Recap of Last Week

- Programmer's view of virtual memory
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions
- Challenges addressed:
  - Double (or more) the number of memory accesses
    - Solution: translation lookaside buffer (TLB)
  - Size of translation information (page table)
    - Solution: multi-level page tables

1

Today

- Simple virtual memory system example
- Linux and the virtual memory system
- Dynamic memory allocation

2

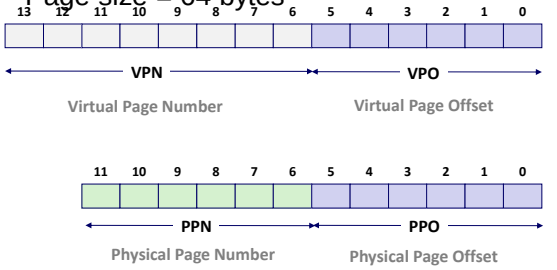
Review of Symbols

- Basic Parameters
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (bytes)
- Components of the virtual address (VA)
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

3

Simple Memory System Example

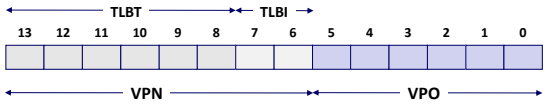
- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



4

1. Simple Memory System TLB

- 16 entries
- 4-way associative



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

5

2. Simple Memory System Page Table

Only show first 16 entries (out of 256)

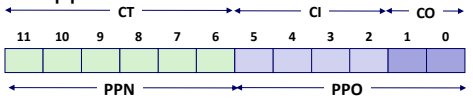
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

6

3. Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped



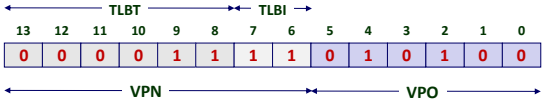
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

7

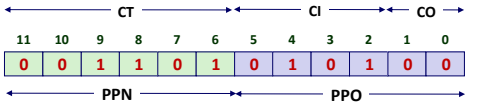
Address Translation Example #1

Virtual Address: `0x03D4`



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

Physical Address

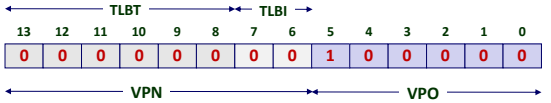


CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

8

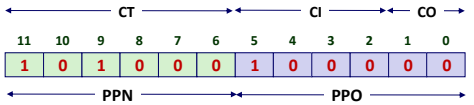
Address Translation Example #2

Virtual Address: 0x0020



VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address



CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

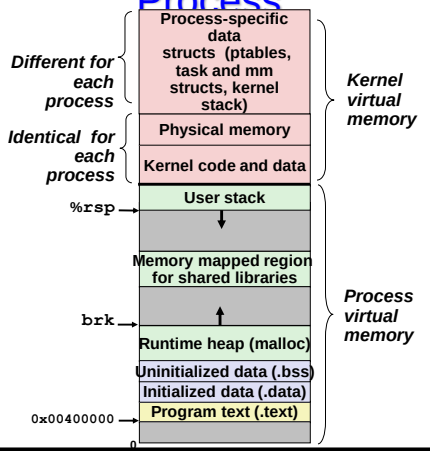
9

Today

- Simple virtual memory system example
- Linux and the virtual memory system
- Dynamic memory allocation

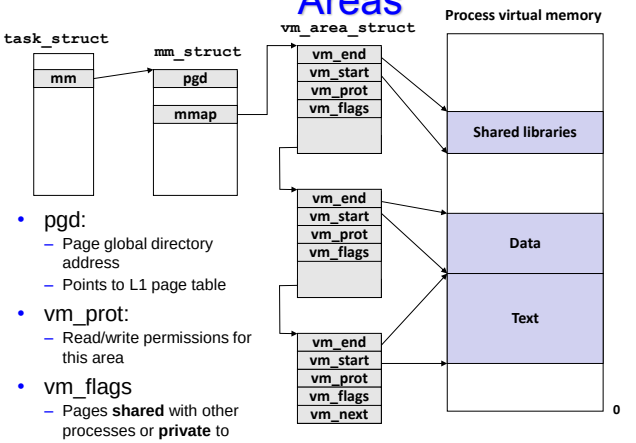
10

Virtual Address Space of a Linux Process



11

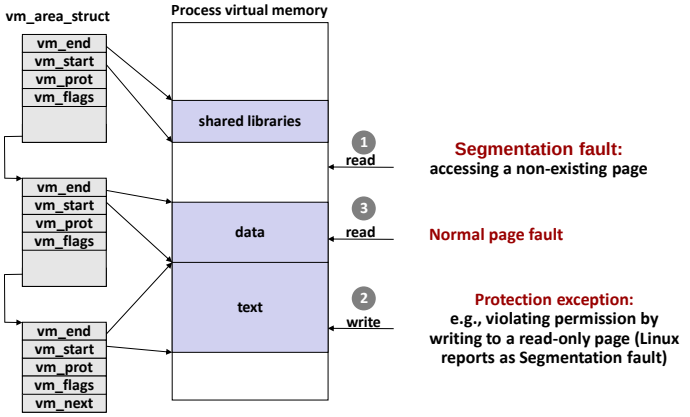
Linux Organizes VM as Collection of "Areas"



- pgd:
  - Page global directory address
  - Points to L1 page table
- vm_prot:
  - Read/write permissions for this area
- vm_flags
  - Pages **shared** with other processes or **private** to this process

12

Linux Page Fault Handling



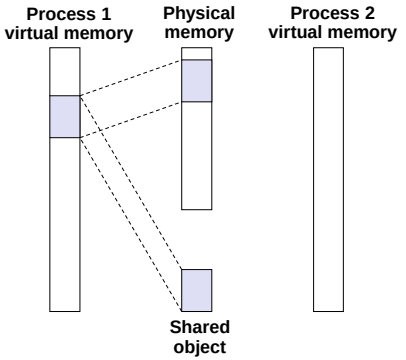
13

Memory Mapping

- VM areas initialized by associating them with disk objects.
  - Process is known as ***memory mapping***.
- Area can be *backed by* (i.e., get its initial values from) :
  - ***Regular file*** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - ***Anonymous file*** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (***demand-zero page***)
    - Once the page is written to (***dirtied***), it is like any other page
- Dirty pages are copied back and forth between memory and a special *swap file*

14

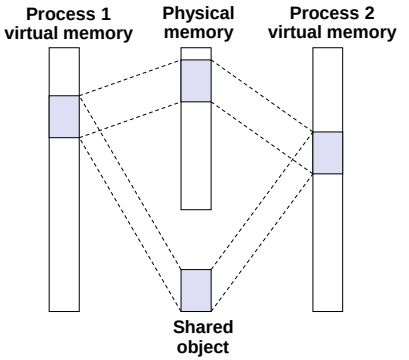
Sharing Revisited: Shared Objects



- Process 1 maps the shared object

15

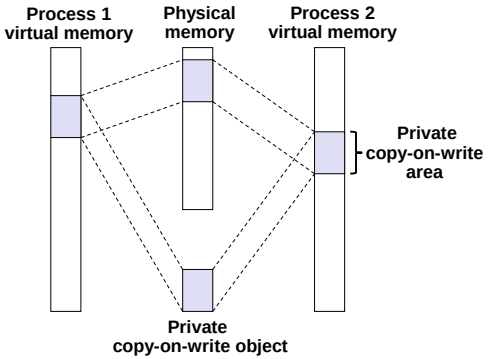
Sharing Revisited: Shared Objects



- **Process 2 maps the shared object.**
- **Notice how the virtual addresses can be different**

16

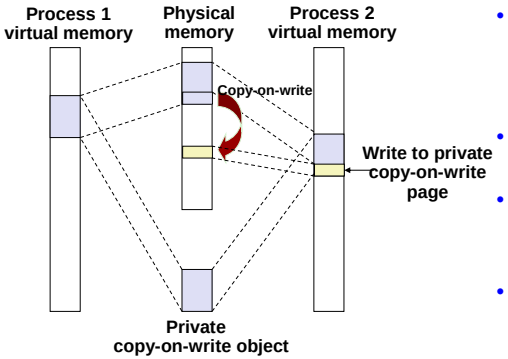
Sharing Revisited: Private Copy-on-write (COW) Objects



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

17

Sharing Revisited: Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

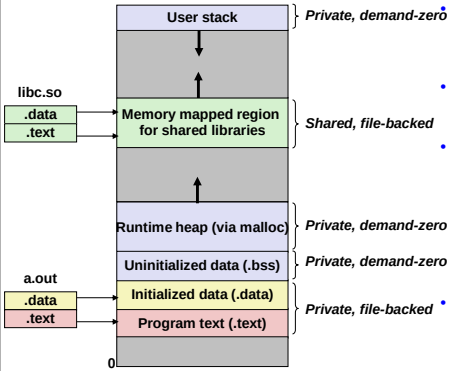
18

The `fork` Function Revisited

- VM and memory mapping explain how `fork` provides private address space for each process.
- To create virtual address for new new process
  - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new pages using COW mechanism.

19

The `execve` Function Revisited



To load and run a new program `a.out` in the current process using `execve`:

- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files
  - `.bss` and stack backed by anonymous files
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed

20

User-Level Memory Mapping

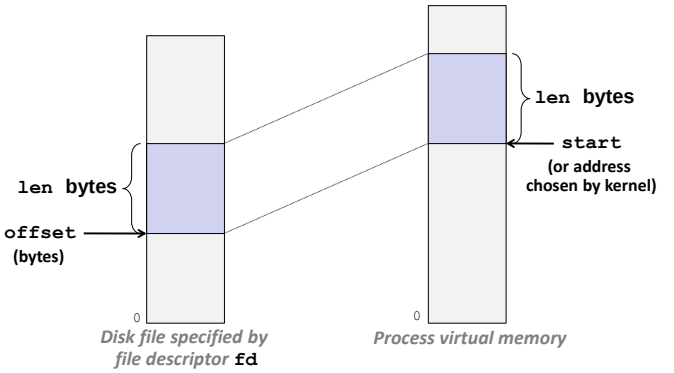
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map **`len`** bytes starting at offset **`offset`** of the file specified by file description **`fd`**, preferably at address **`start`**
  - **`start`**: may be 0 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`**)

21

User-Level Memory Mapping

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



22

Example: Using `mmap` to Copy Files

- Copying a file to `stdout` without transferring data to user space

```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

23

Memory System Summary

- Cache Memory
  - Purely a speed-up technique
  - Behavior invisible to application programmer and OS
  - Implemented totally in hardware
- Virtual Memory
  - Supports many OS-related functions
    - Process creation
      - Initial
      - Forking children
    - Task switching
    - Protection
  - Combination of hardware & software implementation
    - Software management of tables, allocations
    - Hardware access of tables
    - Hardware caching of table entries (TLB)

24

25

Today

- Simple virtual memory system example
- Linux and the virtual memory system
- Dynamic memory allocation

26

Dynamic Memory Allocation

27

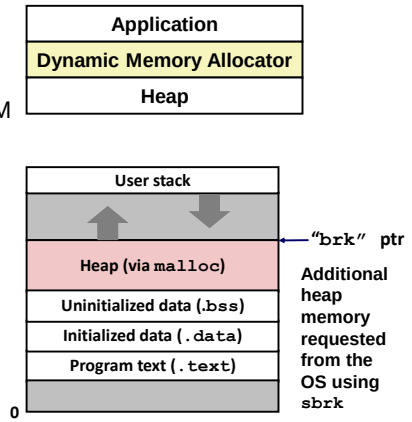
Harsh Reality

- *Memory Matters*
- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
    - Especially those based on complex, graph algorithms
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

28

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time
  - For data structures whose size is only known at runtime
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*



29

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - ***Explicit allocator*:** application 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, ML, and Lisp
- Will discuss explicit memory management today

30

The `malloc` Package

```
#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

32

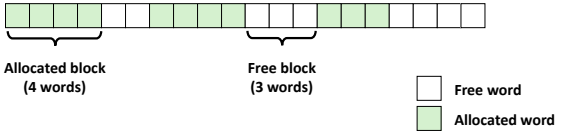
`malloc` Example

```
#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);
}
```

33

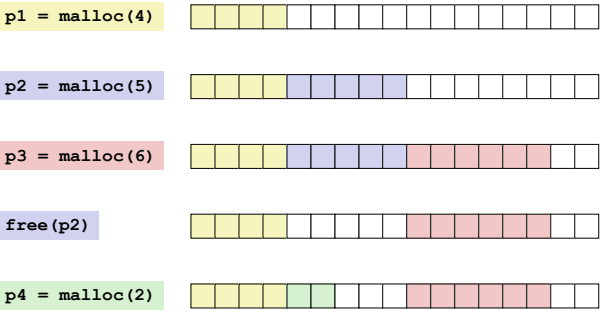
Assumptions Made in This Lecture

- Memory is word addressed.
- Words are int-sized.



34

Allocation Example



35

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 only place allocated blocks 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

36

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- Goals: maximize throughput and peak memory utilization
  - These goals are often conflicting
- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds
    - Throughput is 1,000 operations/second

37

Performance Goal: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- *Def: Aggregate payload* P_k
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- *Def: Current heap size* H_k
  - Assume H_k is monotonically nondecreasing
    - i.e., heap only grows when allocator uses **`sbrk`**
- *Def: Peak memory utilization after k+1 requests*
  - $U_k = (max_{i<=k} P_i) \ / \ H_k$

38

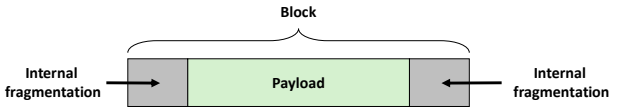
Fragmentation

- Poor memory utilization caused by *fragmentation*
  - ***internal*** fragmentation
  - ***external*** fragmentation

39

Internal Fragmentation

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

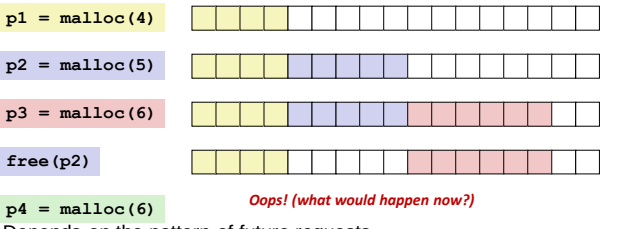


- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
  - Thus, easy to measure

40

External Fragmentation

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



- Depends on the pattern of future requests
  - Thus, difficult to measure

41

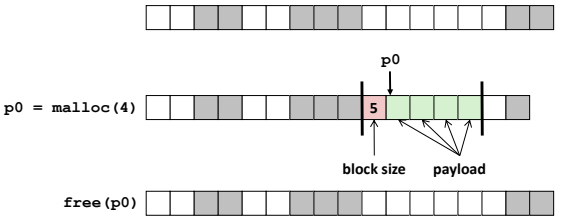
Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- 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?
- How do we reinsert freed block?

42

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

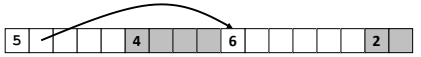


43

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

44

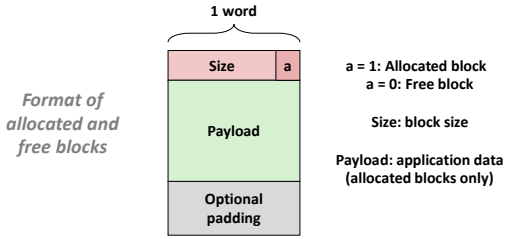
Today

- Simple virtual memory system example
- Linux and the virtual memory system
- Dynamic memory allocation
  - Explicit memory management
    - Implicit free lists

45

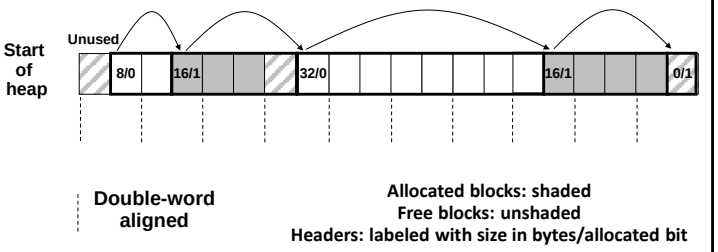
Method 1: Implicit List

- 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



46

Detailed Implicit Free List Example



47

Implicit List: Finding a Free Block

- *First fit:*
  - Search list from beginning, choose ***first*** free block that fits:

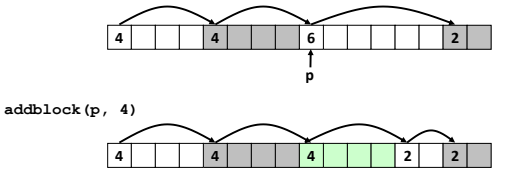
```
p = start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p  <= len)))    \\ too small
  p = p + (*p & -2);     \\ goto next block (word addressed)
```

  - 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

48

Implicit List: Allocating in Free Block

- Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to 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
```

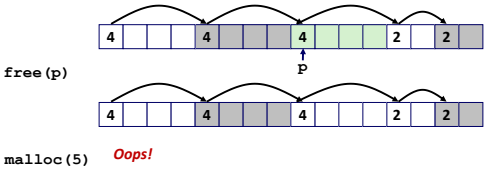
49

Implicit List: 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"

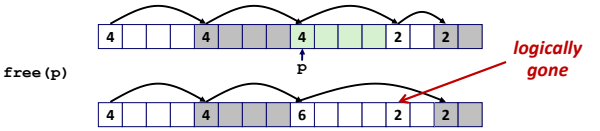


There is enough free space, but the allocator won't be able to find it

50

Implicit List: Coalescing

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



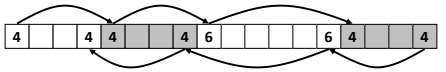

```
void free_block(ptr p) {
    *p = *p & -2;          // clear allocated flag
    next = p + *p;         // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;     // add to this block if
}                          //    not allocated
```

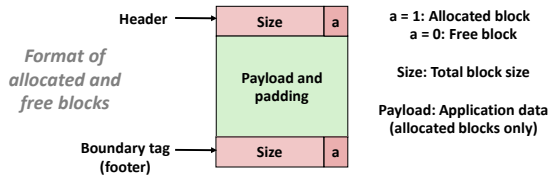
  - But how do we coalesce with *previous* block?

51

Implicit List: Bidirectional Coalescing

- *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!



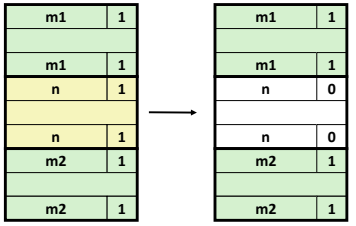


52

Constant Time Coalescing

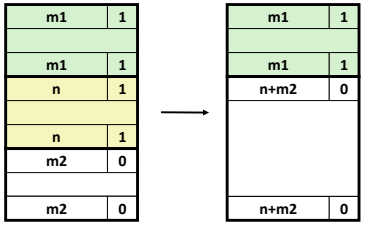
53

Constant Time Coalescing (Case 1)



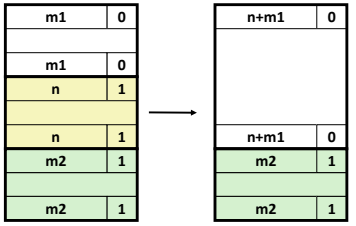
54

Constant Time Coalescing (Case 2)



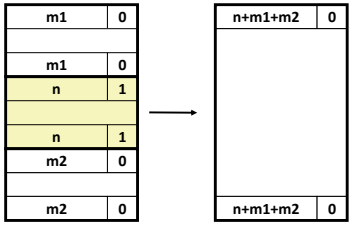
55

Constant Time Coalescing (Case 3)



56

Constant Time Coalescing (Case 4)



57

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation***: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list
- Splitting policy:
  - When do we go ahead and 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

59

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case
    - even with coalescing
- Memory usage:
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for `malloc/free` 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

60

Today

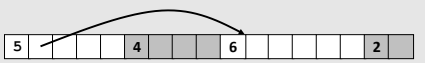
- Simple virtual memory system example
- Linux and the virtual memory system
- Dynamic memory allocation
  - Explicit memory management
    - Implicit free lists
    - Explicit free lists

61

Keeping Track of Free Blocks

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

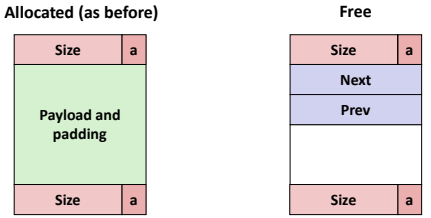
- Method 2: *Explicit free list* among the free blocks using pointers



- 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

62

Explicit Free Lists



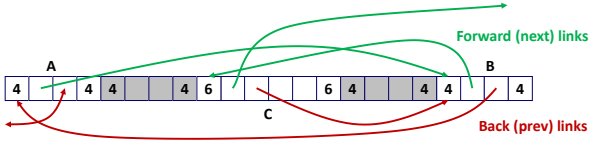
- Maintain list(s) of *free* blocks, not *all* blocks
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

63

Explicit Free Lists

- Logically:

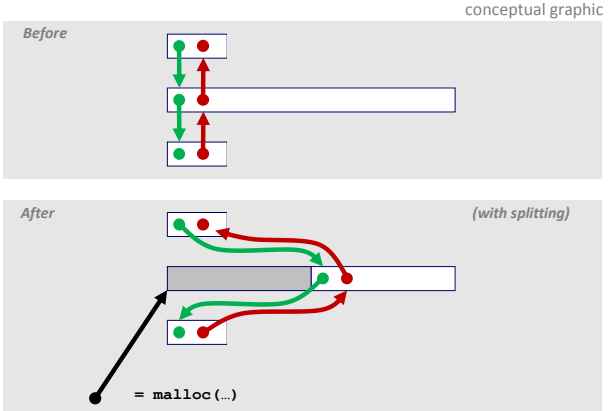
- Physically: blocks can be in any order



64

Allocating From Explicit Free Lists

conceptual graphic



65

Freeing With Explicit Free Lists

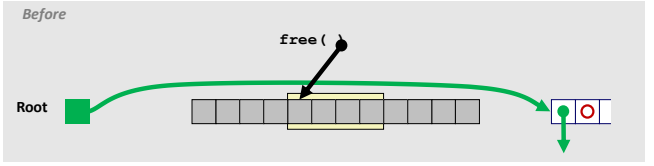
- *Insertion policy*: Where in the free list do you put a newly freed block?
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:

    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

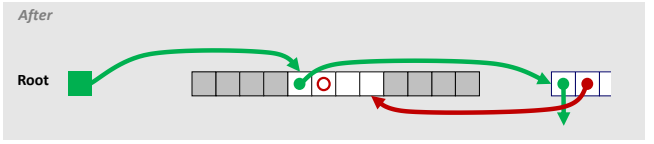
66

Freeing With a LIFO Policy (Case 1)

conceptual graphic



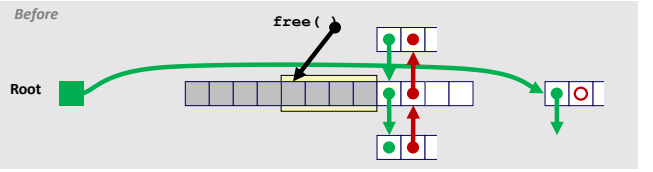
- Insert the freed block at the root of the list



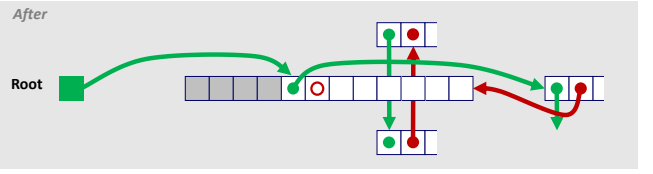
67

Freeing With a LIFO Policy (Case 2)

conceptual graphic



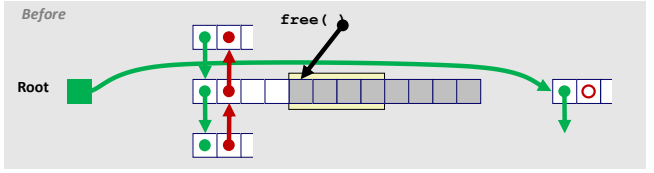
- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



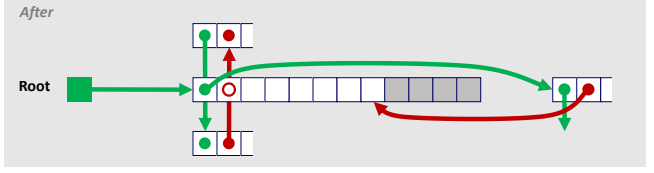
68

Freeing With a LIFO Policy (Case 3)

conceptual graphic



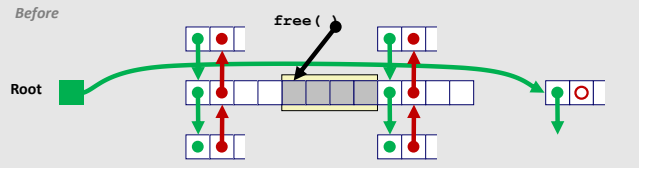
- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



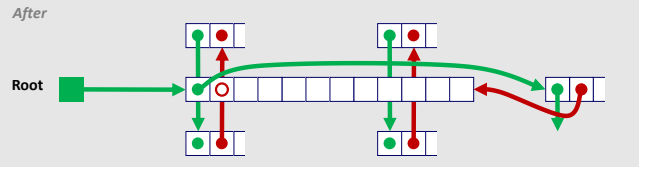
69

Freeing With a LIFO Policy (Case 4)

conceptual graphic



- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new



70

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?
- Most common use of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

71

Today

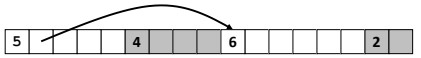
- Simple virtual memory system example
- Linux and the virtual memory system
- Dynamic memory allocation
  - Explicit memory management
    - Implicit free lists
    - Explicit free lists
    - Segregated free lists

72

Keeping Track of Free Blocks

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

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

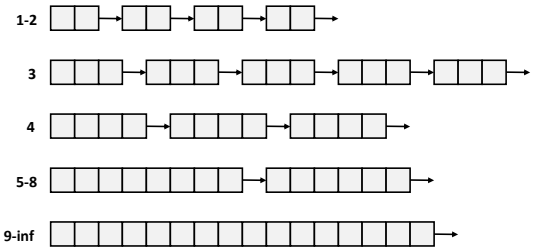


- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

73

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

74

Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size *n*:
  - Search appropriate free list for block of size *m* > *n*
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using **`sbrk()`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

75

Seglist Allocator (cont.)

- To free a block:
  - Coalesce and place on appropriate list
- Advantages of seglist allocators
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap
    - Extreme case: Giving each block its own size class is equivalent to best-fit

76

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation
- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

77