Heap Memory Management

CS439: Principles of Computer Systems

March 11, 2015

Last Time

- Paging Mechanism
  - Page Faults
- Paging Policies
  - Replacement algorithms
    - FIFO, Optimal, LRU, Clock, Second Chance
  - Local vs. Global

Today's Agenda

- Paging Policies
  - Load Control Strategies
  - Page Sizes
- Heap Memory Management
  - Explicit vs Automatic/Implicit
  - Allocation techniques
    - Contiguous allocation (bump pointer)
    - Free lists (analogous to pages in memory)
  - Explicit deallocation

Virtual Memory

Thrashing

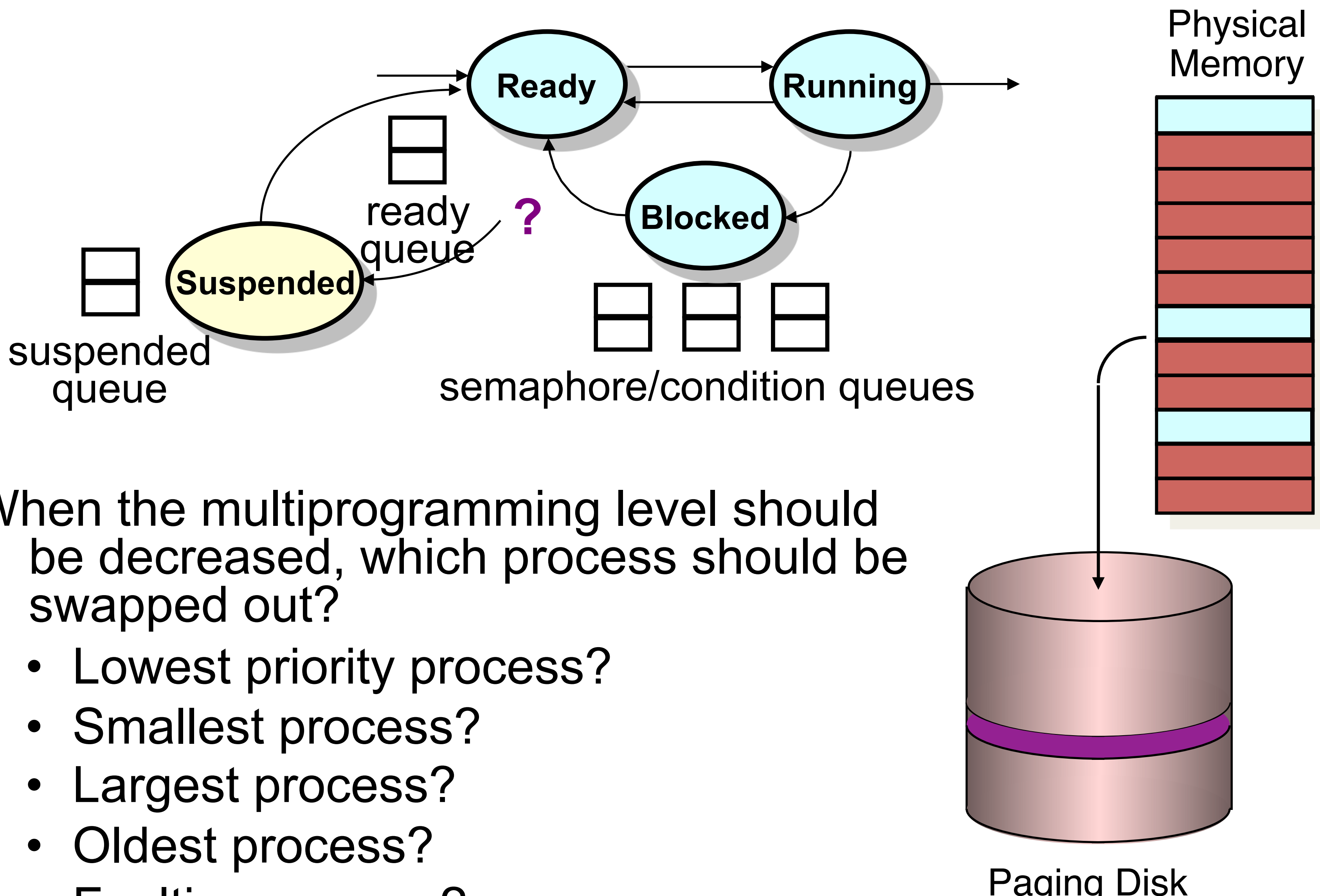
- *Thrashing* occurs when the memory is over-committed and pages are tossed out while they are still in use
- Many memory references cause pages to be faulted in
  - Very serious and very noticeable loss of performance

How do we limit thrashing in a multiprogrammed system?

Load Control

- *Load control* refers to the number of processes that can reside in memory at one time
- Working set model provides implicit load control by only allowing a process to execute if its working set fits in memory
- BUT process frame allocations are variable
- What happens when the total number of pages needed is greater than the number of frames available?
  - Processes are swapped out to disk

Load Control



When the multiprogramming level should be decreased, which process should be swapped out?

- Lowest priority process?
- Smallest process?
- Largest process?
- Oldest process?
- Faulting process?

Load Control: Text Description

- When a process is totally swapped out of memory it is put onto swap (aka the paging disk)
- Adds another stage to the process life cycle
  - This new stage is called suspended
    - We also saw this stage in relocation, when we also swapped out entire processes
  - Can go from any of the other states to suspended
    - Usually blocked or ready
  - Process can go from suspended to ready

Another Decision: Page Sizes

Page sizes are growing slowly but steadily. Why?

- Benefits for small pages: more effective memory use, higher degree of multiprogramming possible
- Benefits for large pages: smaller page tables, reduced I/O time, fewer page faults
- Growing because:
  - memory is cheap---page tables could get huge with small pages and internal fragmentation is less of a concern
  - CPU speed is increasing faster than disk speed, so page faults cause a larger slow down

iClicker Question

Can an application modify its own translation tables (however they are implemented)?

A. Yes
B. No

Summary: Paging

We've considered:

- Placement Strategies
  - None needed, can place pages anywhere
- Replacement Strategies
  - What to do when more jobs exist than can fit in memory
- Load Control Strategies
  - Determine how many jobs can be in memory at one time

Summary: Paging

The Good

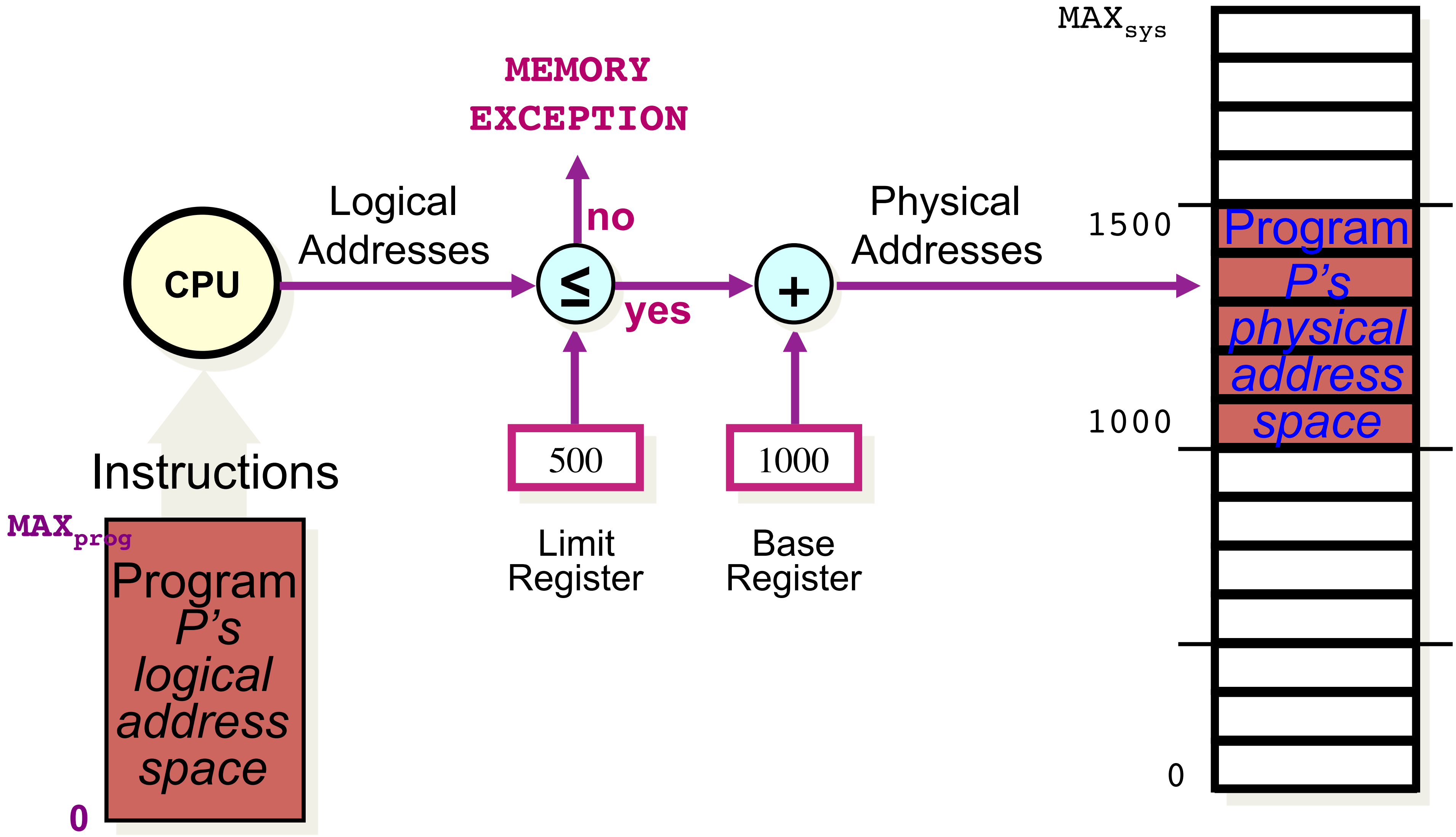
- Eliminates the problem of external fragmentation
- Allows sharing of memory pages amongst processes
- Enables processes to run when they are only partially loaded into main memory

The Cost

- Translating from a virtual address to a physical address is time consuming
- Requires hardware support (TLB) to be decently efficient
- Requires more complex OS to maintain the page table

The expense of memory accesses and the flexibility of paging make paging cost effective.

Dynamic Relocation: Address Translation

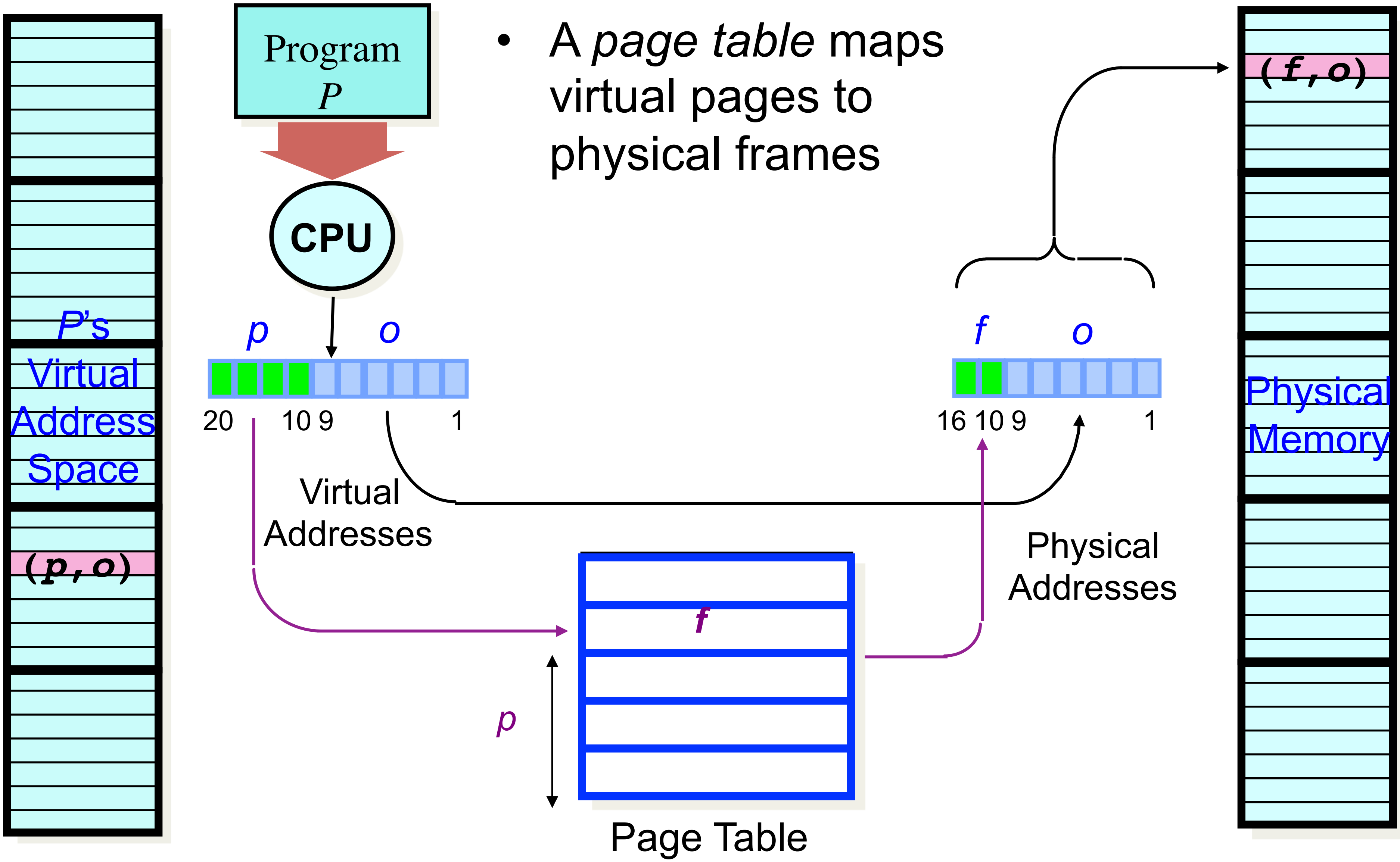


Dynamic Relocation: Text Description

- How we get from a program's logical address space to physical memory.
- The bound register is used to make sure that the program isn't trying to access memory outside of its space. For example, if the programs logical address space ranges from 0 to 500 then the bound register would hold a value of 500.
- The base register holds the beginning of that program's addresses in physical memory. So if the programs memory started at address 1000 then the base register would hold 1000.
- Steps to Address Translation:
  - The program gives a logical address, or instruction, to the CPU.
  - The MMU now does 2 things at once:
    - It checks the address against the bound register, if the address is greater than the bound register a memory exception is thrown. This exception is a hardware interrupt that will be handled by the OS. This exception indicates the program was trying to access something that doesn't belong to it.
    - It adds the base register to the logical address to get the physical address. For example logical address 8 would become 1008 if the base register was 1000.
- NOTE: Even when the address is successfully translated, you are not protected from memory errors: you could still accidentally try to access memory in your address space that hasn't be initialized yet.

Virtual Address Translation

- A *page table* maps virtual pages to physical frames



Memory Management: Putting it all Together

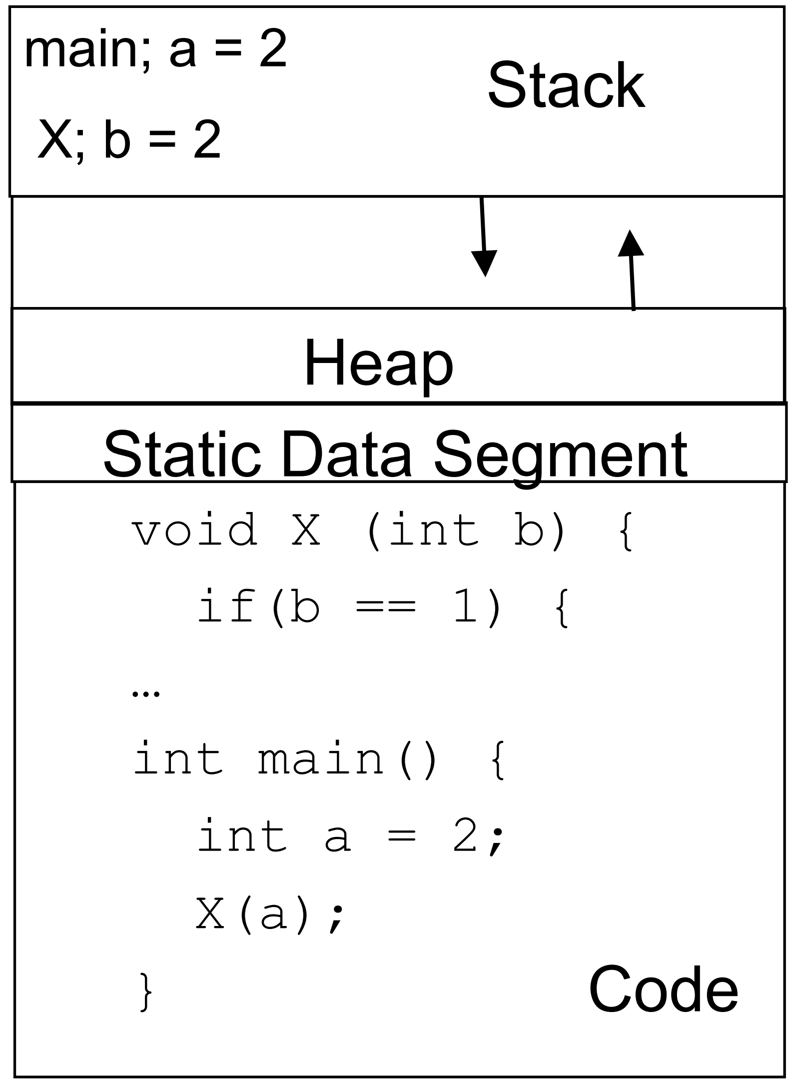
- Dynamic Relocation with Base and Bounds:
  - Simple, but inflexible
  - Degree of multiprogramming limited, memory limited to physical memory size, no sharing of memory, memory allocation/deallocation difficult
  - Use compaction to solve external fragmentation
- Paging:
  - Process generates virtual addresses from 0 to Max
  - OS divides processes into pages
    - manages a page table for every process
    - manages the pages in memory
  - Simplifies memory allocation since any page can be allocated to any frame
  - Page tables can be very large
  - Page Replacement Algorithms
    - FIFO, Optimal, LRU, Clock, Enhanced Clock, Working Set
  - Design Considerations (page size, global vs. local, …)

Heap Memory Management

Heap Memory Management

- Where and how do we manage dynamically allocated (user) memory (a.k.a. the heap!)?
  - Program/runtime system requests memory from the OS for the heap
  - OS gives memory to a process 1 to *k* pages at a time (Why?)
  - The runtime system manages the heap memory
  - Typically, the memory is not returned to the OS until the program ends
- How does the runtime system *efficiently* create and recycle memory on behalf of the program?
  - What makes this problem important?
  - What makes this problem hard?

Reminder



main; a = 2
Stack
X; b = 2
Heap
Static Data Segment
void X (int b) {
if(b == 1) {
…
int main() {
int a = 2;
X(a);
}
Code

What’s in the heap?

- Dynamically allocated program objects and data
- Needed when required memory size is not known until the program runs

Two Categories of Heap Memory Management

- Explicit memory management
  - The program(mer) explicitly manages all of the memory
  - Allocation: malloc/new
  - Deallocation: free/delete
  - Pointers: anything may or may not be a pointer
  - Example languages: C, C++
- Automatic memory management (Garbage Collection)
  - The program(mer) explicitly allocates memory, but the runtime system manages it
  - Allocation: new
  - Deallocation: *None*
  - Pointers: Program and runtime system know all pointers
  - Example languages: Java, ML, Python

Key Issues

- How to allocate the memory
  - How to organize the memory space
  - Fast allocation
  - Low fragmentation (wasted space)
- How to deallocate the memory
  - Fast reclamation
  - Discriminating live (in use) objects and garbage (automatic memory management only)

Explicit Memory Management

Two Pieces

- User
  - Explicitly allocates memory by requesting a number of bytes
  - May explicitly request deallocation of memory when it is no longer used
- Runtime System
  - Receives requests for memory
  - Identifies appropriate location for allocation
    - If allocation doesn’t fit, requests more memory from the Operating System
  - Returns pointer
  - Later, frees allocation on request

Runtime System Requirements

- Handle arbitrary request sequences
  - Memory may be allocated and freed in any order
- Make immediate responses to requests
  - Cannot reorder/buffer requests to improve performance
- Use only the heap
  - Any data structures (such as free list) used by `malloc()`/`free()` must be stored on the heap
- Align blocks (e.g., on 8-byte boundary)
  - Blocks must be able to hold any type of data object
- Not modify allocated blocks
  - Can only manipulate or change free blocks
  - Cannot modify (or move!) other blocks after they are allocated
  - Results in fairly simple allocation policies

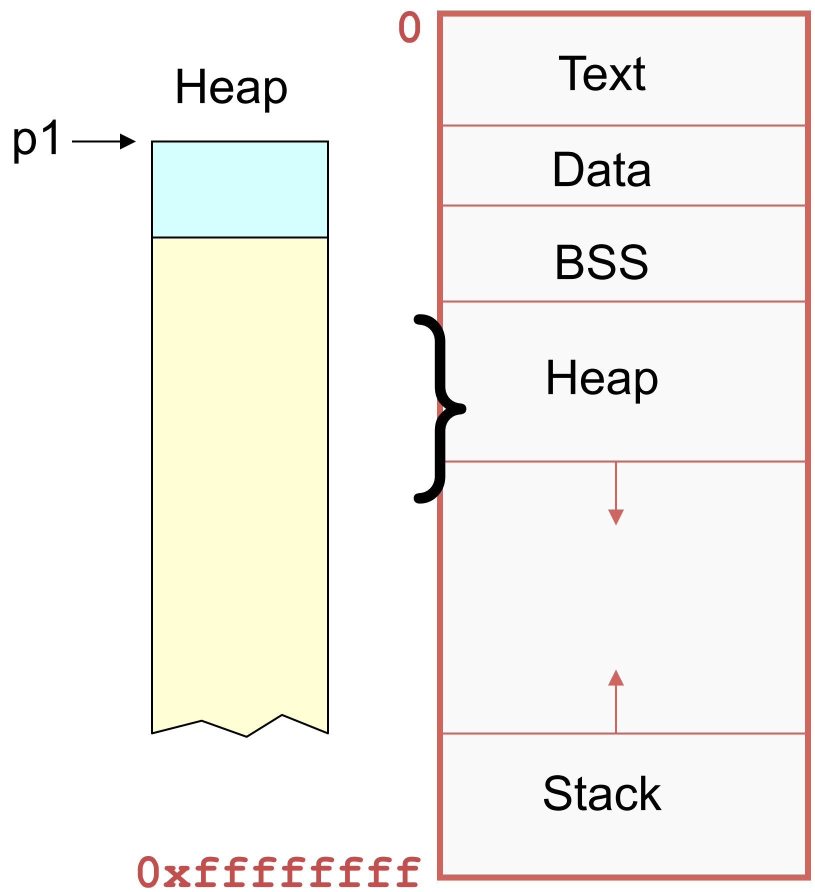
Allocation Techniques

1. Bump-pointer
   – Contiguous allocation (for all requested blocks)
   – Pointer begins at start of heap
   – As requested, bytes allocated, and pointer is "bumped" past allocation

Bump Pointer (C example)

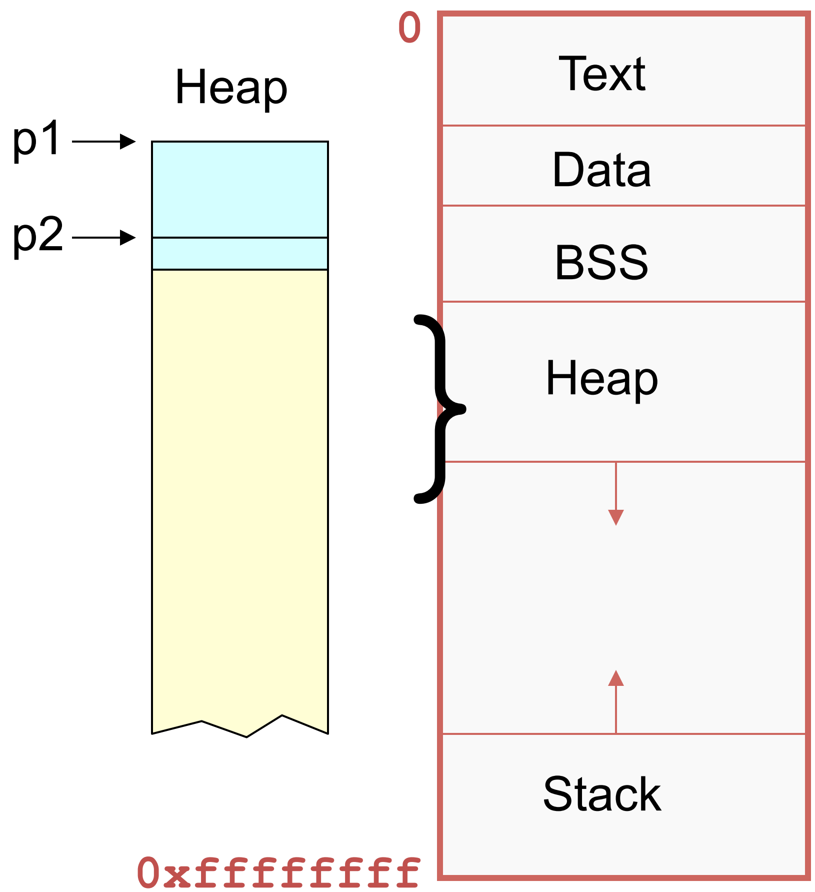
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Bump Pointer (C example)

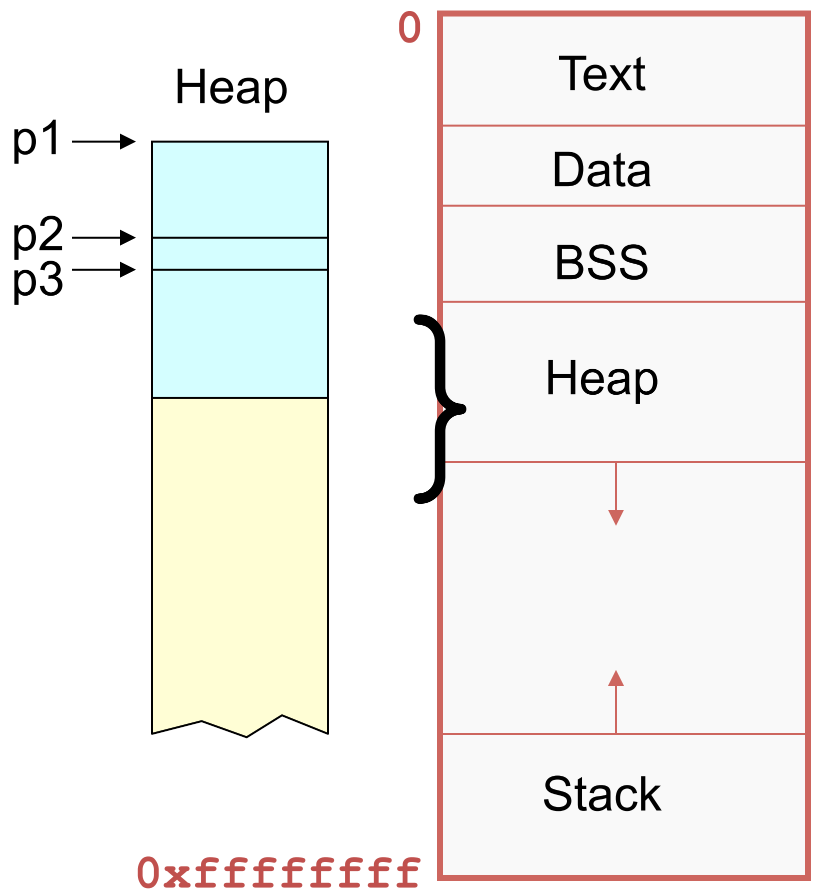
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

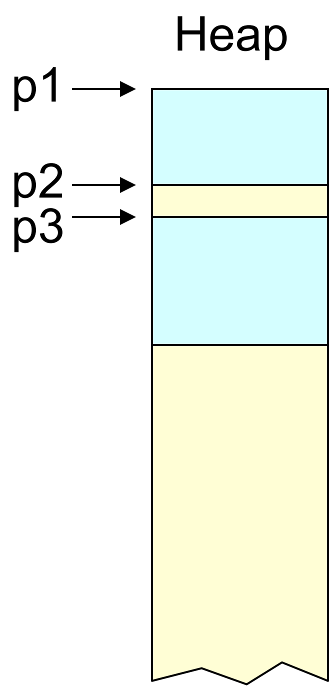
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

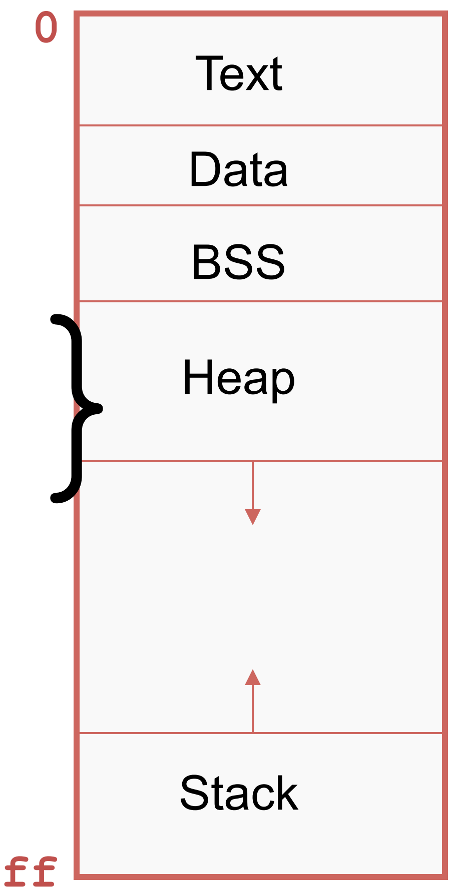


Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

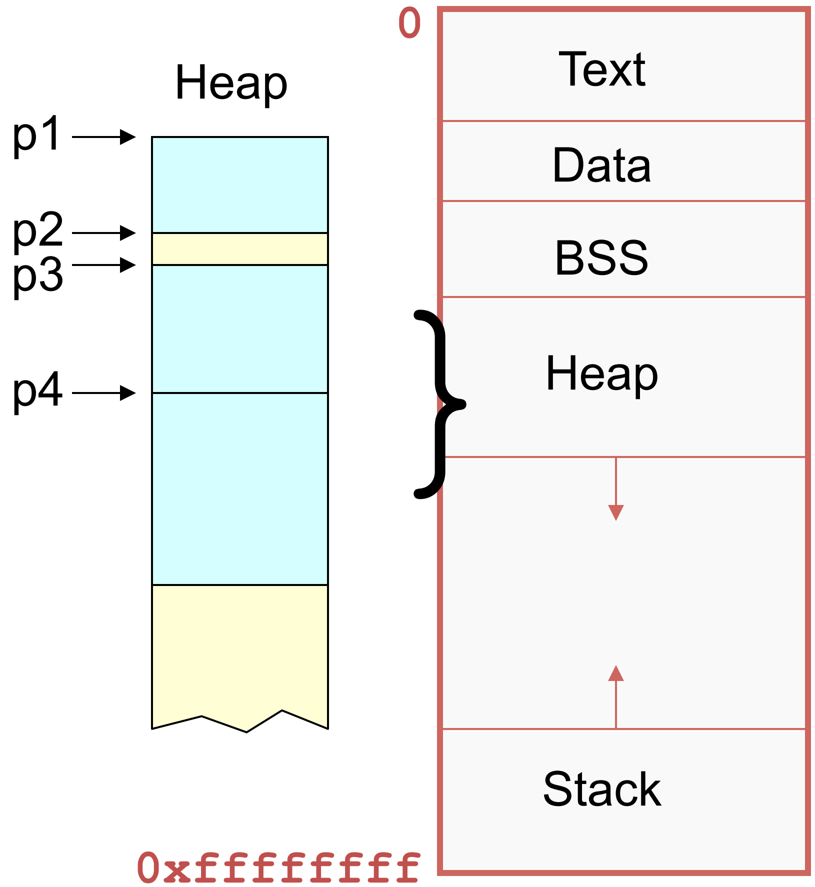




Bump Pointer (C example)

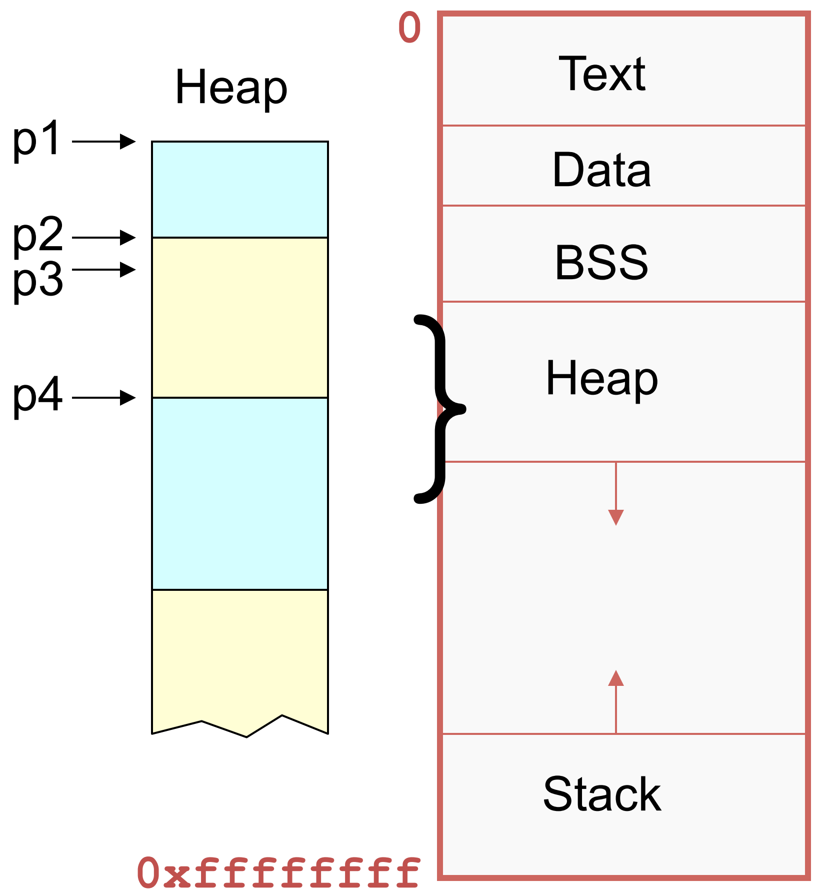
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



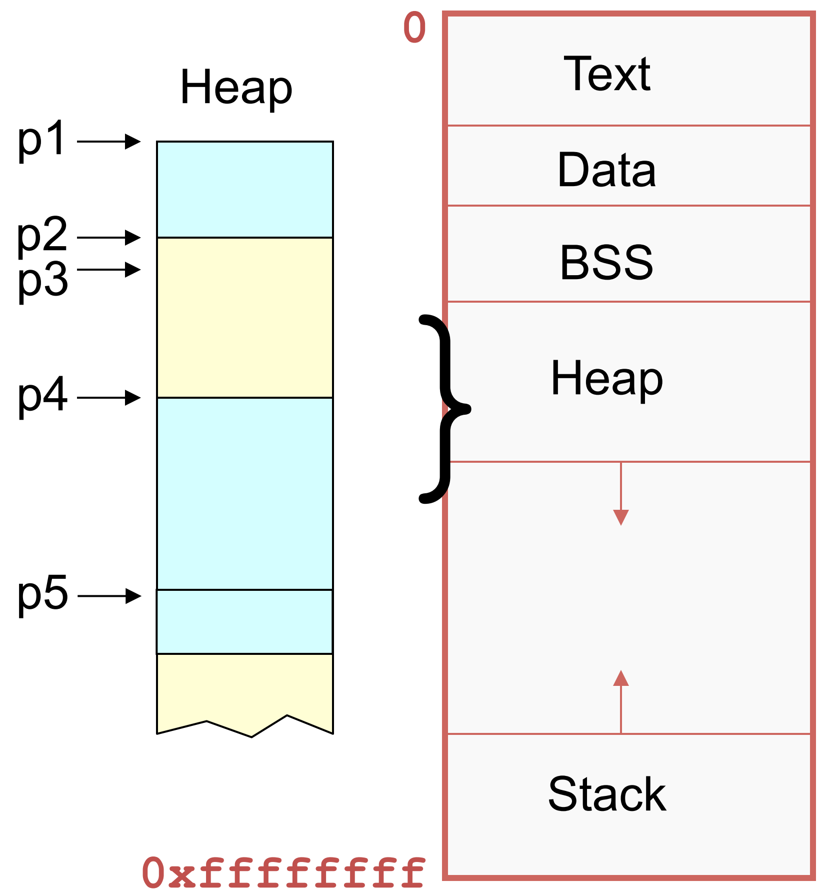
Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



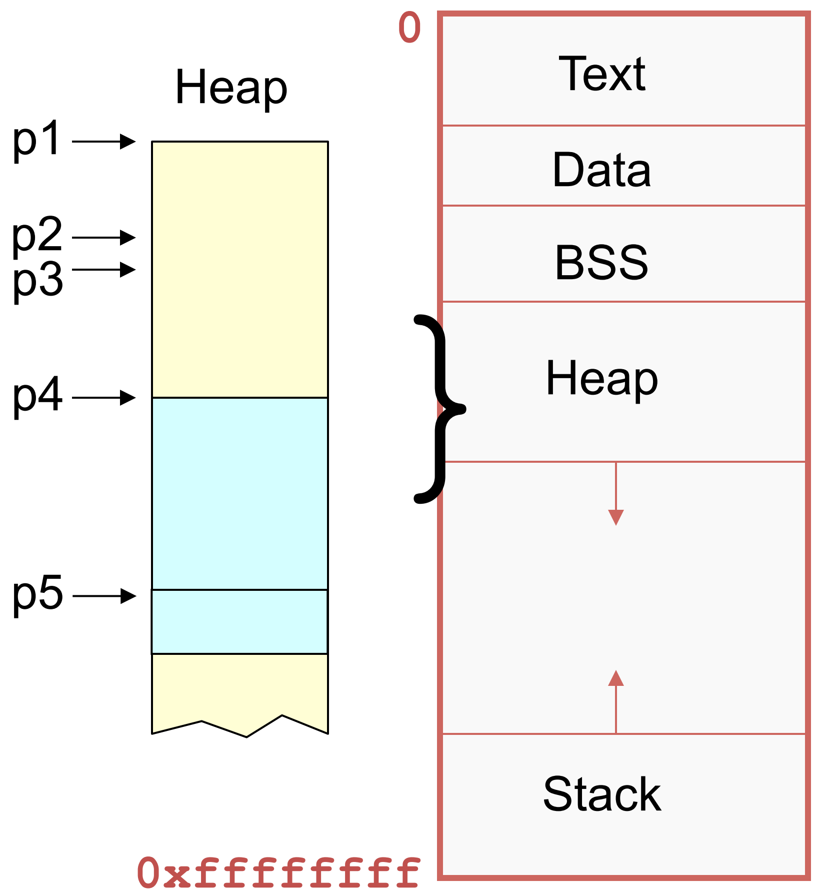
Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



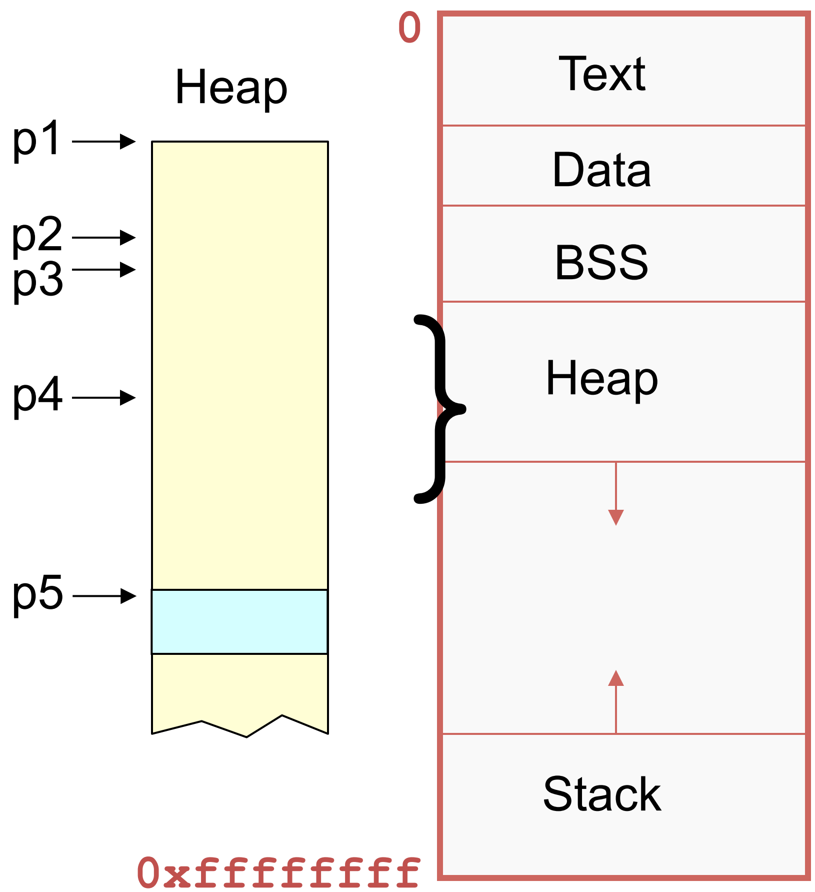
Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



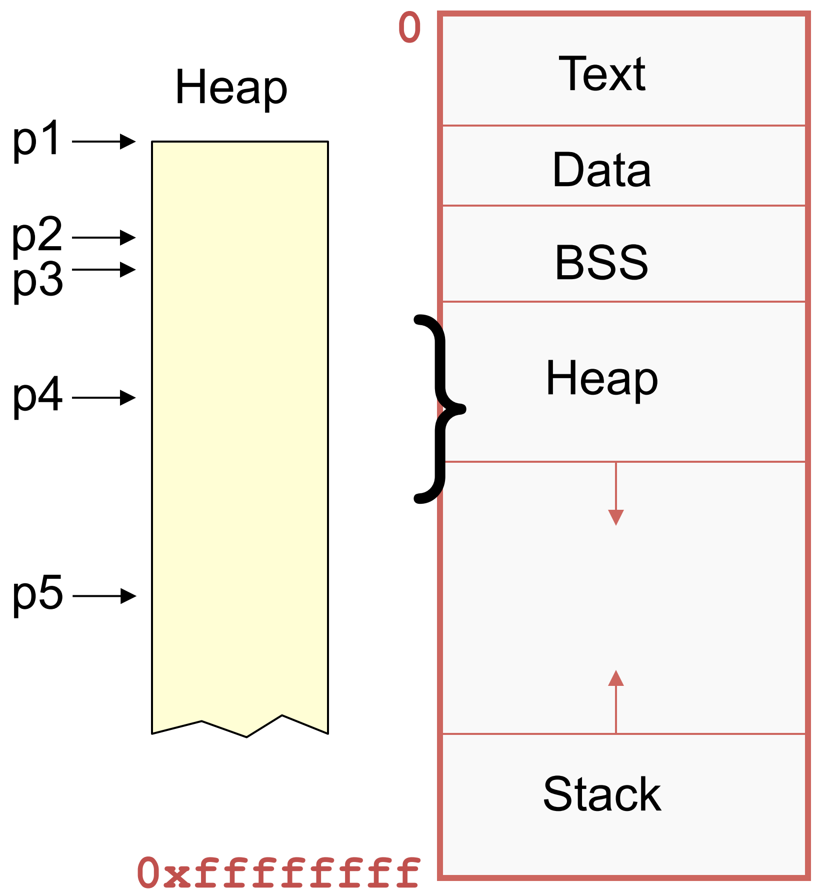
Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer (C Example): Text Description

- In this address space, the heap grows downwards towards higher addresses
  - order of segments starting at address 0: text, Data, BSS and heap
    - stack starts at some high address later and grows toward lower addresses
    - note: stack and heap grow towards each other
- List of C commands and their consequences:
  - note: originally heap is empty and bump pointer at address 100
  - char *p1 = malloc(3)
    - 3 bytes are allocated on the heap
    - p1 holds address 100, the beginning of the 3 bytes
    - bump pointer moved to address 103
  - char* p2 = malloc(1)
    - 1 byte is allocated on the heap
    - p2 holds address 103, the beginning of the 1 byte
    - bump pointer moved to address 104
  - char* p3 = malloc(4)
    - 4 bytes are allocated on the heap
    - p3 holds address 104, the beginning of the 4 bytes
    - bump pointer moved to address 108
  - free(p2)
    - memory that p2 points to is deallocated
    - now is a gap in the heap from 103 to 104
      - gap stays because bump pointer allocation doesn't move memory around once it has been allocated
  - char* p4 = malloc(6)
    - 6 bytes allocated on the heap
    - p4 holds address 108, the beginning of the 6 bytes
    - bump pointer moved to address 114
  - free(p3)
    - memory that p3 points to is deallocated
    - now is a gap in the heap from 103 to 108
      - this new gap is coalesced with the old gap created by freeing p2
  - char* p5 = malloc(2)
    - 2 bytes allocated on the heap
    - p5 points to address 114, the beginning of the 2 bytes
    - bump pointer is moved to address 116
  - free(p1)
    - memory that p1 points to is deallocated
    - now a gap in heap from 100 to 108
  - free(p4)
    - memory that p4 points to is deallocated
    - now gap in heap from 100 to 114
  - free(p5)
    - memory that p5 points to is deallocated
    - now a gap in heap from 100 to 116
  - now: all of the memory that was originally allocated on the heap has been reclaimed
  - note: pointers still hold their old addresses because they were not NULLed out
    - this means can still try to use them but will get a segfault

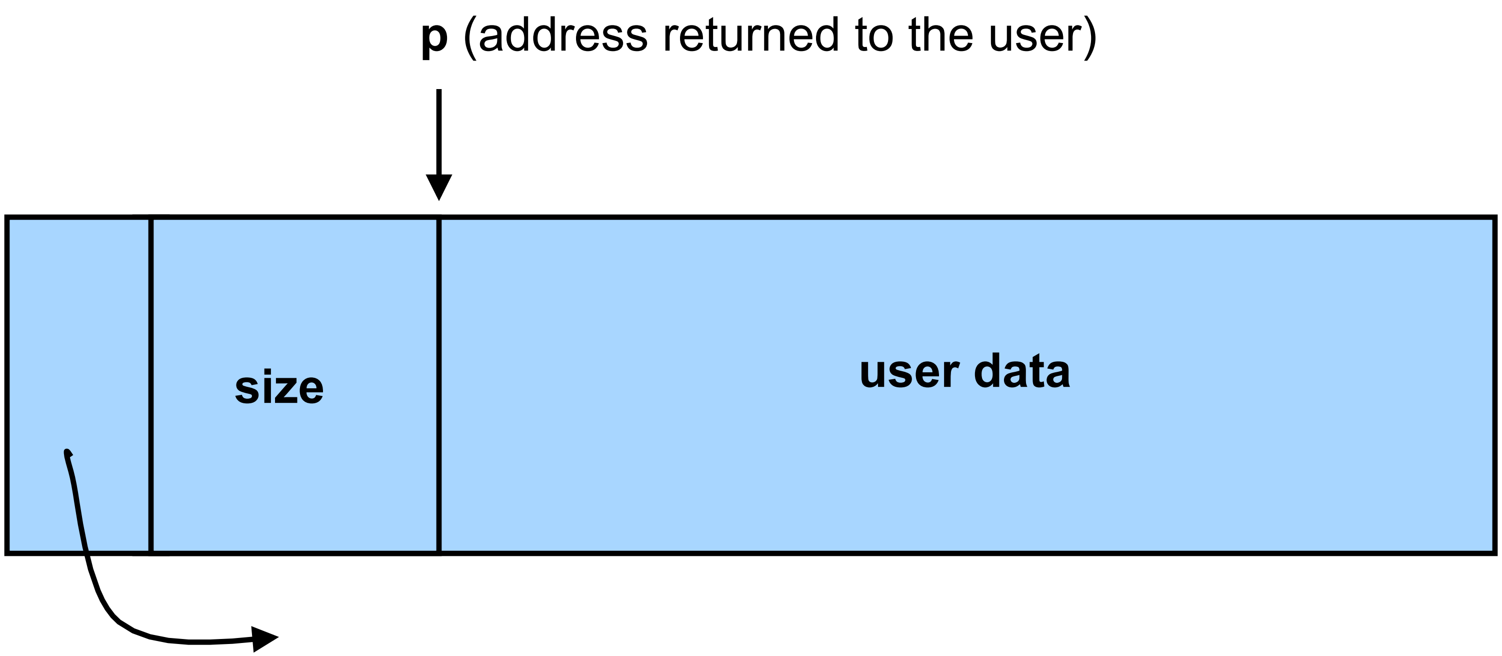
Allocation Techniques

2. Free List
   - Divides memory into some size blocks
   - Maintains a free list
     - Must be stored in the heap
     - Uses a special structure for free blocks
   - To allocate memory, find block in the free list
     - Using what algorithm? Guess!
     - If the right size does not exist, carves up a bigger piece
   - To deallocate memory, put memory back on the free list

Free Block: Pointer, Size, Data

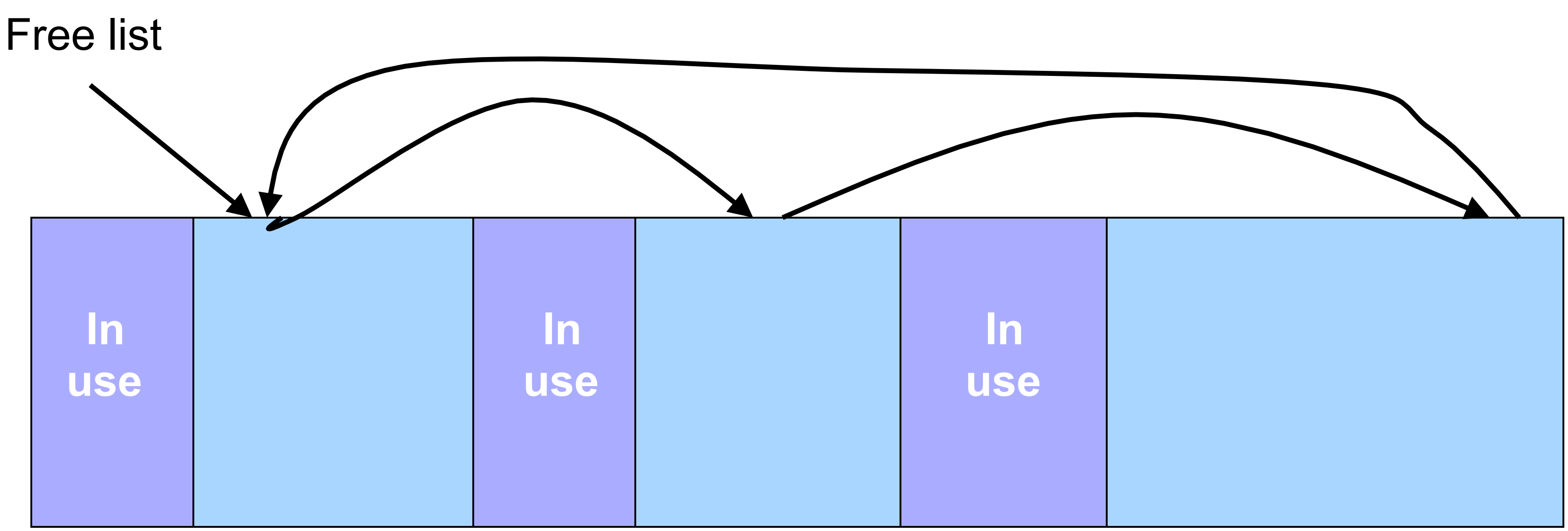
Free block in memory

- Pointer to the next free block
- Size of the free block
- Free space (that can be allocated to user)



Free List: Circular Linked List

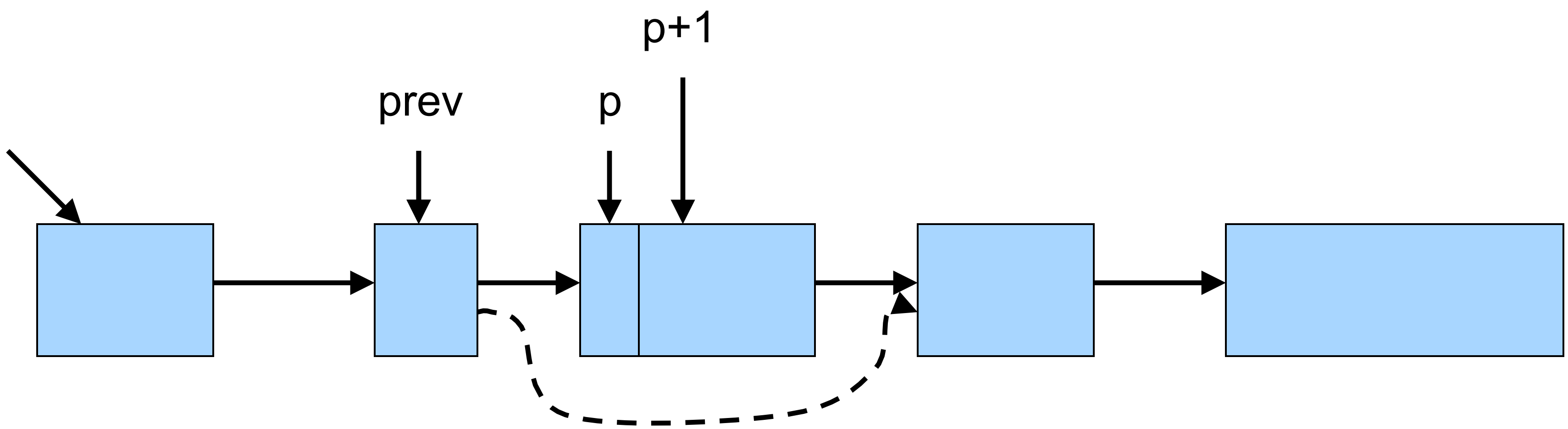
- Free blocks, linked together
  - Example: circular linked list
- List may be ordered by address or by size, depending on allocation algorithm



Choosing the Spot
First Case: A Perfect Fit

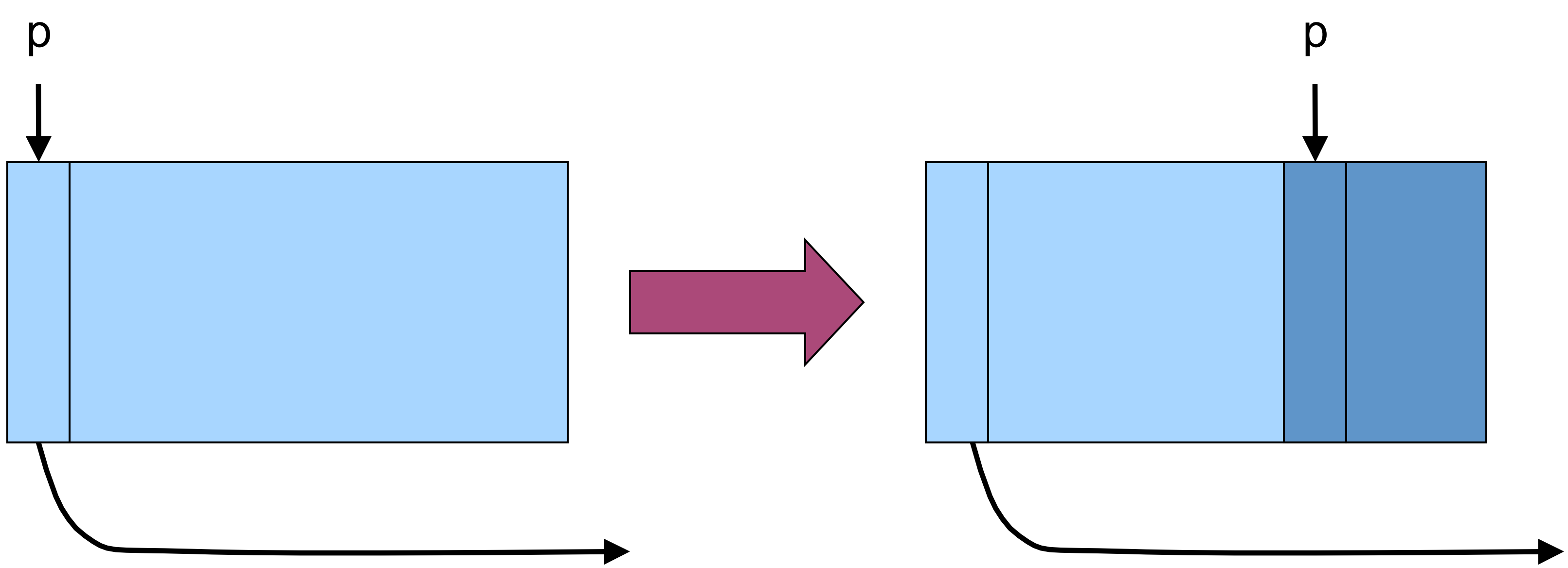
Suppose the block is a perfect fit

- Remove the element from the list
- Link the previous element with the next element
- Return the current element to the user (skipping header)



Choosing the Spot
Second Case: Block is Too Big

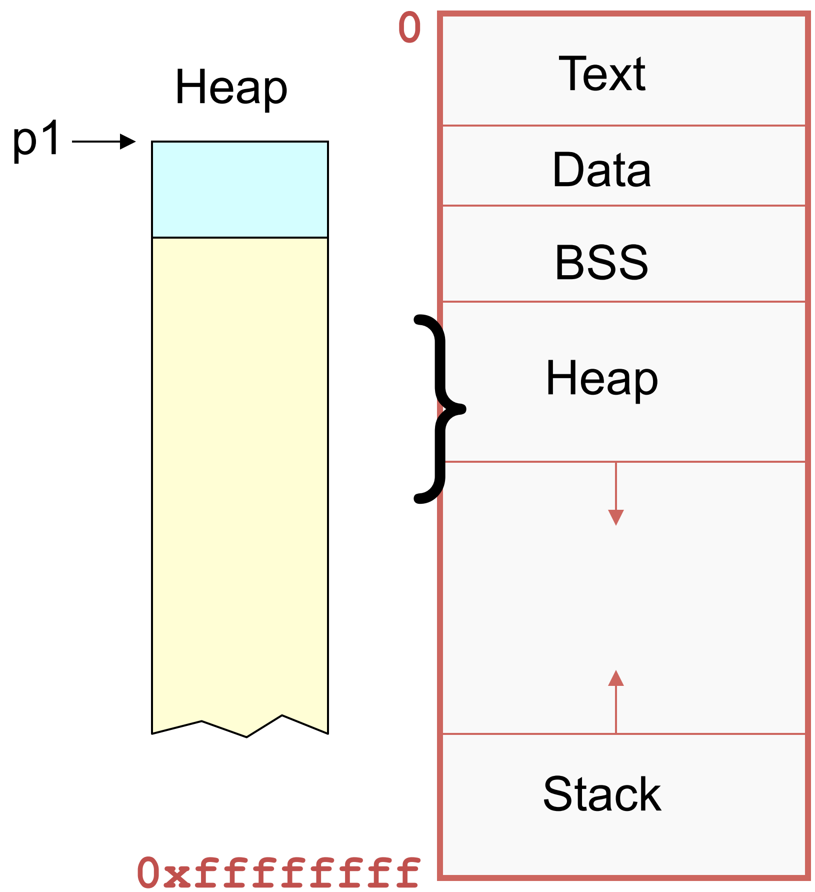
- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list
  - Allocate the second block to the user



Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

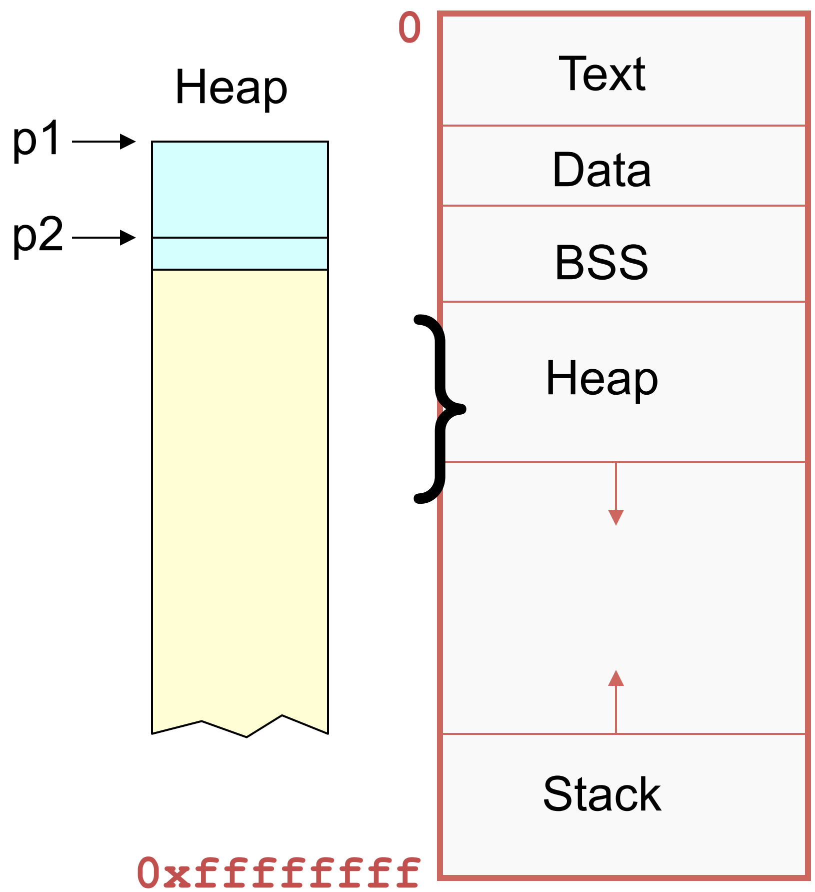
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

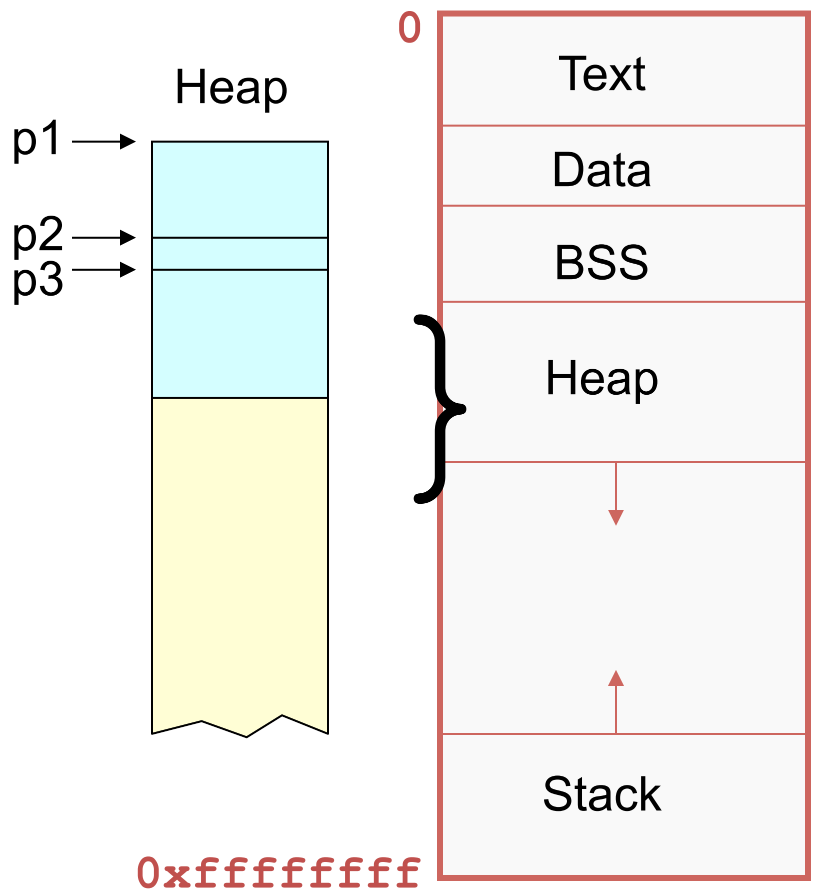
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

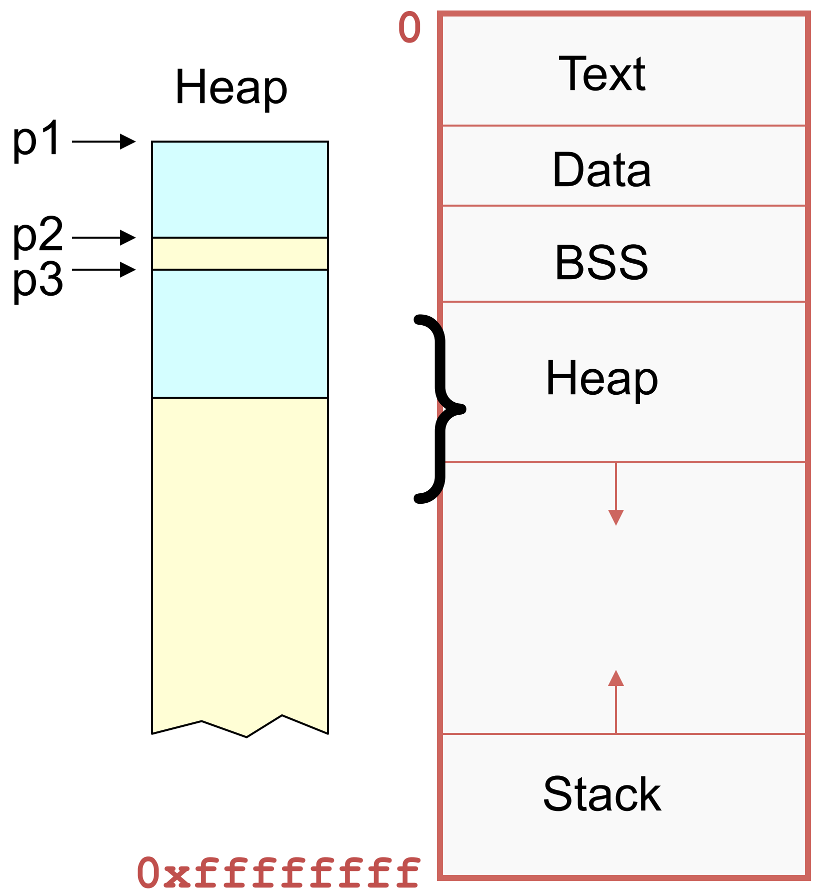
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

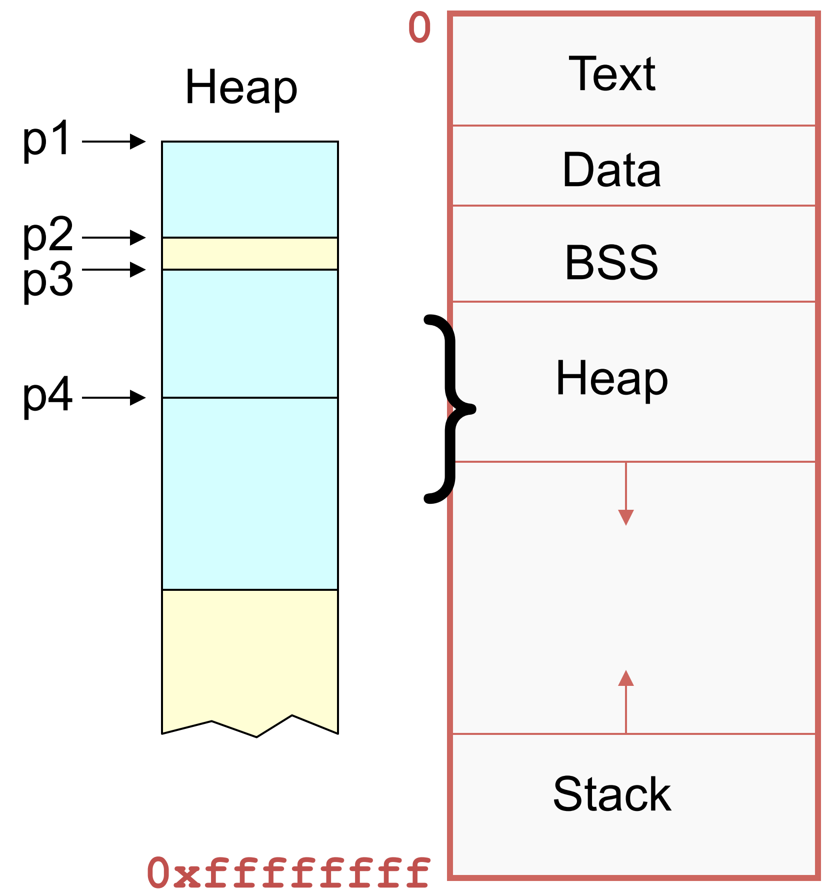
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List (C example)

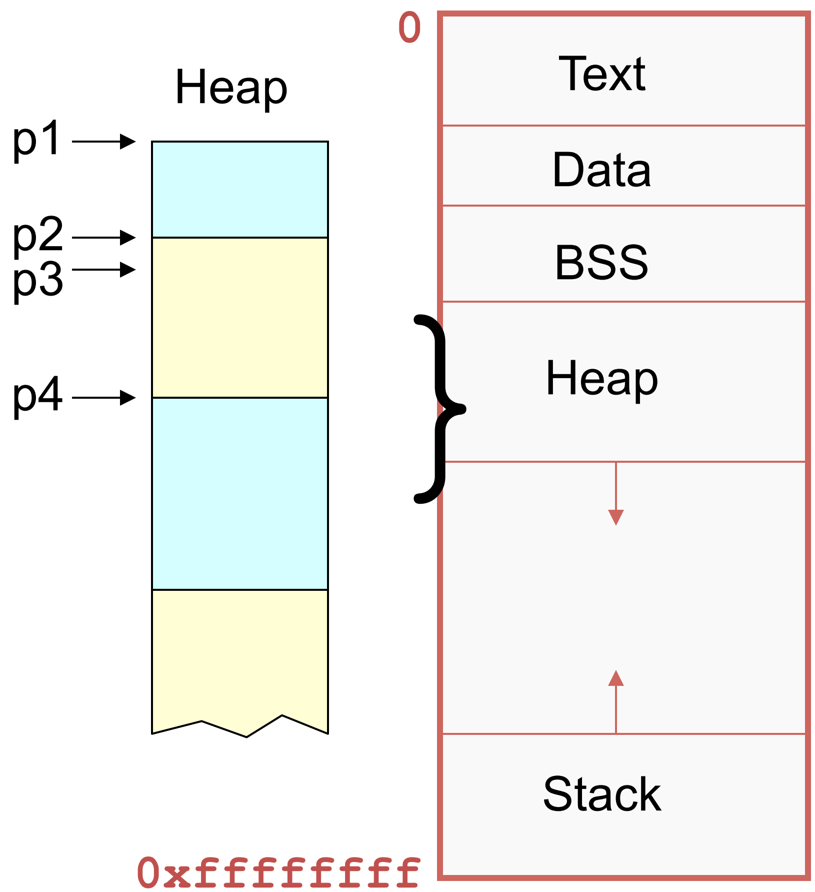
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List (C example)

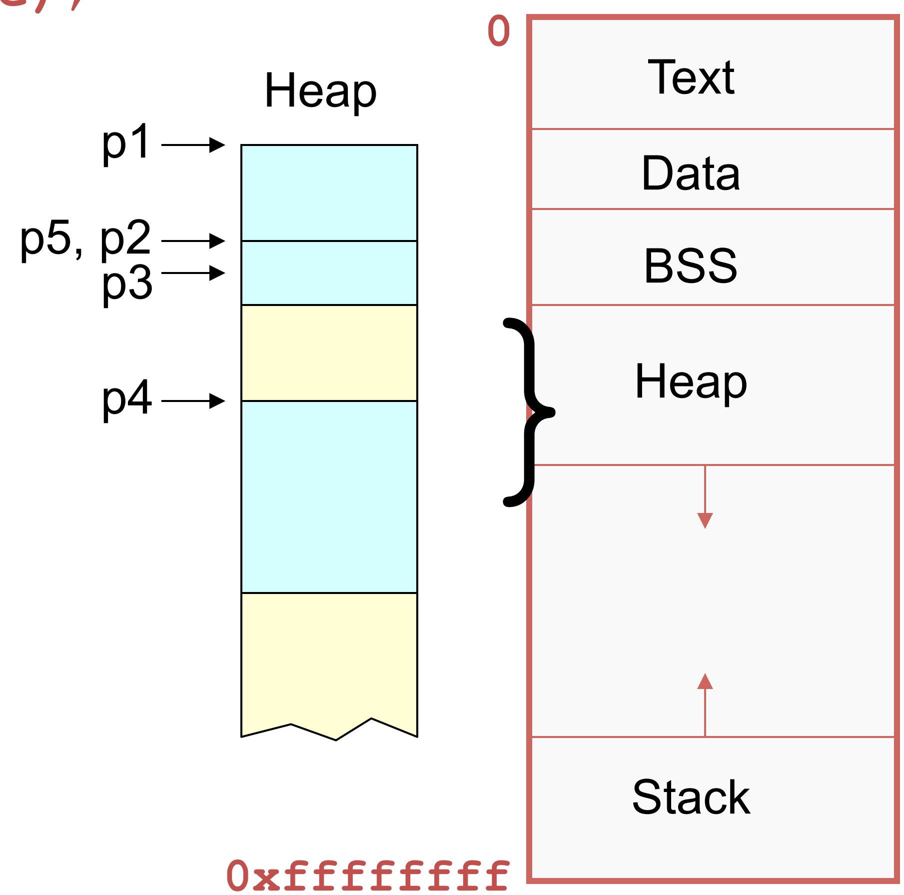
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List (C example)

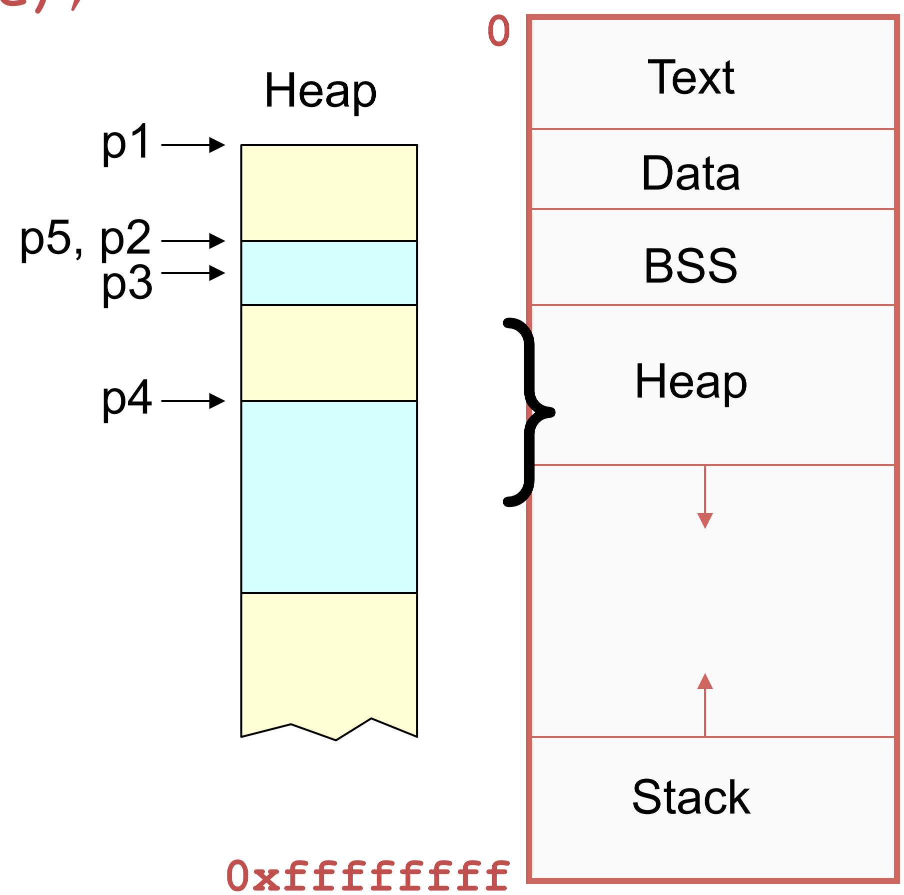
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



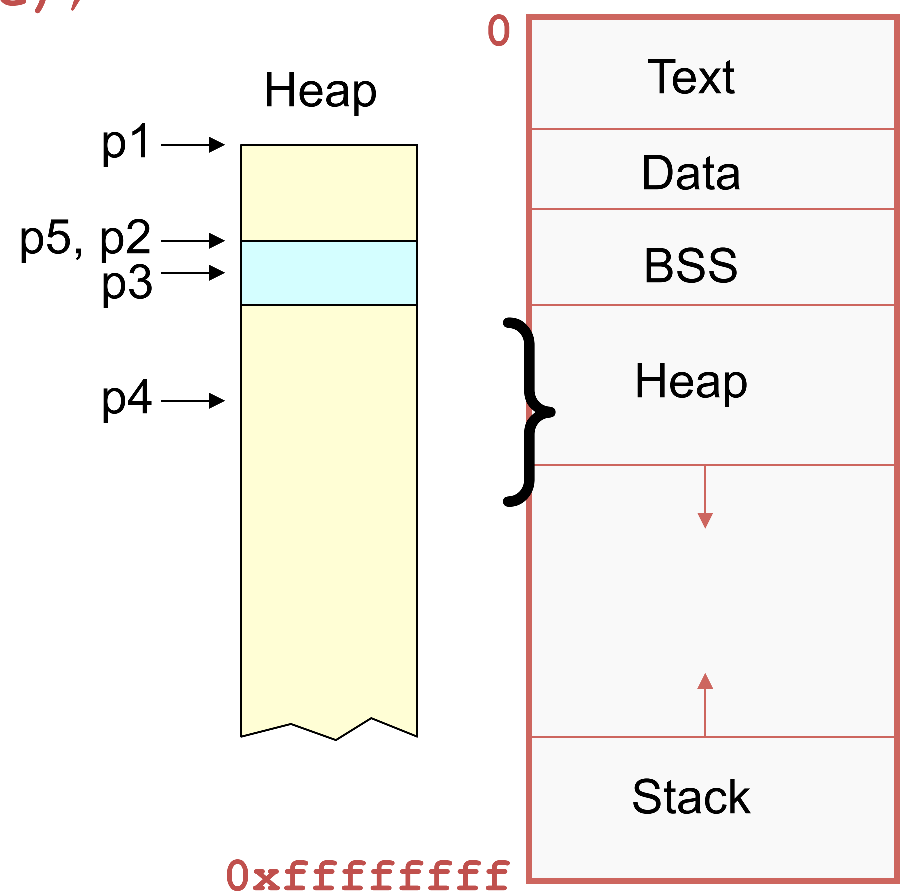
Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



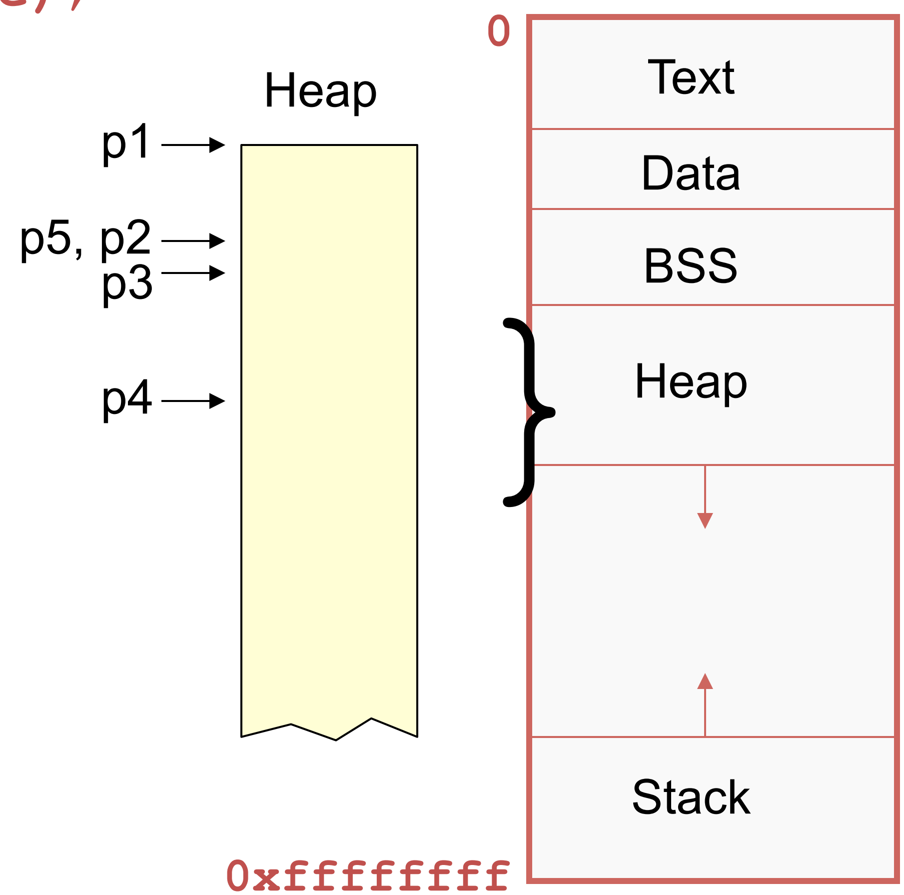
Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List (C example)

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List (C example): Text Description

- Originally free list just holds one big piece of open memory
- List of C commands and their consequences:
  - char *p1 = malloc(3)
    - 3 bytes are allocated on the heap
    - p1 holds address 100, the beginning of the 3 bytes
  - char *p2 = malloc(1)
    - 1 byte is allocated on the heap
    - p2 holds address 103 the beginning of the byte
  - char *p3 = malloc(4)
    - 4 bytes are allocated on the heap
    - p3 holds address 104 the beginning of the 4 bytes
  - free(p2)
    - deallocate the memory pointed to by p2
    - now there is a gap of free memory from addresses 103 to 104
      - note: since we are using a free list allocation scheme this memory can be reallocated later
      - free list now has 2 members on it
  - char *p4 = malloc(6)
    - look through free list for a piece of memory big enough
    - 6 bytes allocated on the heap
    - p4 holds address 108 the beginning of those 4 bytes
  - free(p3)
    - deallocate the memory pointed to by p3
    - now there is a gap of free memory from addresses 103 to 108.
      - the gap from freeing p2 is coalesced with the new gap made by freeing p3
  - char *p5 = malloc(2)
    - look through free list for a piece of memory big enough
      - see the gap from 103 to 108 and notice that it can hold 2 bytes
      - split the free memory and allocate p5 in part of it
    - 2 bytes allocated on the heap
    - p4 holds address 103 the beginning of those 2 bytes
    - now is a gap from 105 to 108
  - free(p1)
    - deallocate the memory pointed to by p1
    - now is a gap from addresses 100 to 103
      - free list now has 3 members on it
  - free(p4)
    - deallocate the memory pointed to by p4
    - gaps in memory are coalesced
      - now free list only has 2 members
        - a gap from 100 to 103
        - the rest of the free heap memory
  - free(p5)
    - deallocate memory pointed to by p5
    - now all of the original heap memory is free and can be reallocated as needed

What to Do When You Run Out of Heap

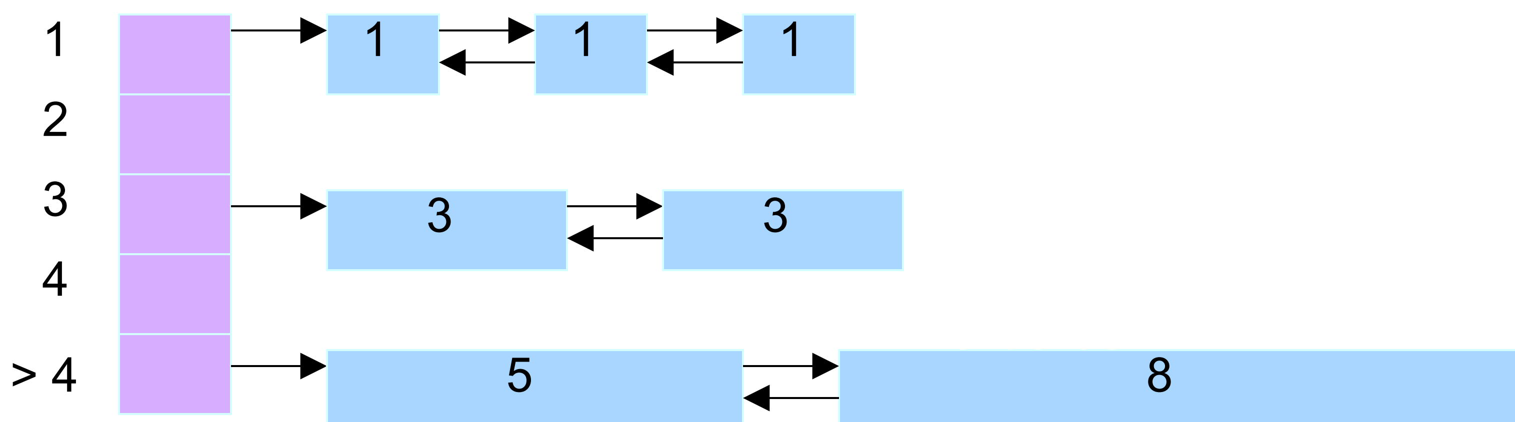
- Ask the operating system for additional memory
  - Ask for a very large chunk of memory
  - … and insert the new chunk into the free list
  - ... and then try again, this time successfully
- Operating-system dependent
  - E.g., **`sbrk`** command in UNIX

Performance

- What do we know about the performance of best-fit?
- Slow! Need to scan the free list.
- Trouble: Free chunks are different sizes
- Solution: Binning!
  - Divide list by chunk size

Binning Strategies: Exact Fit

- Have a bin for each chunk size, up to a limit
  - Advantages: no search for requests up to that size
  - Disadvantages: many bins, each storing a pointer
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller chunks, when needed

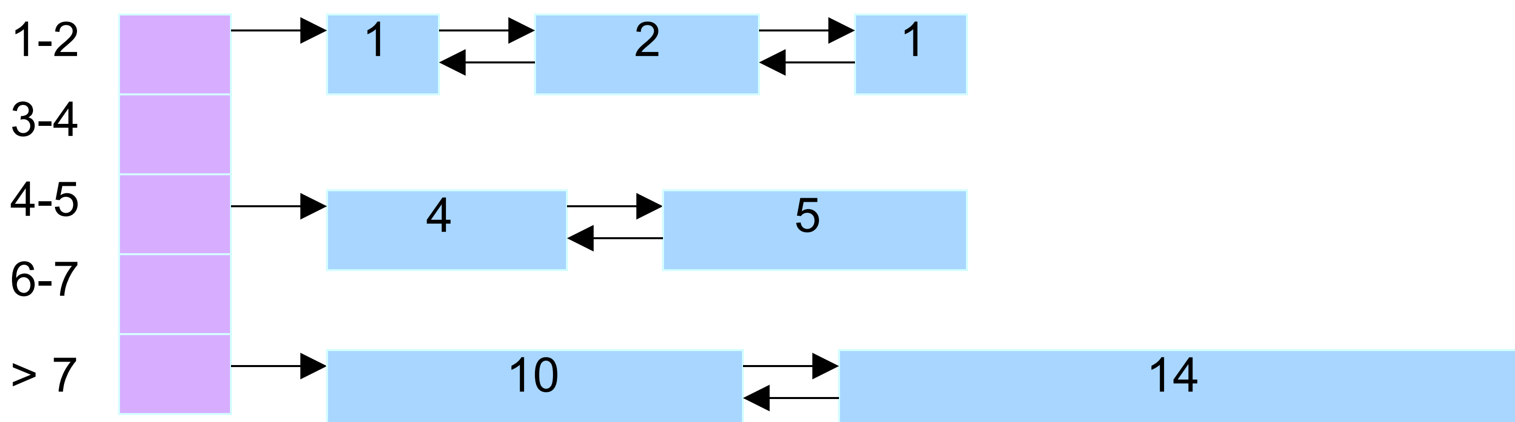


Binning Strategies: Exact Fit Text Description

- Bins are implemented with linked lists of open memory chunks
- Bin for size 1 has 3 chunks of open memory in it
- Bin for size 2 is empty
- Bin for size 3 has 2 chunks of open memory in it
- Bin for size 4 is empty
- Bin for sizes greater than 4 has 2 chunks of open memory in it
  - one chunk of size 5
  - another chunk of size 8
  - these can be taken and split up later if needed

Binning Strategies: Range

- Have a bin cover a range of sizes, up to a limit
  - Advantages: fewer bins
  - Disadvantages: need to search for a big enough chunk
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller chunks, when needed

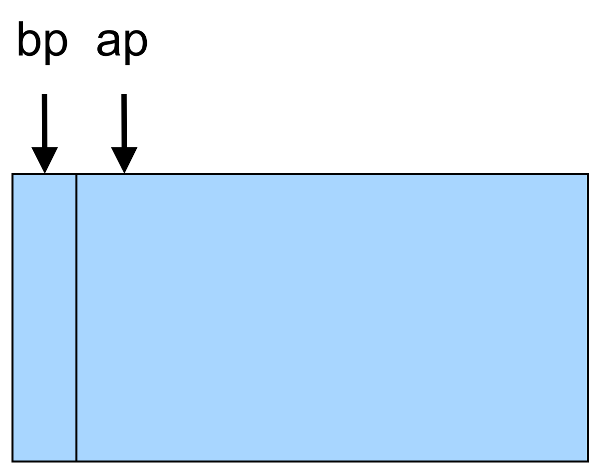


Binning Strategies: Range Text Description

- Bin for sizes 1 and 2 holds 3 chunks of open memory
  - 2 chunks of size 1 and 1 chunk of size 2
- Bin for sizes 3 and 4 is empty
- Bin for sizes 4 and 5 holds 2 chunks of open memory
  - 1 chunk of size 4 and the other of size 5
- Bin for sizes 6 and 7 is empty
- Bin for sizes greater than 7 holds 2 chunks of memory
  - One of size 10 and the other of size 14

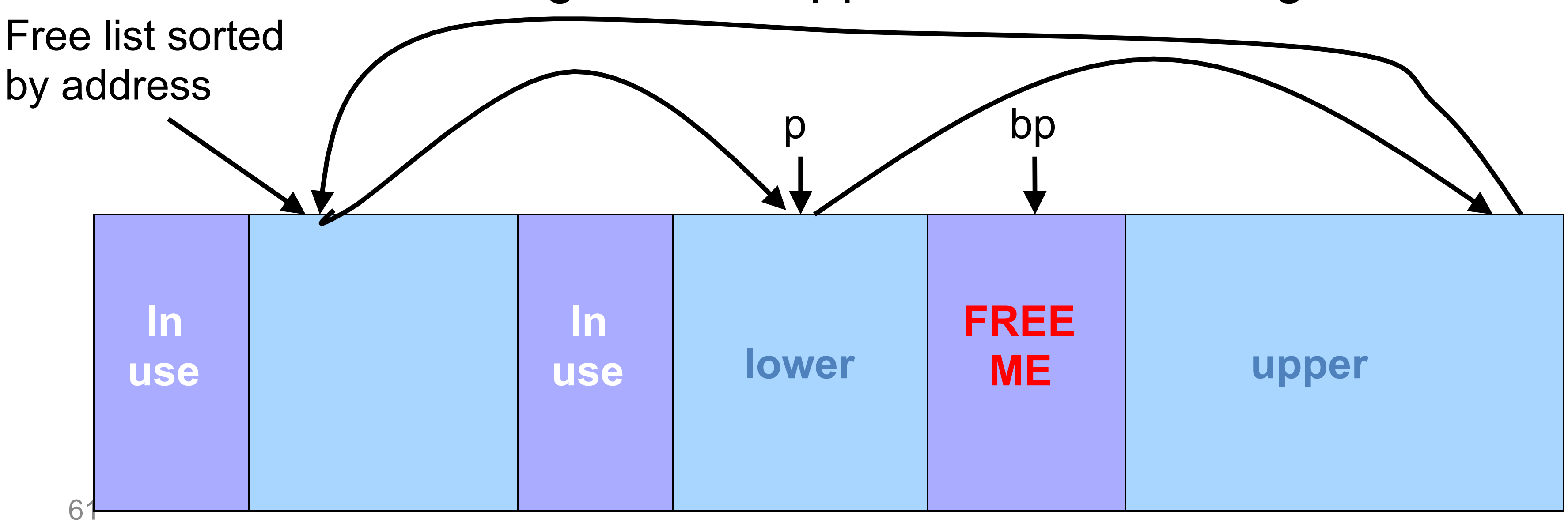
Deallocation with Free

- User passes a pointer to the memory block
- Free function inserts block into the list
  - Identify the start of entry
  - Find the location in the free list
  - Add to the list, coalescing entries, if needed



Coalescing With Neighbors

- Scanning the list finds the location for inserting
  - Pointer to to-be-freed element: **`bp`**
  - Pointer to previous element in free list: **`p`**
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors



Explicit Memory Management Challenges for the User

- More code to maintain
- Correctness
  - Free an object too soon -> core dump
  - Free an object too late -> waste space
  - Never free -> at best waste, at worst fail
- Efficiency can be very high
- Gives programmers control

iClicker Question

What advantage does bump pointer allocation have over free-list allocation?

A. No internal fragmentation
B. No external fragmentation
C. Memory re-use
D. Fast allocation

Summary

- Finished Virtual Memory
- Discussed explicit memory management
  - Allocation policies (bump pointer, free list)
  - De-allocation policies (free)
  - Free-list management

Announcements

- Homework 6 due Friday 8:45a
- Project 2 due Friday, 3/27
- Project 3 posted tonight

Have a good Spring Break! (Be safe.)