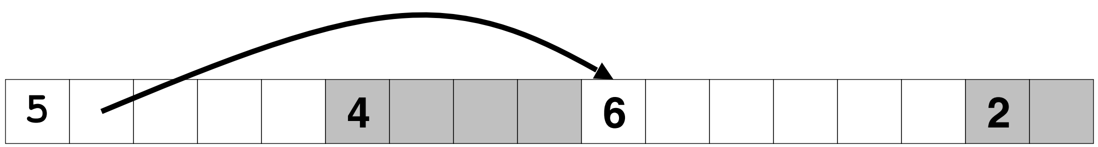
- Method 3: *Segregated free list*
  - Different free lists for different size classes

Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes

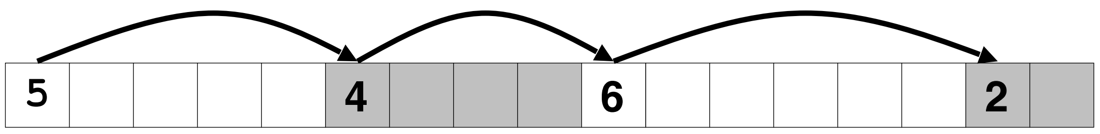
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Today

- Case study: Core i7/Linux memory system
- Memory mapping
- Dynamic memory allocation
  - Basic concepts
  - Implicit free lists

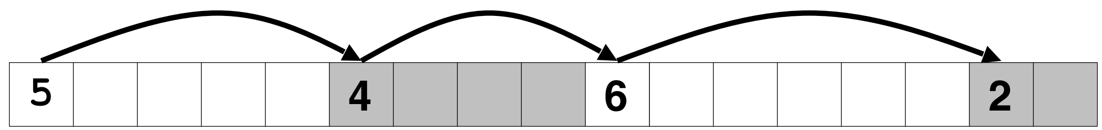
Implicit List

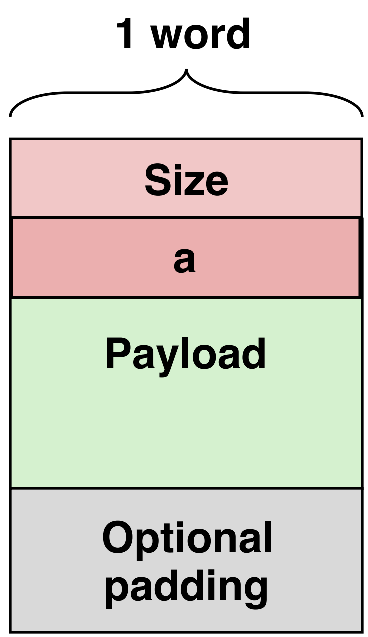
- For each block we need both size and allocation status
    - Could store this information in two words: wasteful!



Implicit List

- For each block we need both size and allocation status
    - Could store this information in two words: wasteful!





a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

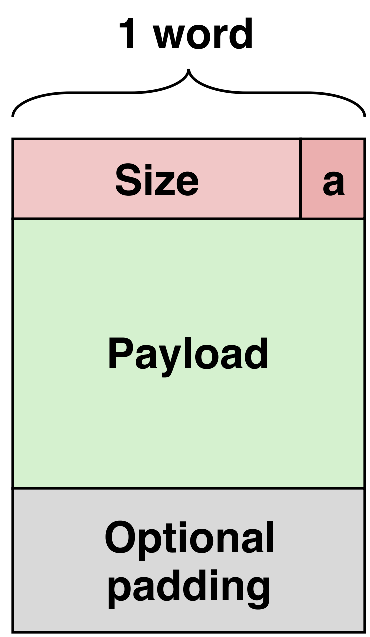
Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

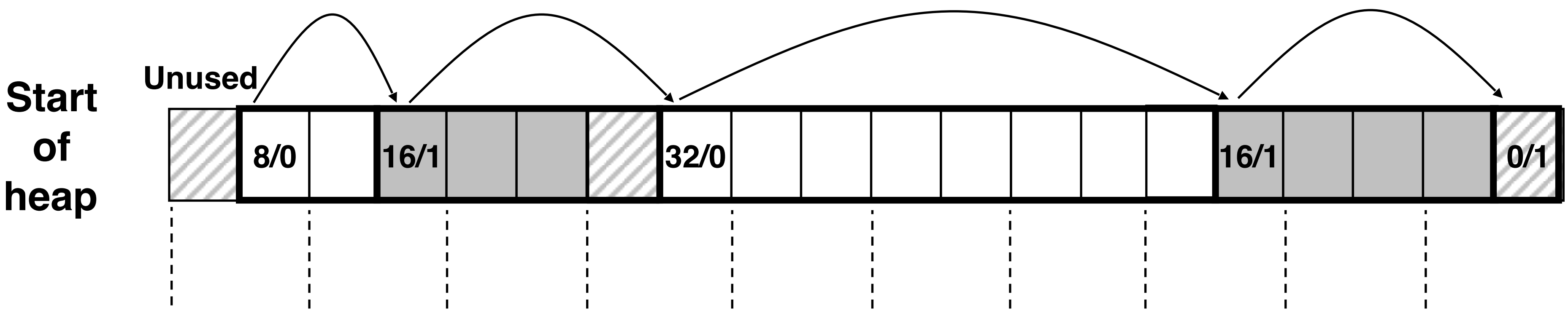


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list

Finding a Free Block

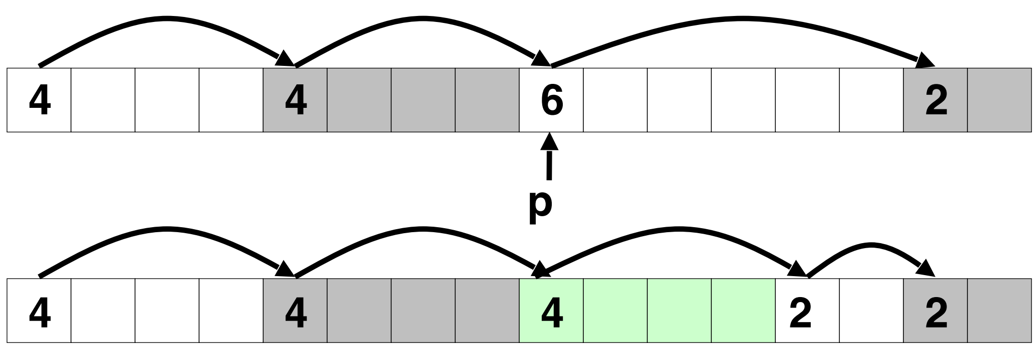
- First fit:
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse

Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit

Allocating in Free Block

- Allocated space might be smaller than free space
- We could simply leave the extra space there. Simple to implement but causes internal fragmentation
- Or we could split the block



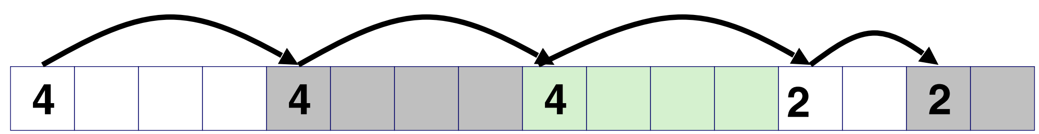

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to “false fragmentation”

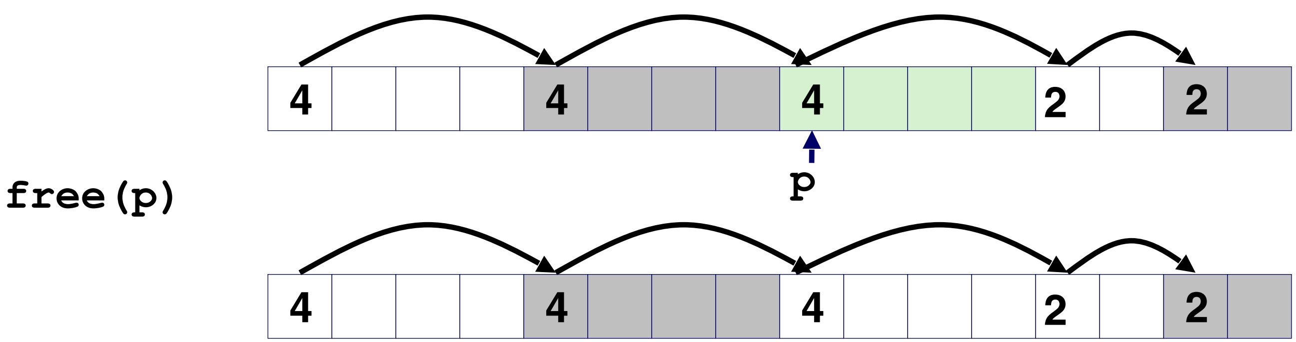


Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to “false fragmentation”

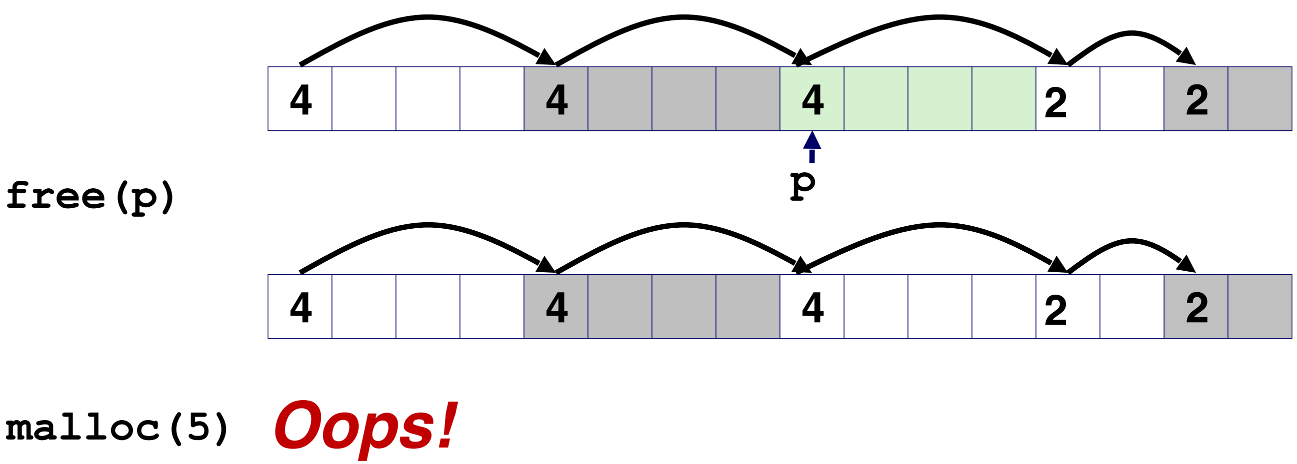


Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to “false fragmentation”

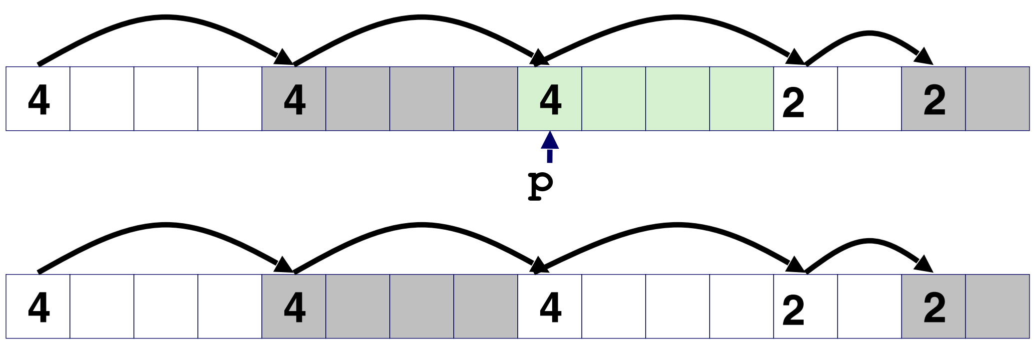


Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to “false fragmentation”



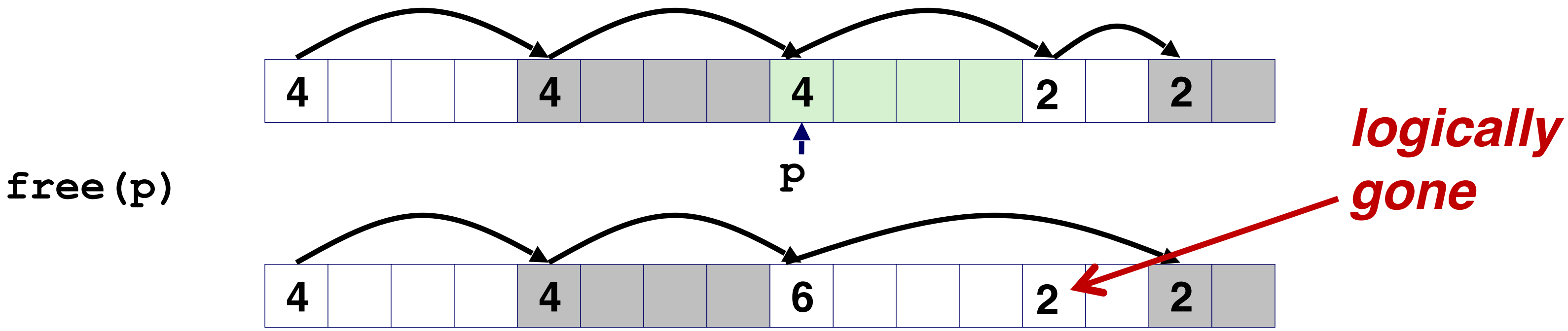
`free(p)`

`malloc(5)` ***Oops!***

There is enough free space, but the allocator won’t be able to find it

Coalescing

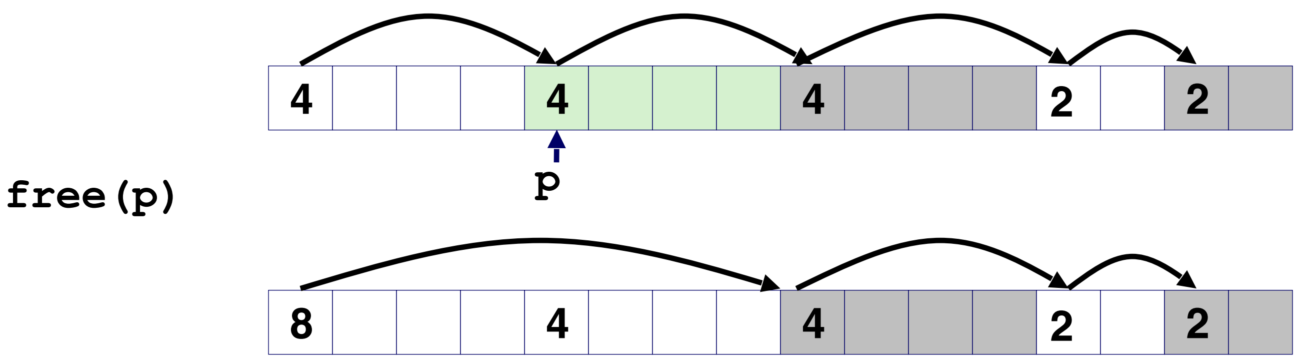
- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block




```
void free_block(ptr p) {
    *p = *p & -2;              // clear allocated flag
    next = p + *p;             // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;         // add to this block if
}                              //    not allocated
```

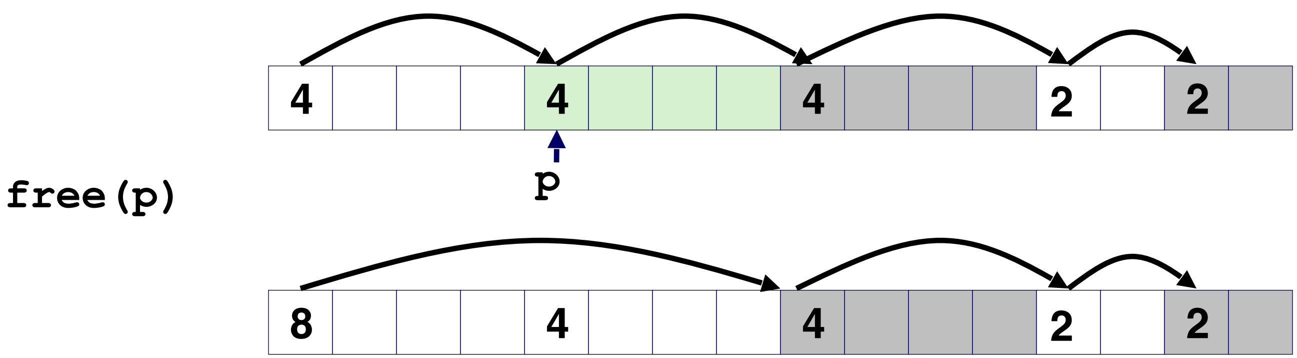
Coalescing

- How about now?



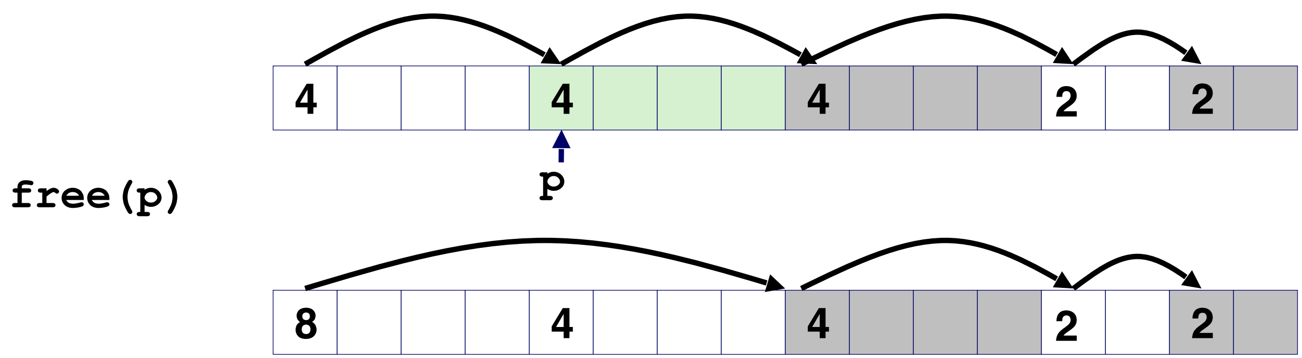
Coalescing

- How about now?
- How do we coalesce with previous block?



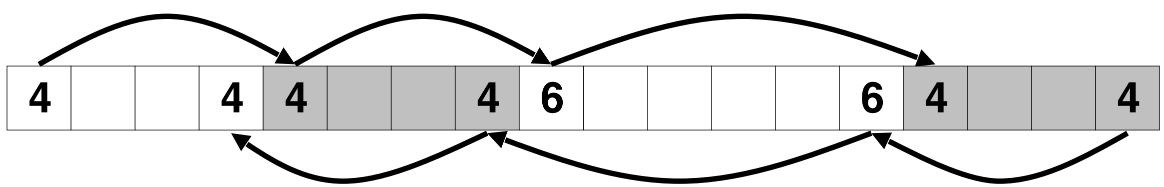
Coalescing

- How about now?
- How do we coalesce with previous block?
  - Linear time solution: scans from beginning



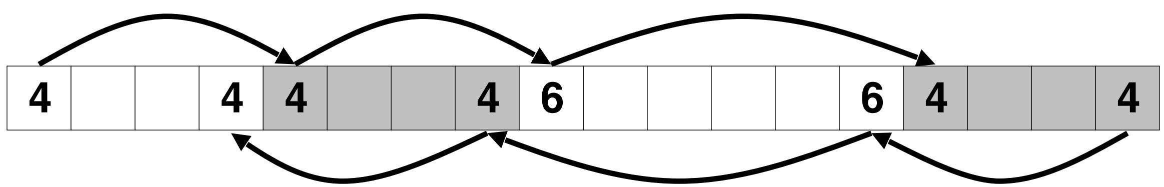
Bidirectional Coalescing (Constant Time)

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!

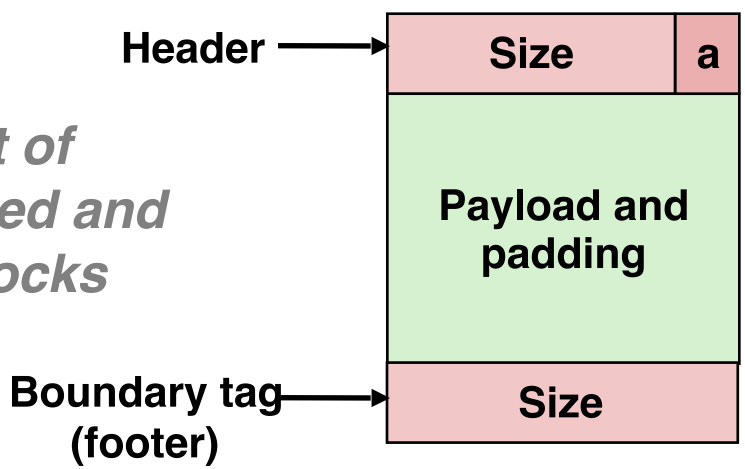


Bidirectional Coalescing (Constant Time)

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at “bottom” (end) of free blocks
  - Allows us to traverse the “list” backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks



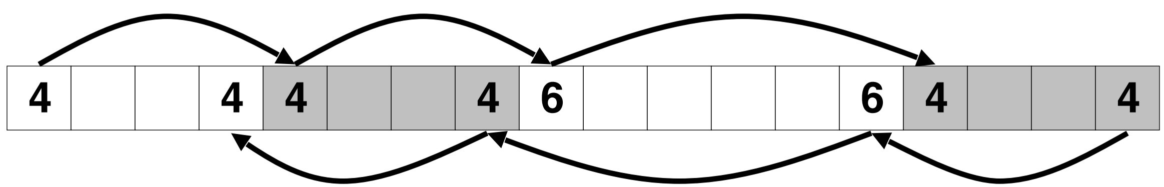
a = 1: Allocated block
a = 0: Free block

Size: Total block size

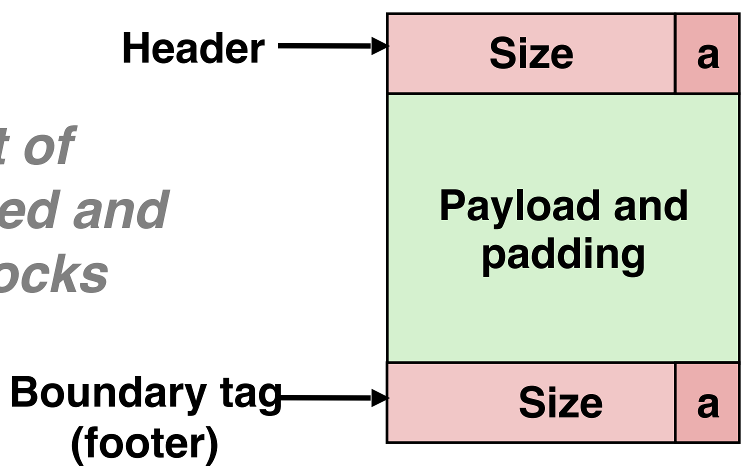
Payload: Application data (allocated blocks only)

Bidirectional Coalescing (Constant Time)

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at “bottom” (end) of free blocks
  - Allows us to traverse the “list” backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks



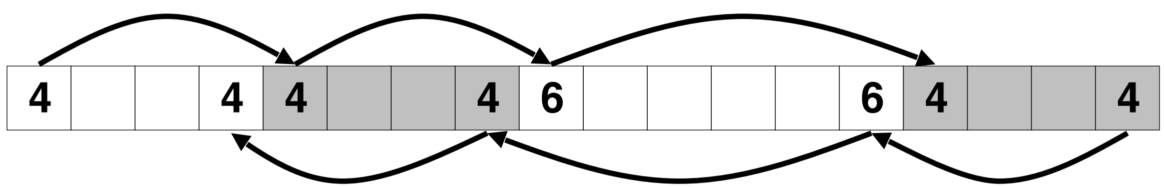
a = 1: Allocated block
a = 0: Free block

Size: Total block size

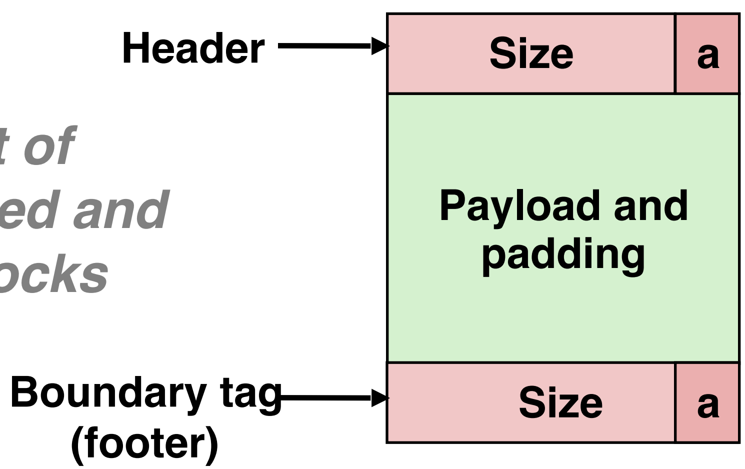
Payload: Application data (allocated blocks only)

Bidirectional Coalescing (Constant Time)

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at “bottom” (end) of free blocks
  - Allows us to traverse the “list” backwards, but requires extra space
  - Important and general technique!
- Disadvantages? (Think of small blocks…)



Format of allocated and free blocks



a = 1: Allocated block
a = 0: Free block

Size: Total block size

Payload: Application data (allocated blocks only)

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Splitting policy:
  - When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Splitting policy:
  - When do we split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- Coalescing policy:
  - Immediate coalescing: coalesce each time `free` is called
  - Deferred coalescing: try to improve performance of `free` by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for `malloc`
    - Coalesce when the amount of external fragmentation reaches some threshold

Implicit Lists: Summary

- Implementation: very simple

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case
- Memory usage:
  - Will depend on placement policy
  - First-fit, next-fit, or best-fit

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
  - Identify free blocks requires scanning *all* the blocks!
- Free cost:
  - constant time worst case
- Memory usage:
  - Will depend on placement policy
  - First-fit, next-fit, or best-fit
- Not used in practice because of linear-time allocation
  - used in many special purpose applications

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
    - linear time worst case
    - Identify free blocks requires scanning *all* the blocks!
- Free cost:
    - constant time worst case
- Memory usage:
    - Will depend on placement policy
    - First-fit, next-fit, or best-fit
- Not used in practice because of linear-time allocation
    - used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators