CS 475
Operating Systems

Department of Mathematics
and Computer Science

Lecture 9
Virtual Memory (Part II)

Brad and Adam Go to the Post Office

- Post Office keeps mailboxes for customers out in the front.

Fast, but limited (RAM)

- Mail that "spills over" go in the warehouse in the back end.

Slow, but large (Swap disk)

Brad and Adam Go to the Post Office

- The Post Office assigns 3 mailboxes each to Adam & Brad
  - Brad only uses one of his mailboxes as a secondary storage for ham.
  - Adam owns a t-shirt company called *"Purrrrfectly Adam"* that is surprisingly successful
    - **Needs 9 (virtual) PO Boxes**

Brad's Mapping

PO Box	V	Real Box
1	1	F
--	--	E
--	--	A

Adam's Mapping

PO Box	V	Real Box
1	1	B
2	1	D
3	0	
4	0	
5	1	C
6	0	
7	0	
8	0	
9	0	

Motivation: Memory Allocation

- If a process is given *too few* frames...
  - Could lead to many page faults during the runtime of a process
- If a process is given *too many* frames...
  - Frames could go unused
    - Wasteful
    - May lead to page faults in *other* processes
- Why does memory allocation matter?
  - Minimize page faults!
  - Avoid *thrashing:* process spends more time swapping than executing

Goals for This Lecture...

- Demand Paging
  - Motivation
  - Implementation
- Page Replacement Policies
- Memory Allocation
  - Equal
  - Proportional
  - Working Set Model
  - How Do malloc() and free() Work?

Equal Allocation (This is the P.O. Problem)

- Equal Allocation
  - Suppose we have m frames and n processes $(m > n)$
  - Give every process an equal share of m/n frames
  - Share of frames doesn't change over time

- Example: m = 93 frames, n = 10 processes
  - Each process gets 9 frames
  - Remaining 3 frames used as *page buffers*
- Pros: Easy, static algorithm. Fair.
- Cons?
  - Larger processes starve for frames (swap with disk more often)

Aside: Page Buffers

- *Page (Frame) Buffering is an OS optimization.*

- When a new page gets allocated and an "dirty page" gets replaced:
  - Without page buffering:
    - Write out old frame to disk
    - Give old frame to new page
  - With page buffering:
    - Select a new frame from page-buffer
    - Give to new frame while writing old page out to disk

Proportional Allocation

- *Proportional Allocation Policy*
  - Suppose we have m frames in memory and n processes
  - Continuously monitor memory usage for all processes
    - Let s_i denote the current virtual memory size of process P_i
    - Allocate a_i frames for P_i where $a_i = m\frac{s_i}{\sum_{i=1}^{n} s_i}$

- Pros:
  - Allows memory allocation to change over time
- Cons:
  - Unfair to existing processes when adding a new process
  - Existing processes all have to contribute some frames

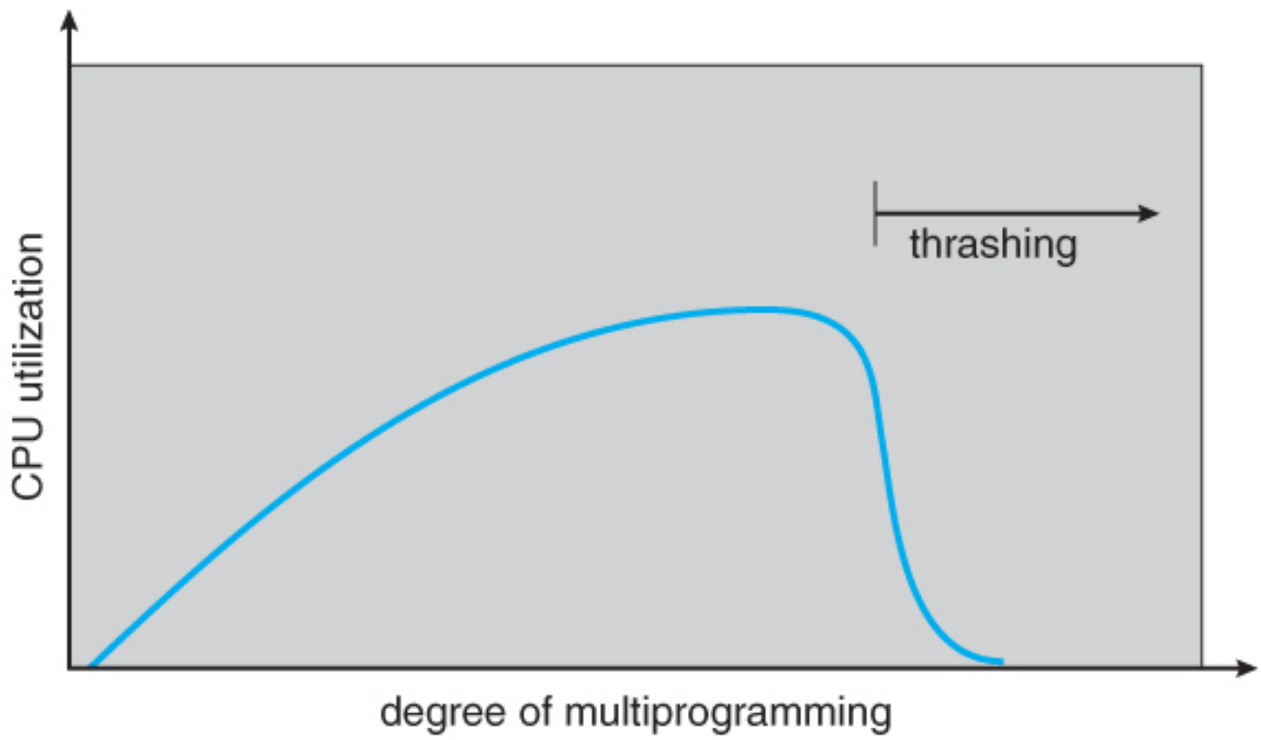
Important Side Note: "Thrashing"

- Page faults are expensive
  - Suppose a process wasn't allocated enough frames
  - During execution: page-in, page-out, page-in, page-out.
    - Not doing anything effective!

- We say that a process *"thrashes"* when it is spending more time paging than executing effective instructions.

Why Does Thrashing Occur?

- OS wants to keep CPU fully utilized
  - If CPU utilization % is low, OS dispatches another process to run
    - New processes → More memory usage
    - When memory runs out, we have increased page-fault rate
    - CPU utilization decreases due to page replacements
    - Causing OS to add even more processes… CPU utilization % eventually goes to 0



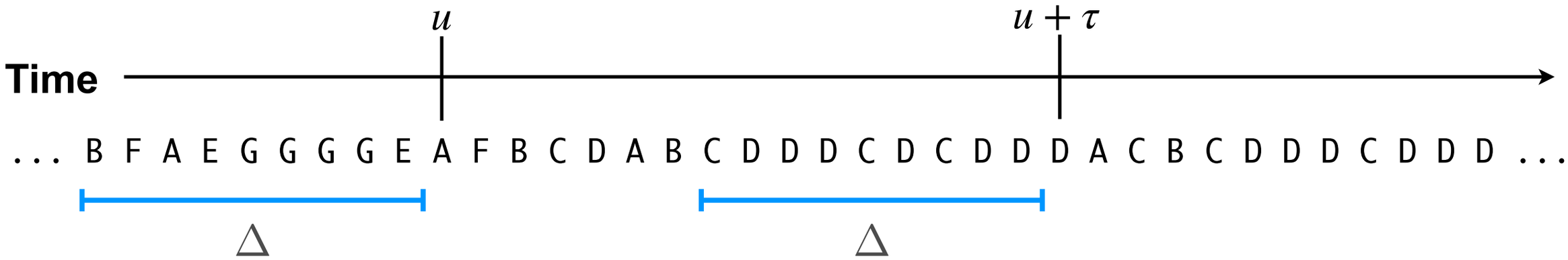
Working Set Model (1969, Multics)

- Process execution observes temporal and spatial locality
  - This implies there is a "working set" of pages that a process commonly uses at any time instant.

- To prevent thrashing we need to:
  - Track the *working set* across all processes over time
  - Try to give a process as many frames as it needs
  - Suspend a process if the system's *page demand* would exceed number of available frames

The "Working Set" Model

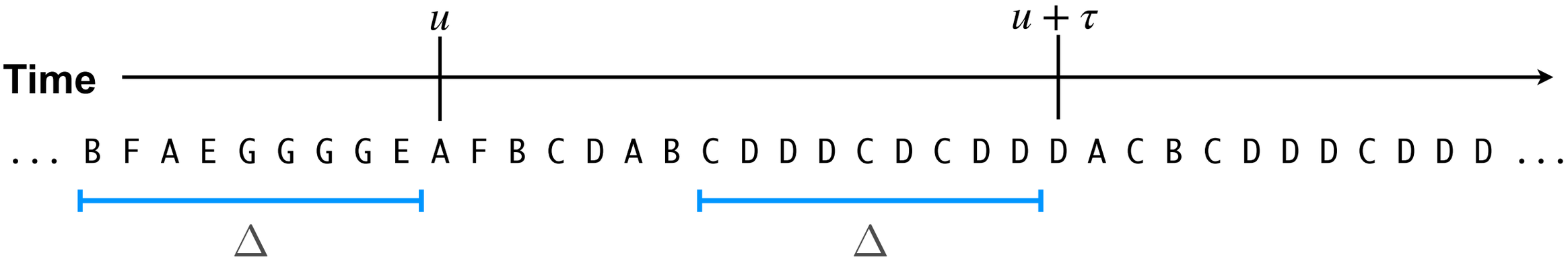
- For each process, track its page reference history
  - *Working Set:* the set of the Δ most recently-used pages
    - Δ is the size of the "sliding window"
    - τ (tau) is the time interval between checks
  - Consider the following page references across time:



- What's the working set at time u? At time $u + \tau$?

The "Working Set" Model (2)

- Example: Consider the following page references



- Working sets for process P_i

$$WS_i(u) = \{B, A, F, E, G\}$$

$$WS_i(u + \tau) = \{C, D\}$$

- *(What do they tell us?)*

Working Set Model (Cont.)

- If $WS_i(t)$ denotes the *working set* of process P_i at time t, then the OS's total demand for pages at time t is:

$$pageDemand(t) = \sum_{i=1}^{n} |WS_i(t)|$$

- Suppose our system supports m frames of memory, thrashing occurs when $pageDemand(t) > m$.

Working-Set Model (Cont.)

- *Working-Set Allocation Policy*
  - Select a value for Δ, the sliding window size
  - Monitor working set $WS_i(t)$ of all existing processes
  - Allocate $|WS_i(t)|$ frames for process P_i
    - If $pageDemand(t) > m$ // *OS will thrash!*
      - Choose a process to suspend (swap it out to reduce degree of multiprogramming)
      - Redistribute its frames to other processes

Working Set: Sliding Window Size

- The sliding window size Δ is an important parameter.

- Too Large:
  - Captures too many references
  - May overestimate page demand
    - Causes processes to swap out when they do not need to → low CPU utilization

- Too Small:
  - Could underestimate page demand
    - Causes OS to aggressively dispatch processes to run → thrashing

Goals for This Lecture...

- Demand Paging
  - Motivation
  - Implementation
- Page Replacement Policies
- Memory Allocation
  - Equal
  - Proportional
  - Working Set Model
  - How Do malloc() and free() Work?

How Does C Implement `malloc()` and `free()`

- Recall C's dynamic (de)allocation we can't live without!

```c
/**
 * Attempts to allocate size bytes on the heap and returns pointer to
 * first byte
 */
void* malloc(size_t size);
```

`@param size`: the number of bytes to allocate on the heap.

`@return` pointer to the first byte if successful, otherwise `NULL`

```c
/**
 * Frees the memory pointed to by ptr
 */
void free(void* ptr);
```

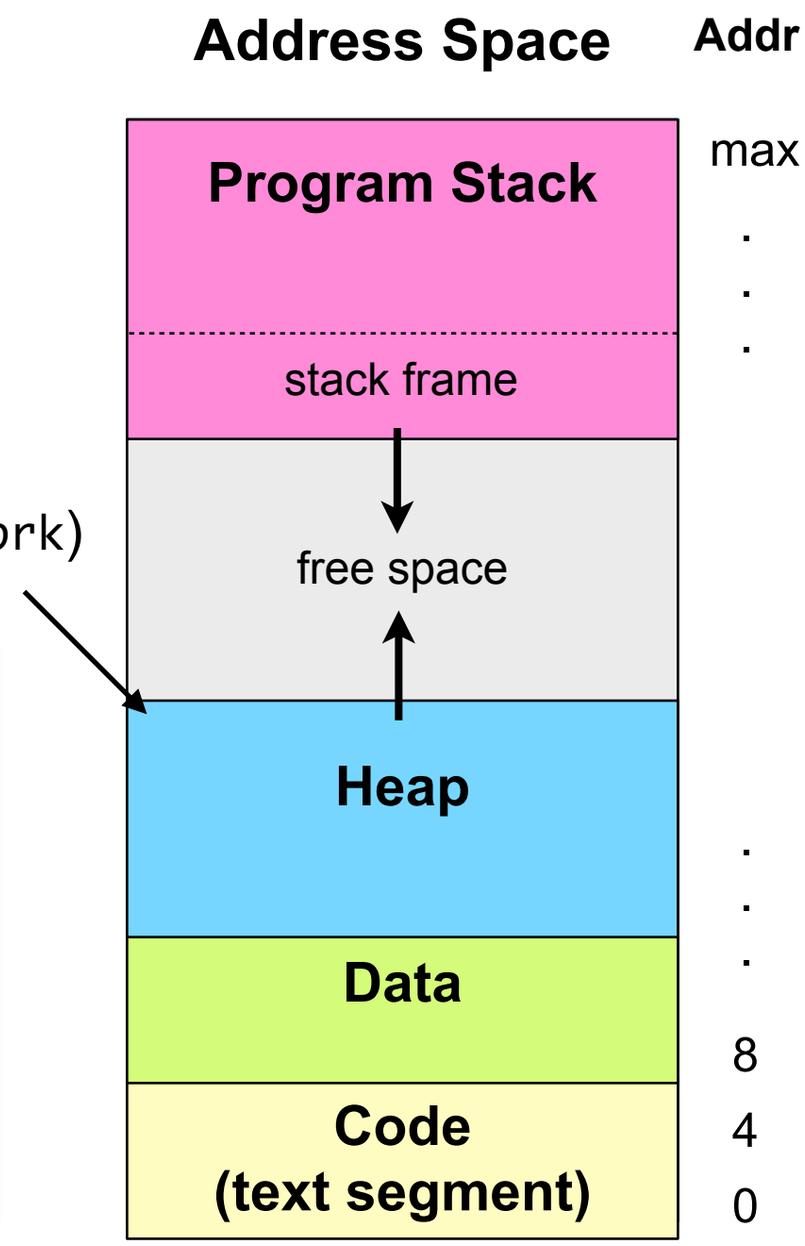
`@param ptr`: the pointer to the allocated memory.

A Review: Process' Address Space

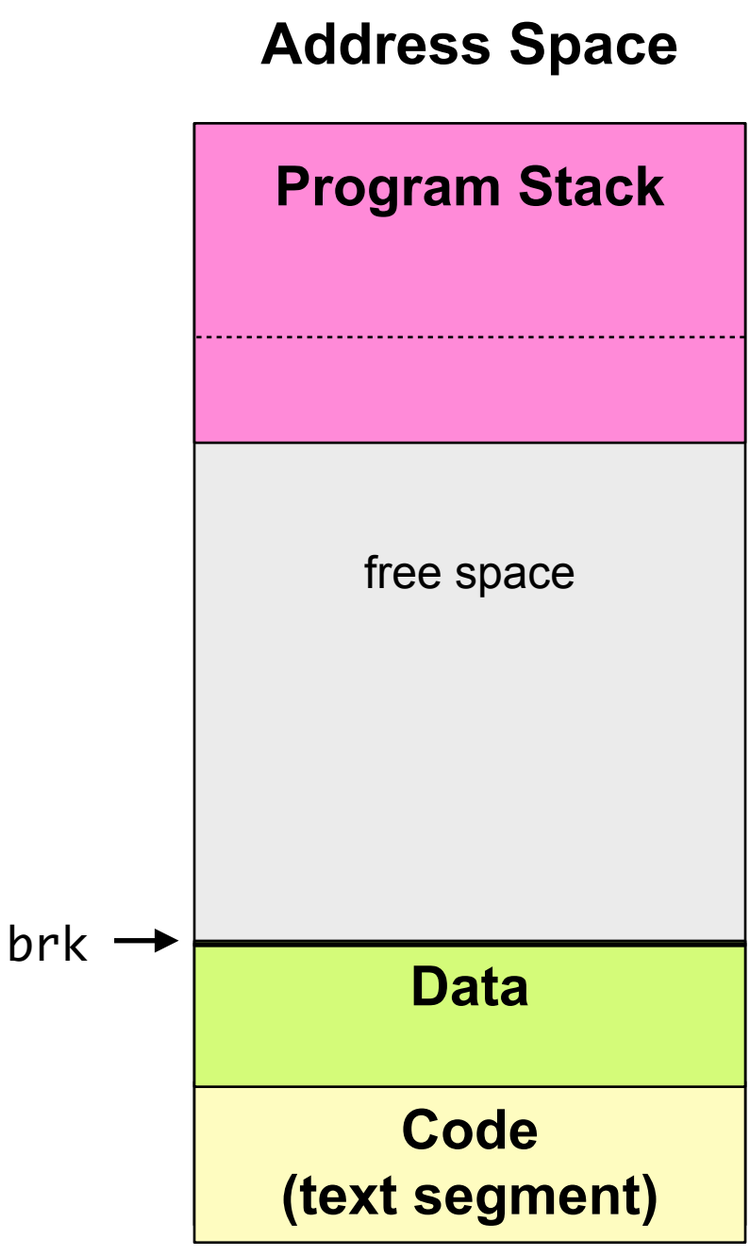
- For each process, its PCB stores a pointer called the *program break* that points to top of the heap
  - How do we move it?

- malloc() calls this system call:

```
#include <stdlib.h>

/**
 * A system call to ask the kernel for numbytes bytes to
 * be allocated to the calling process. Moves brk
 * pointer accordingly.
 */
void *sbrk(intptr_t numbytes);
```

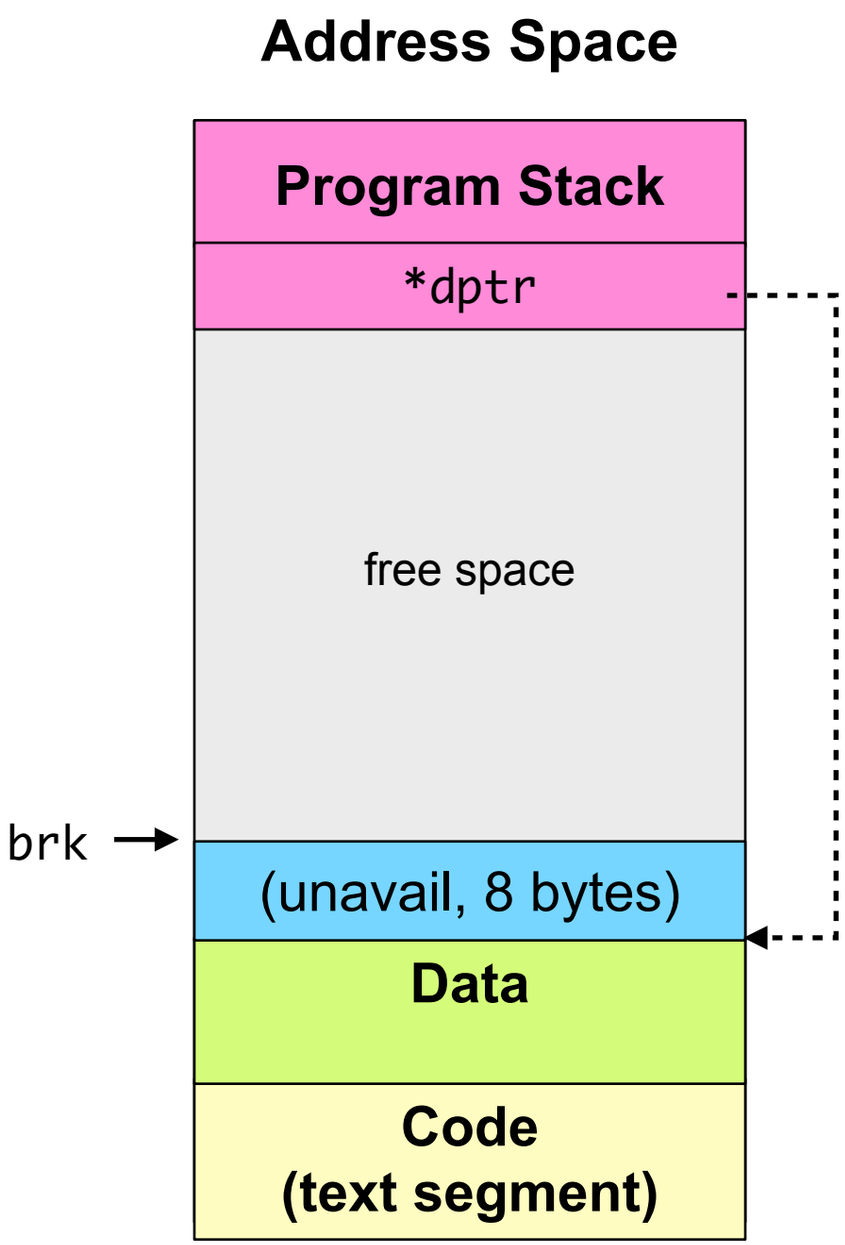


Step 1: Process Starts, No Heap



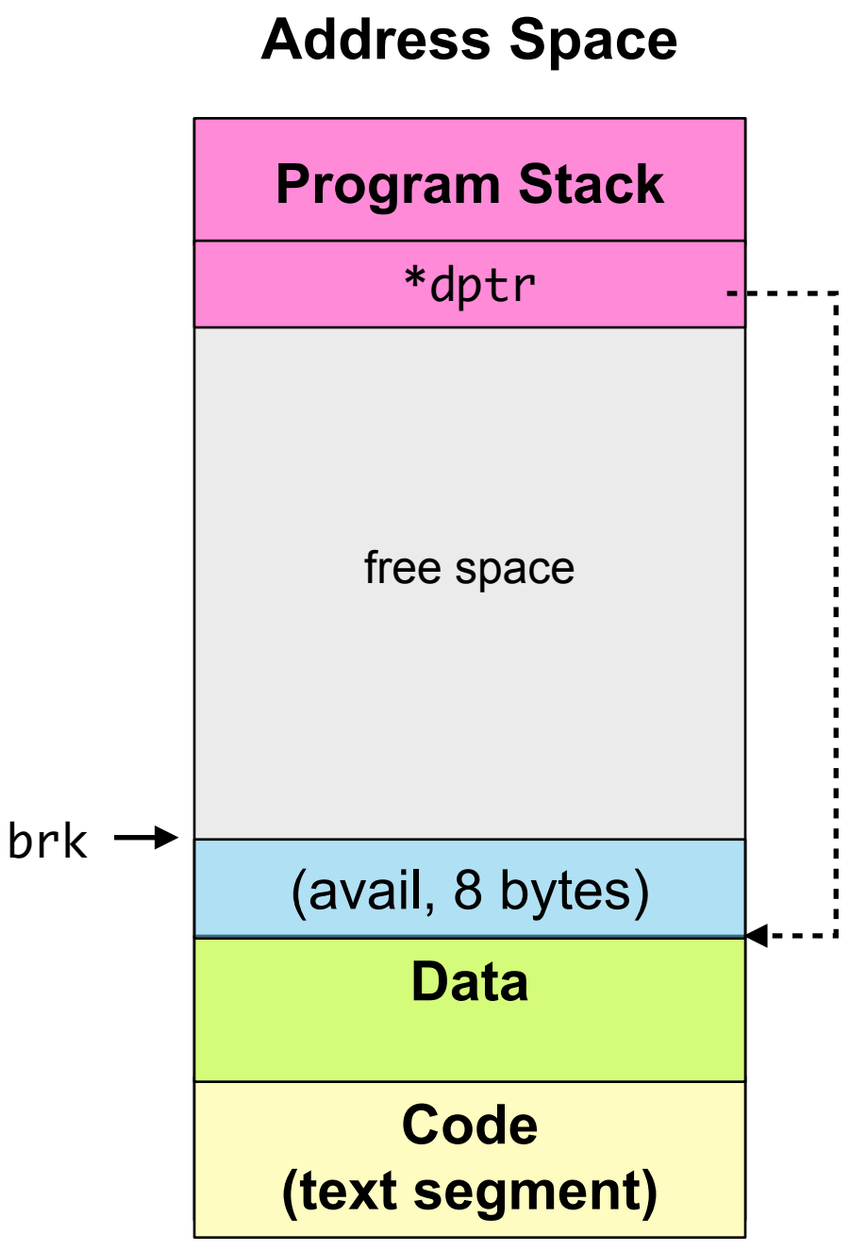

```
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Step 2: Allocate 8 bytes on heap



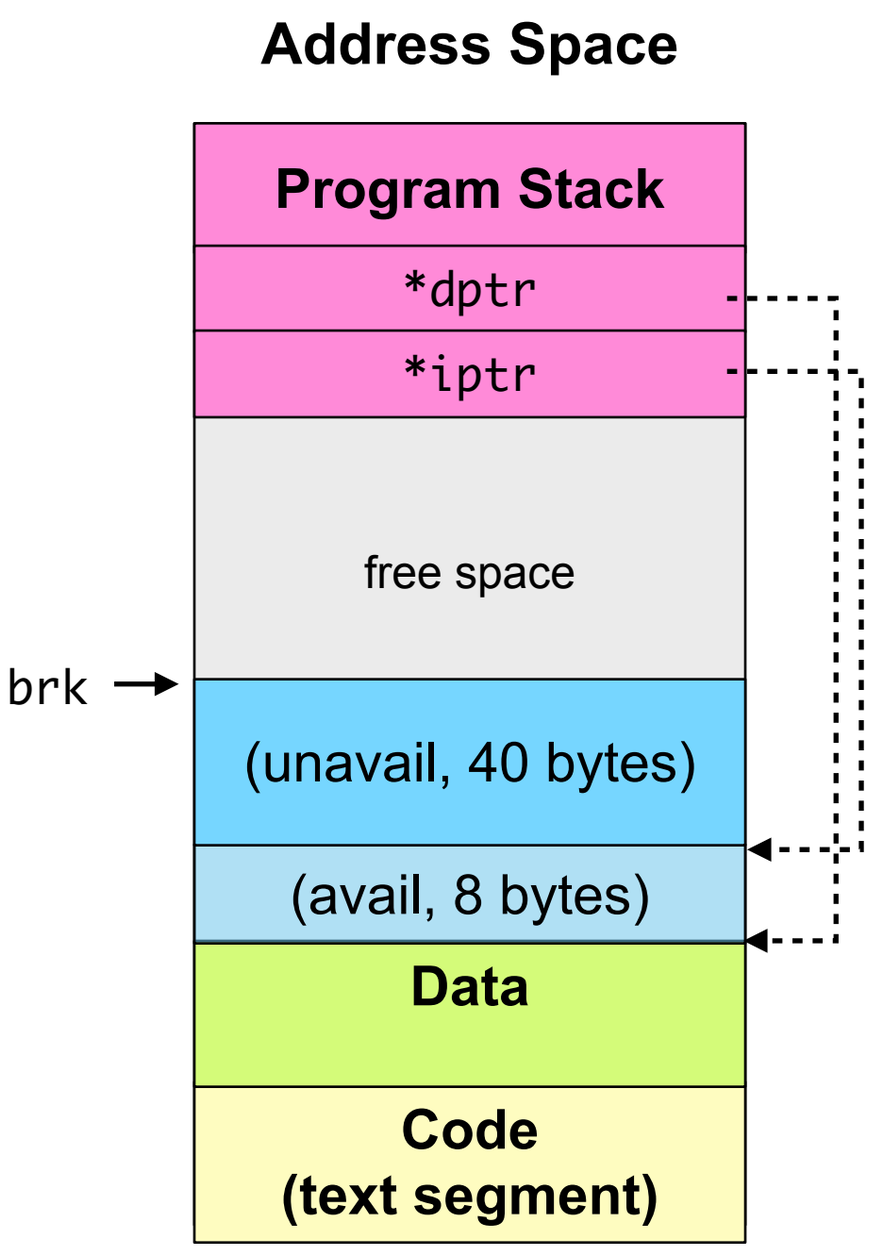

```c
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Step 3: Freeing the Space



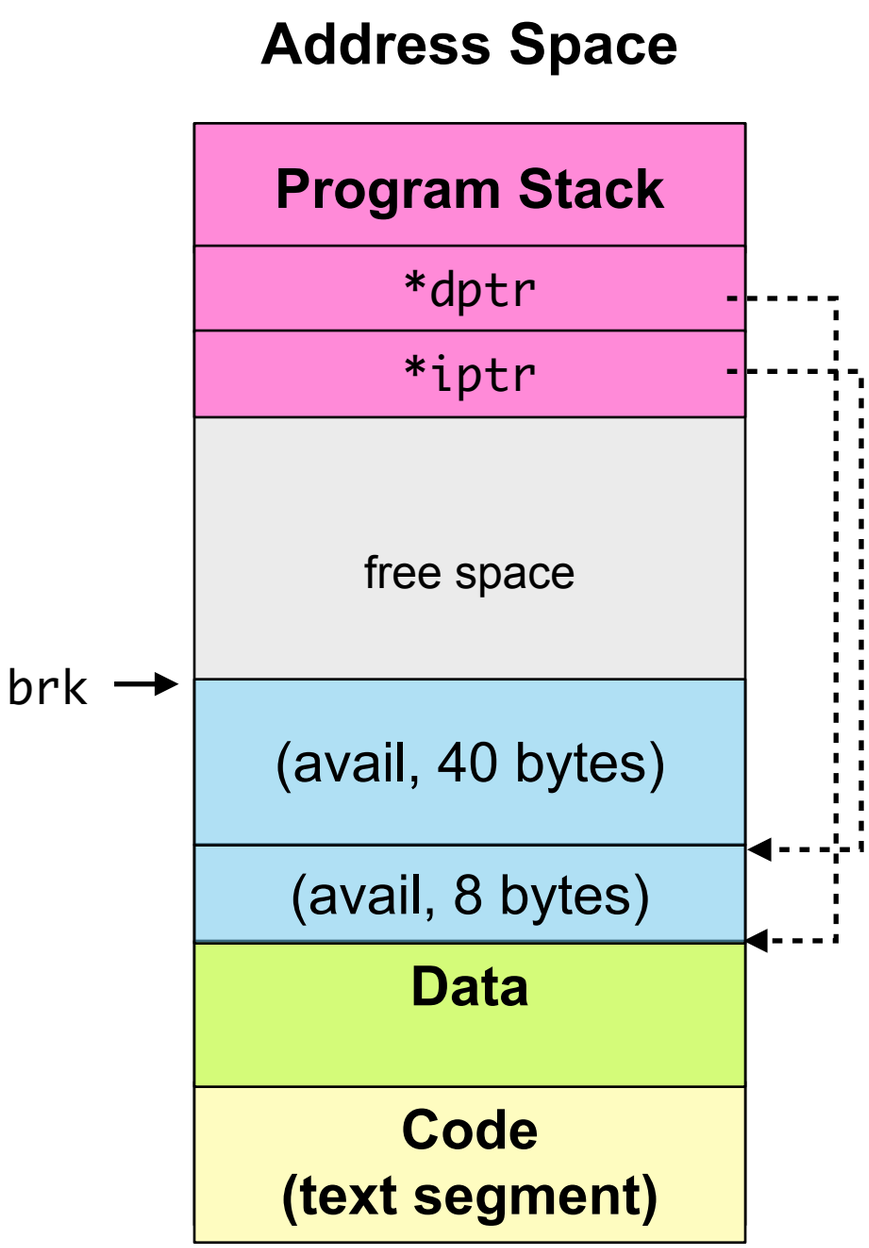

```c
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Step 4: Process Starts, No Heap



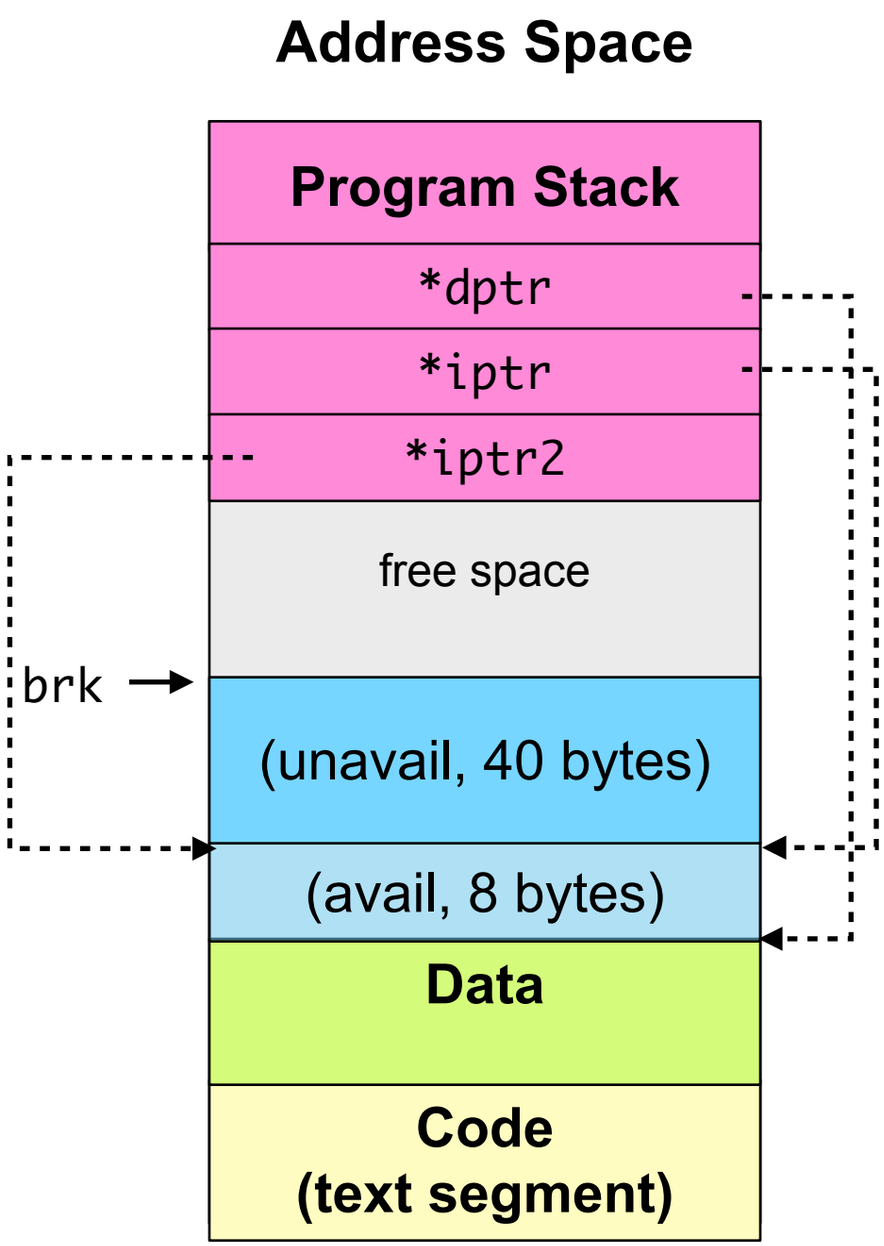

```c
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Step 4: Process Starts, No Heap



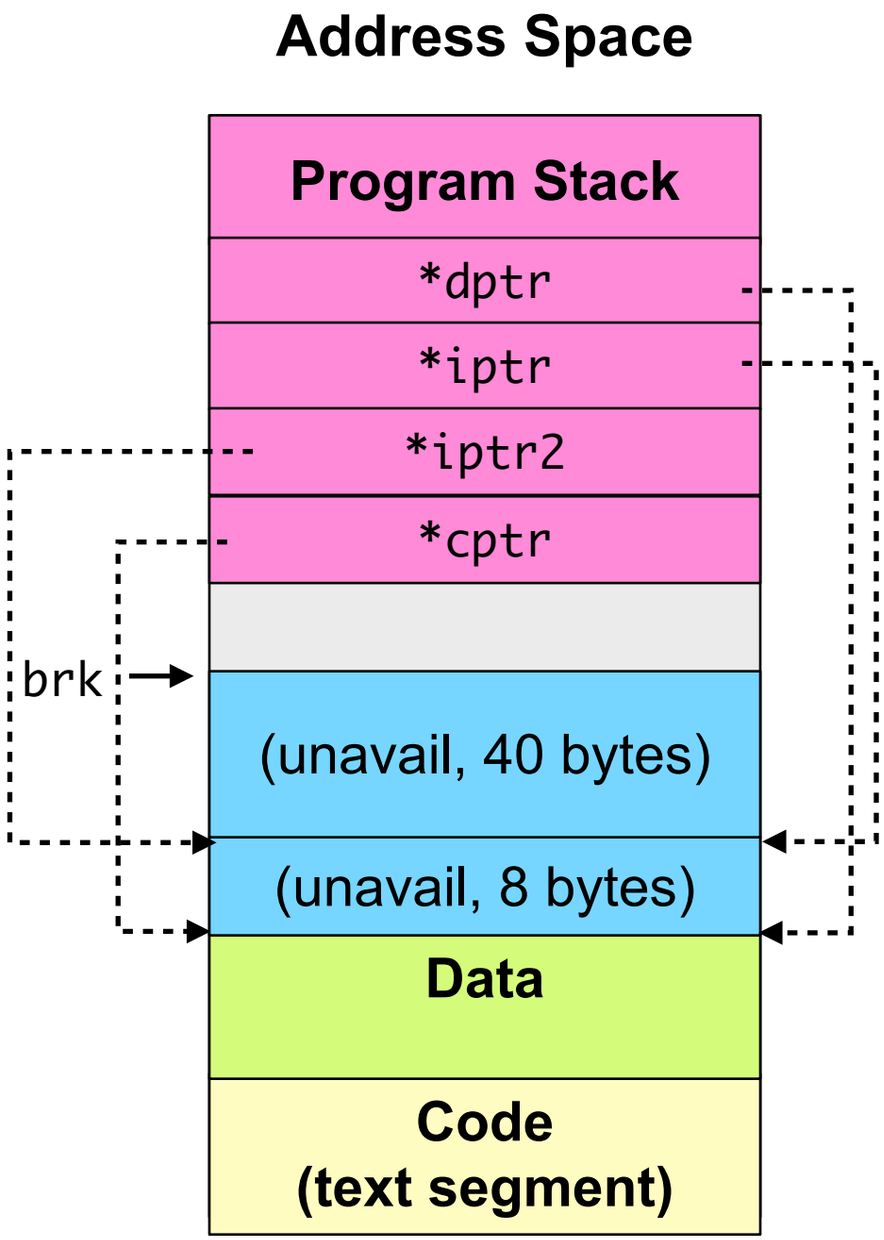

```c
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Step 4: Process Starts, No Heap




```c
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Step 4: Process Starts, No Heap



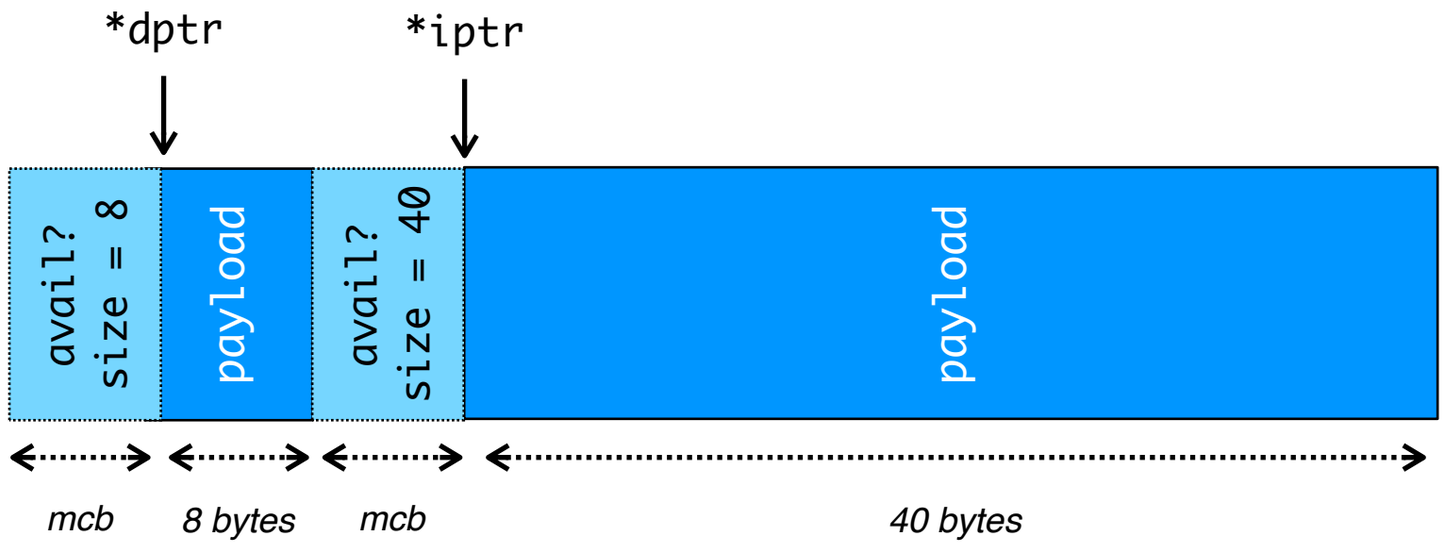

```c
int main() {
  double *dptr = (double*) malloc(sizeof(double));
  //...
  free(dptr);

  int *iptr = (int*) malloc(10*sizeof(int));
  //...
  free(iptr);

  int *iptr2 = (int*) malloc(5*sizeof(int));

  char *cptr = (char*) malloc(6*sizeof(char));
}
```

Let's Focus on the Heap

- How is the *heap* segment managed?
  - Just a list of "chunks": each is the size `malloc()` requested
- A chunk has a header, called the *memory control block (mcb):*

```
struct mem_control_blk {
   unsigned int avail;  //is this chunk available?
   size_t size;         //size of this chunk
};
```

- Heap content:



Example Showing Reuse of Freed Memory

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

int main() {
    double *a = (double*) malloc(sizeof(double));
    *a = 123.4;
    printf("a (before free) = %0.6f\n", *a);
    free(a);

    double *b = (double*) malloc(sizeof(double));
    *b = 5.678;
    printf("b = %0.6f\n", *b);
    printf("a (after free) = %0.6f\n", *a);  // What's printed?

    return 0;
}
```

Example Showing Reuse of Freed Memory

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

int main() {
    double *a = (double*) malloc(sizeof(double));
    *a = 123.4;
    printf("a (before free) = %0.6f\n", *a);
    free(a);

    double *b = (double*) malloc(sizeof(double));
    *b = 5.678;
    printf("b = %0.6f\n", *b);
    printf("a (after free) = %0.6f\n", *a);  // What's printed?

    return 0;
}
```

```
Program Output
----------------------------------
a (before free) = 123.400000
b = 5.678000
a (after free) = 5.678000
```

Importance of setting "loose pointers" to NULL after freeing!

free()

- Things to note:
  - Accessing the MCB
  - No changes to heap contents

```
void free(void *ptr) {
    // remember to move pointer in reverse to account for the MCB header
    struct mem_control_blk *mcb = (struct mem_control_blk*)
                                        (ptr - sizeof(struct mem_control_blk));
    mcb->avail = 1;
}
```

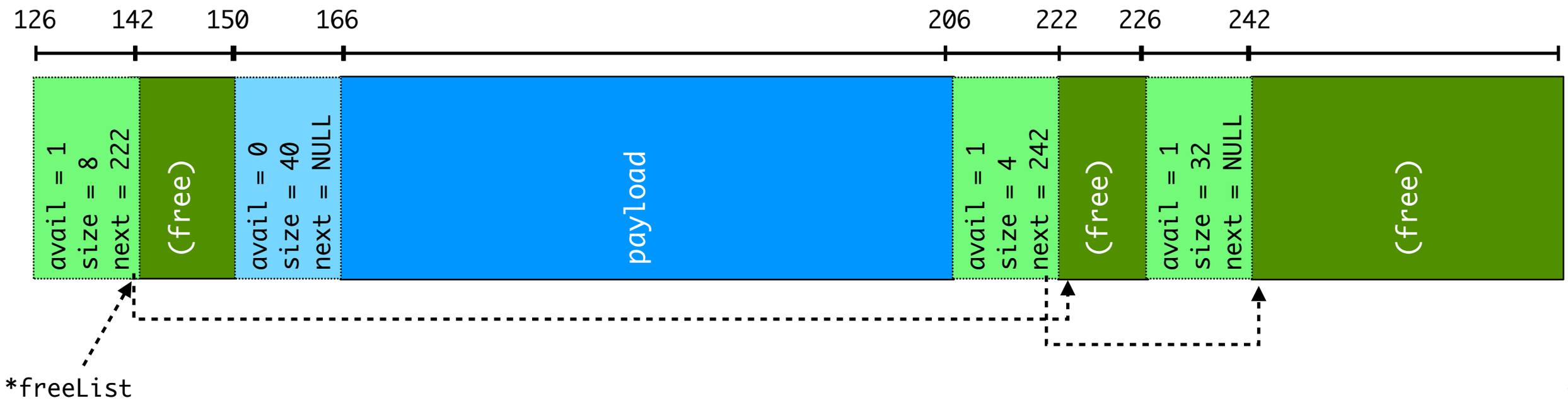
Naive malloc()

```c
void* malloc(size_t size) {
    void* current_addr = start_addr; // get start addr of heap
    void* end_addr = sbrk(0);        // get end addr of heap (top of heap)

    while (current_addr < end_addr) {
        //get the MCB of current chunk
        struct mem_control_blk *mcb = (struct mem_control_blk*) (current_addr);
        if (mcb->avail && size <= mcb->size) {
            //found a freed chunk! Use it!
            mcb->avail = 0;
            return current_addr + sizeof(struct mem_control_blk);
        }

        //skip to next chunk. Need to account for header
        current_addr + mcb->size + sizeof(struct mem_control_blk);
    }

    //must mean there were no freed chunks that are available or large enough
    current_addr = sbrk(size);   //request the chunk from kernel
    struct mem_control_blk *mcb = (struct mem_control_blk*) (current_addr);
    mcb->avail = 0;
    mcb->size = size;
    return current_addr + sizeof(struct mem_control_blk);
}
```

What Made It Naive?

- May have to traverse entire list of chunks
  - Better: use a *free list!*

```
struct mem_control_blk {
   unsigned int avail;  // is this chunk available?
   size_t size;         // size of this chunk
   void*  next;         // <-- ADDED THIS: pointer to the next free chunk
};
```



Goals for This Lecture...

- Demand Paging
  - Motivation
  - Implementation
- Page Replacement Policies
- Memory Allocation
- Conclusion

In Conclusion...

- Virtual Memory: Allows processes to operate when their virtual address space is LARGER than physical address space!

- Demand paging treats RAM as a cache for pages
  - Program stored on disk to begin with
  - Swap pages in and out on as-needed basis
  - Page fault => Heavy penalty (disk access)

In Conclusion... (Cont.)

- Good replacement policies:
  - Minimize page faults
  - Are fast to make decisions on which page to toss out
    - There are many loads and stores in a program
    - That's why LRU is good, but impractical
    - Also why RAND is actually used in practice!