Operating Systems 412

Pete Keleher

(slides mostly from Youjip Won)

Memory

- 14 - Memory API
- 17 - Free Space Management
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Memory API: malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap.
  - Argument
    - `size_t size` : size of the memory block(in bytes)
    - `size_t` is an unsigned integer type.
  - Return
    - Success : a void type pointer to the memory block allocated by `malloc`
    - Fail : a null pointer

sizeof()

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly.
- Careful of `sizeof`!

```
int *x = malloc(10 * sizeof(int));
printf(“%d\n”, sizeof(x));
```

```
4
```

```
int x[10];
printf(“%d\n”, sizeof(x));
```

```
40
```

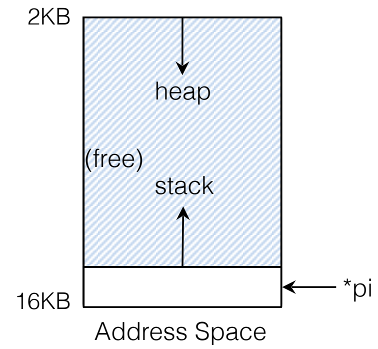
Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

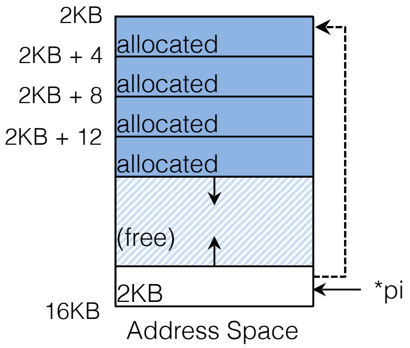
- Free memory region allocated by a call to `malloc`.
    - Argument
        - `void *ptr` : a pointer to a memory block allocated with `malloc`
    - Return
        - none

Memory Allocating

⇢ pointer

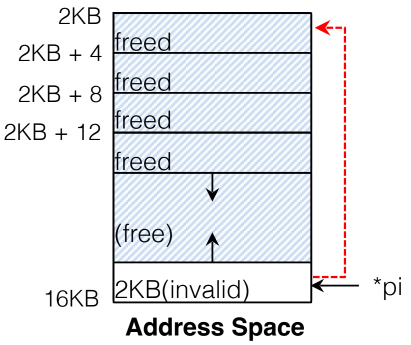


```
int *pi; // local variable
```

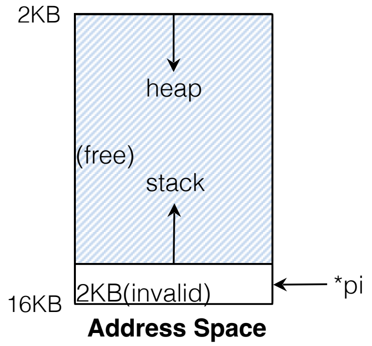


```
pi = (int *)malloc(sizeof(int)* 4);
```

Freeing Memory



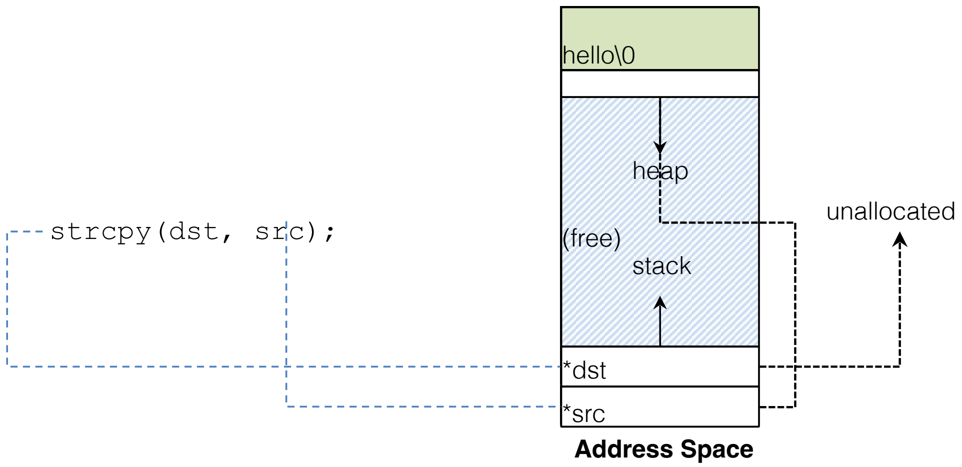
```
free(pi);
```



Forgetting To Allocate Memory

- Incorrect code

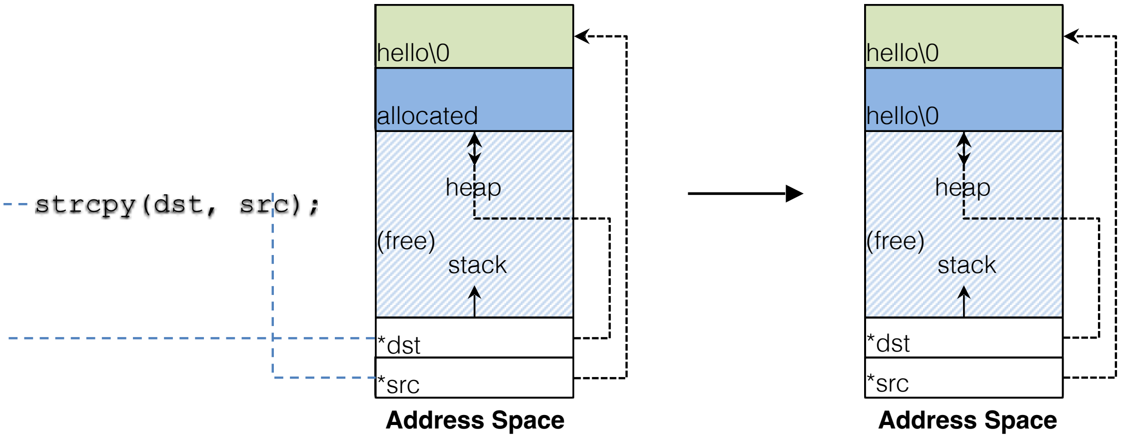
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory(Cont.)

- Correct code

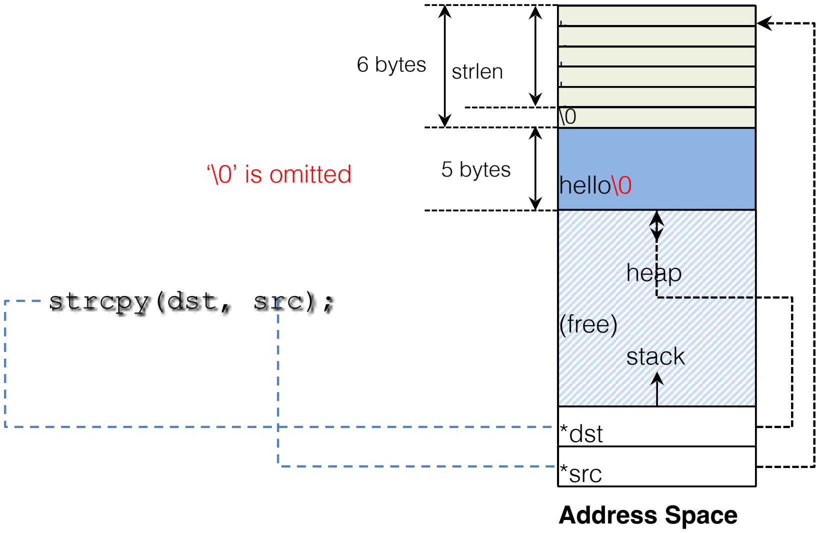
```
char *src = “hello”;   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but sometimes works

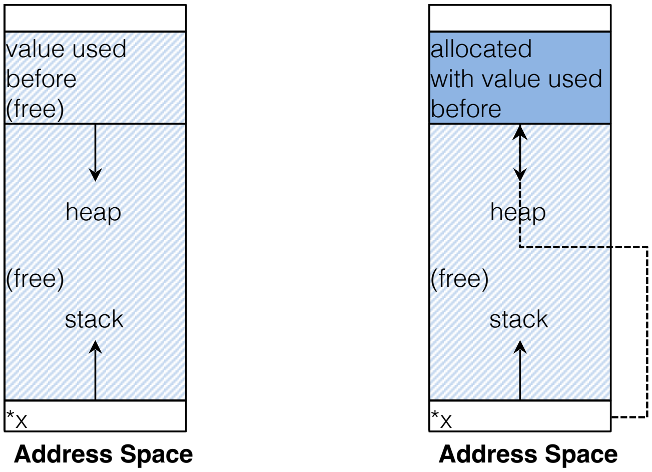
```
char *src = “hello”; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);     //work properly
```



Forgetting to Initialize

- Encounter an uninitialized read

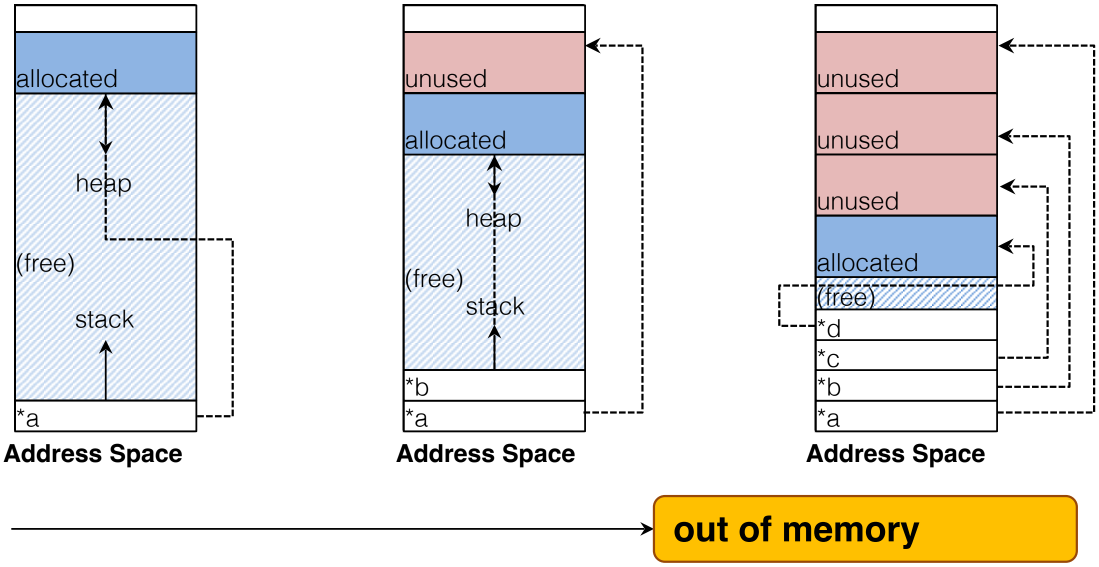
```
int *x = (int *)malloc(sizeof(int)); // allocated
printf(“*x = %d\n”, *x); // uninitialized memory access
```



Memory Leak

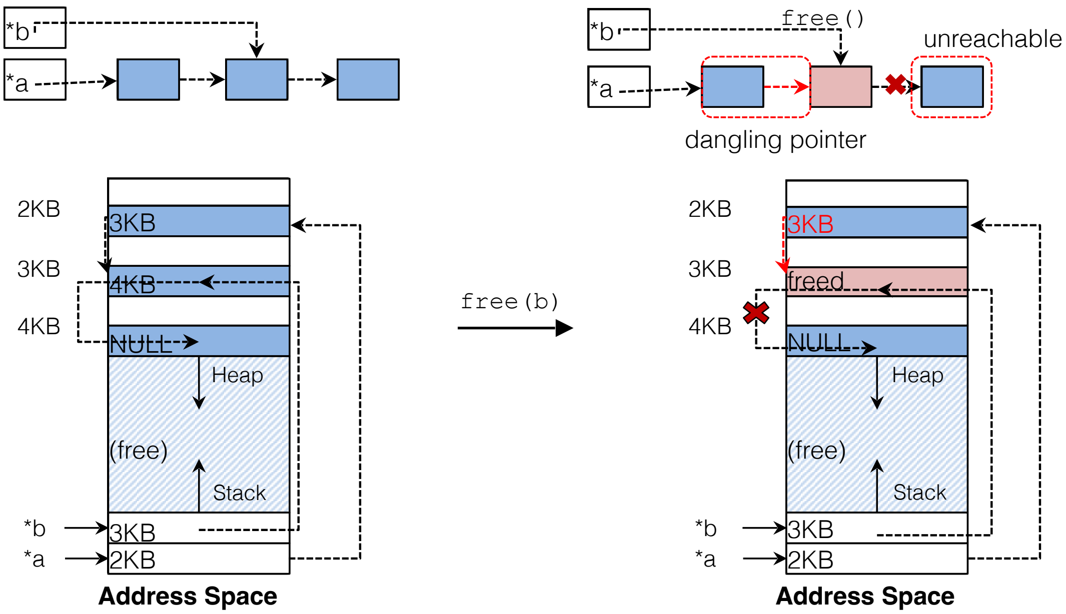
- A program runs out of memory and eventually dies.

unused : unused, but not freed



Dangling Pointer

- Use after memory freed: invalid pointer



Other Memory APIs: calloc()

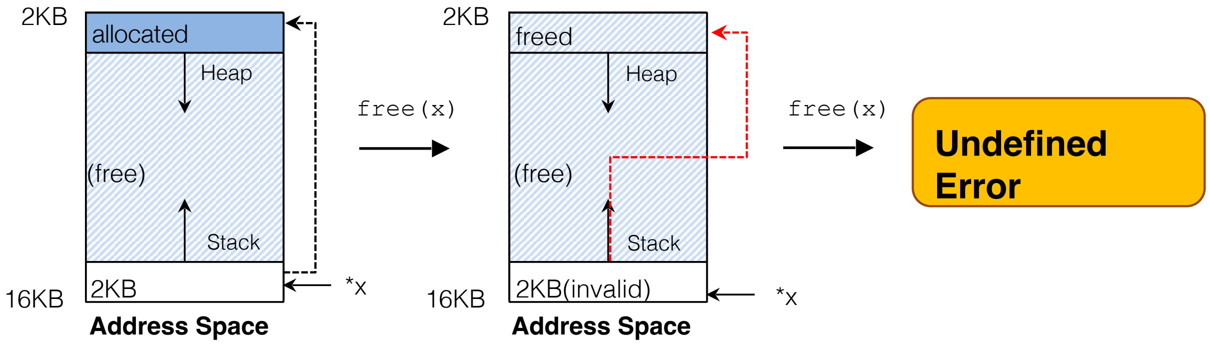
```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate memory on the heap and zeroes it before returning.
  - Argument
    - `size_t num` : number of blocks to allocate
    - `size_t size` : size of each block(in bytes)
  - Return
    - Success : a void type pointer to the memory block allocated by `calloc`
    - Fail : a null pointer

Double Free

- Free memory that was freed already.

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```



Other Memory APIs: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Change the size of memory block.
  - A pointer returned by `realloc` may be either the same as `ptr` or a new.
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc, calloc` or `realloc`
    - `size_t size`: New size for the memory block(in bytes)
  - Return
    - Success: Void type pointer to the memory block
    - Fail : Null pointer

System Calls

```
#include <unistd.h>

int   brk(void *addr)
void  *sbrk(intptr_t increment);
```

- malloc library uses the brk system call
  - brk is called to expand the program's break.
    - break: The location of the end of the heap in address space
  - sbrk is an additional call similar with brk.
  - Programmers should never directly call either brk or sbrk.

System Calls(Cont.)

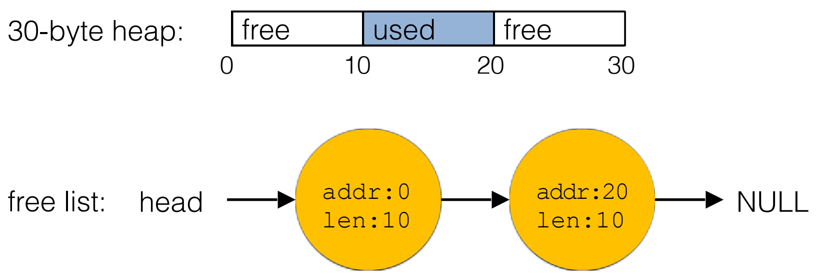
```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags, int
fd, off_t offset)
```

Memory

- 14 - Memory API
- 17 - Free Space Management
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

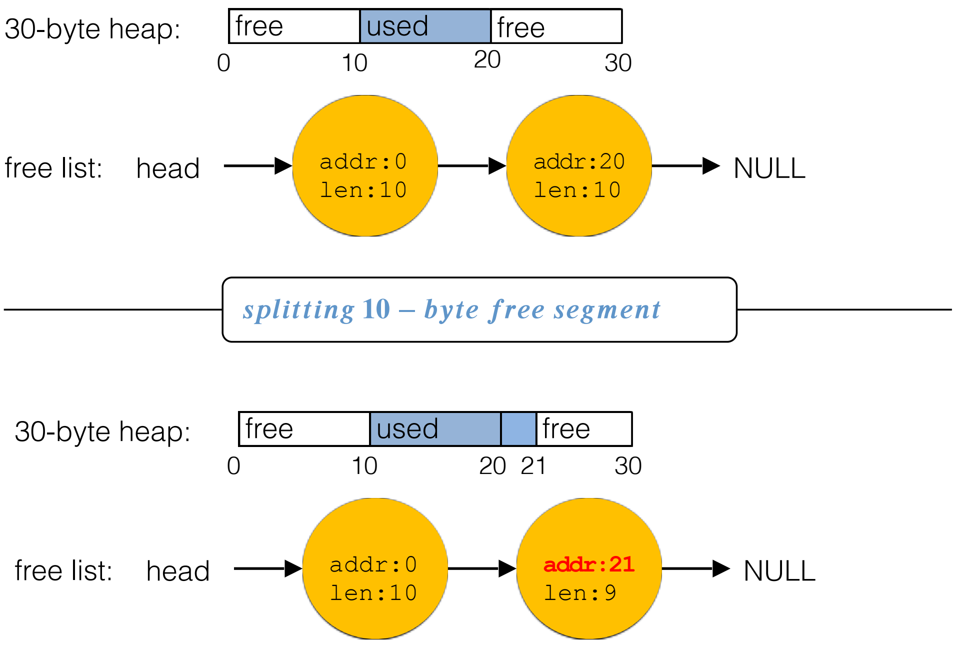
Splitting

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



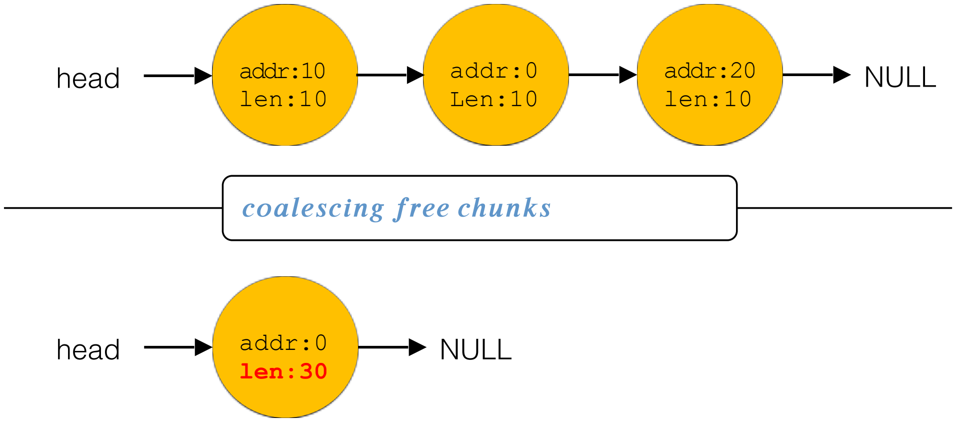
Splitting(Cont.)

- Two 10-byte free segments with a **1-byte request**



Coalescing

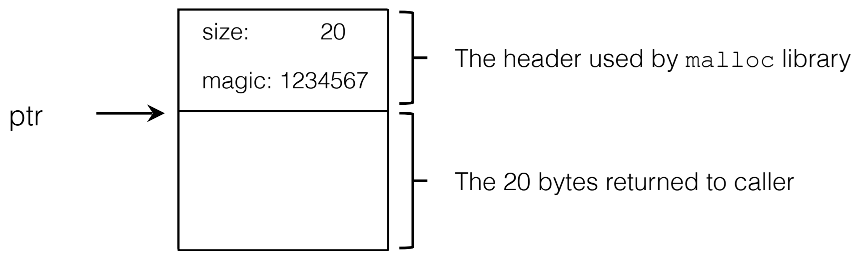
- If no large enough chunk exists, need to *coalesce:*



Tracking The Size of Allocated Regions

- `free(void *ptr)` does *not take a size parameter*.
  - How does the library **know the size** at dealloc time?
- Most allocators store extra information in a header block
  - size
  - additional pointers to speed up deallocation
  - magic number for integrity checking

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



size: 20

magic: 1234567

ptr

The header used by `malloc` library

The 20 bytes returned to caller

An Allocated Region Plus Header

Embedding A Free List

- The memory-allocation library initializes the heap and puts the first element of the free list in the free space.
- Description of a node of the list

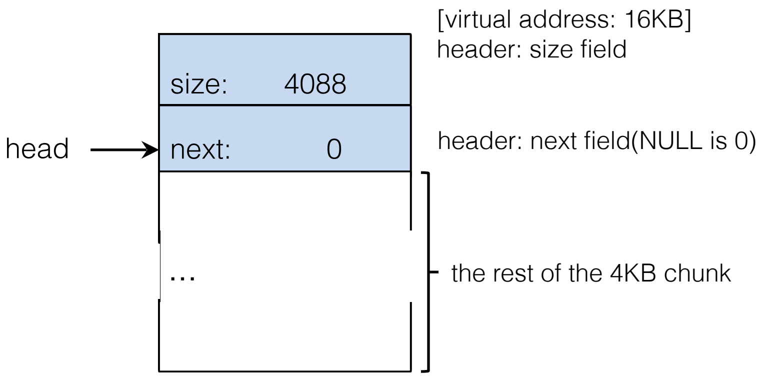
```
typedef struct __node_t {
         int size;
         struct __node_t *next;
} nodet_t;
```

- Building the heap and free list
  - Assume that the heap is built via `mmap()` system call.

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

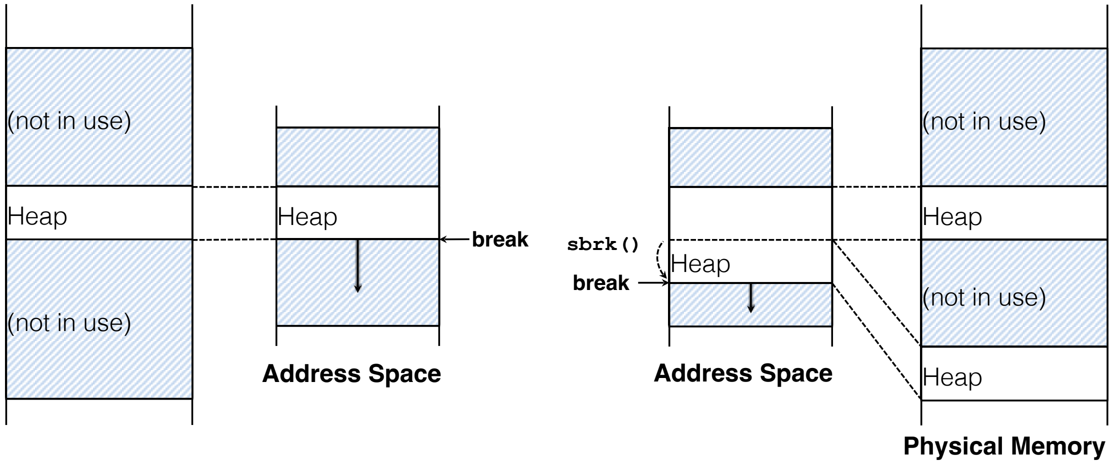
A Heap With One Free Chunk

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



Growing The Heap

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



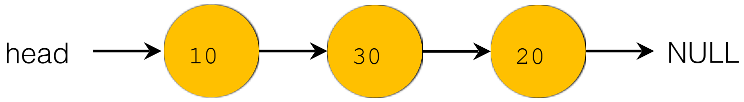
Managing Free Space: Basic Strategies

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

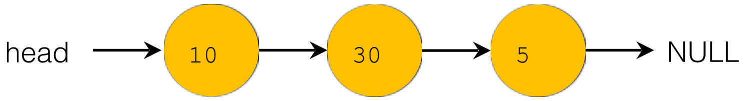
- Worst Fit:
  - Finding the **largest free chunks** and allocation the amount of the request
  - **Keeping the remaining chunk** on the free list.

Examples of Basic Strategies

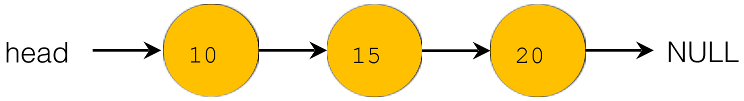
- Allocation Request Size 15



- Best-fit:



- Worst-fit:



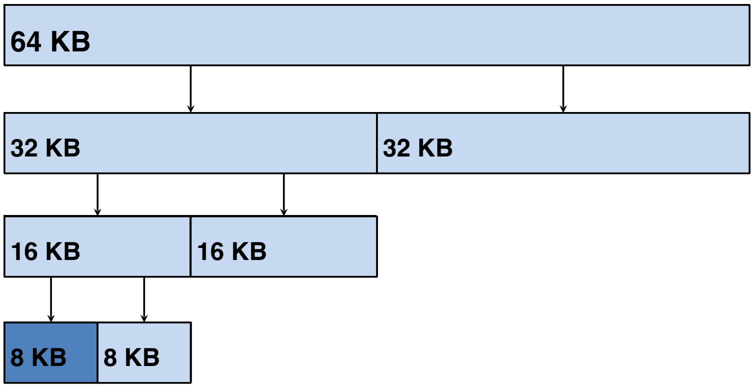
Other Approaches: Segregated List

- Segregated List:
  - Keeping separate free lists for popular requests.
  - New complication:
    - How much memory should dedicate to the pool of memory that serves specialized requests of a given size?
  - Slab allocator handles this issue.

- Slab Allocator
  - Allocate a number of caches for popular specific sizes at system boot.
    - e.g., locks, file-system inodes, etc.
  - Request a new slab from general memory allocator (size multiple of page size * object size) when a given cache is running low.

Buddy Allocation

- Binary Buddy Allocation
  - The allocator divides free space by two until a block that is big enough to accommodate the request is found.



64KB free space for 7KB request

- Internal fragmentation dealt with through coalescing:
  - When block *b* is freed, coalesce w/ buddy if also free, etc.

Memory

- 14 - Memory API
- 17 - Free Space Management
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

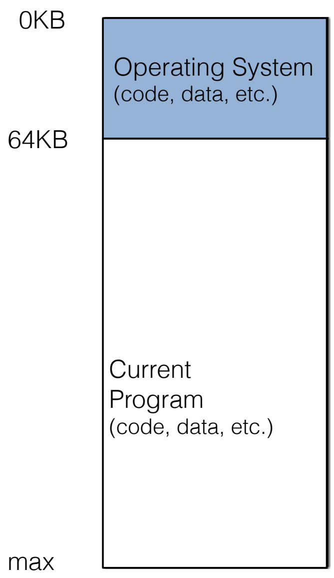
Memory Virtualization

- What is memory virtualization?
  - OS virtualizes its physical memory.
  - OS provides a virtual address space for each process.
  - Illusion of each process using the entire physical memory .

- Why?
  - Ease of use in programming
  - Memory efficiency in time and space
  - *Isolation* for processes as well as OS
    - Protection from errant accesses of other processes

Early Operating Systems

- Load only one process in memory.
    - Poor utilization and efficiency



0KB

Operating System
(code, data, etc.)

64KB

Current
Program
(code, data, etc.)

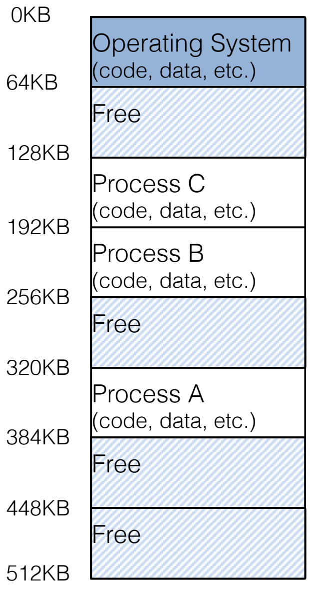
max

Physical Memory

Multiprogramming and Time Sharing

- Load multiple processes in memory
    - Execute one for a short while.
    - Switch processes between them in memory.
    - Increase utilization and efficiency.

- But what about protection?
    - Errant memory accesses from other processes



0KB

Operating System
(code, data, etc.)

64KB

Free

128KB

Process C
(code, data, etc.)

192KB

Process B
(code, data, etc.)

256KB

Free

320KB

Process A
(code, data, etc.)

384KB

Free

448KB

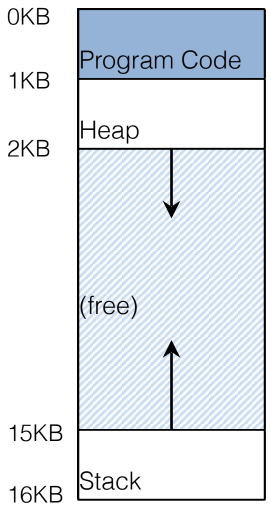
Free

512KB

Physical Memory

Address Space

- An abstraction of physical memory:



Address Space

- Code
    - Where instructions live
- Heap
    - Dynamically allocate memory.
        - `malloc` in C
        - `new` in object-oriented languages
- Stack
    - Store return addresses or values.
    - Contain local variables arguments to routines.

Virtual Addresses

- Every address in a running program is virtual.

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

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

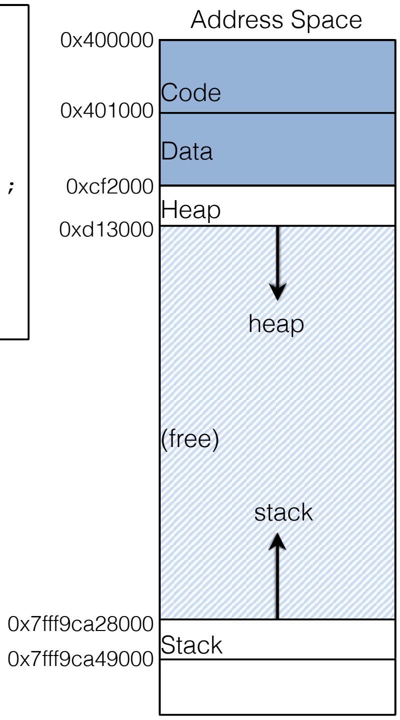
- OS uses hardware to translate virtual addresses to physical

Virtual Addresses

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

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

Output in 64-bit Linux machine:

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



Memory

- 14 - Memory API
- 17 - Free Space Management
- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Need Efficiency, and Control…

- Limited direct execution (LDE)
  - Programs run directly (not emulated)
  - Memory virtualizing, efficiency, control maintained by hardware support.
    - e.g., registers, TLBs (Translation Look-aside Buffers), page-tables

- Hardware transforms virtual addresses to physical addresses
  - Memory only addressed with physical addresses

- The OS sets up the hardware.
  - Hardware raises interrupts when needed.

Example: Address Translation

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

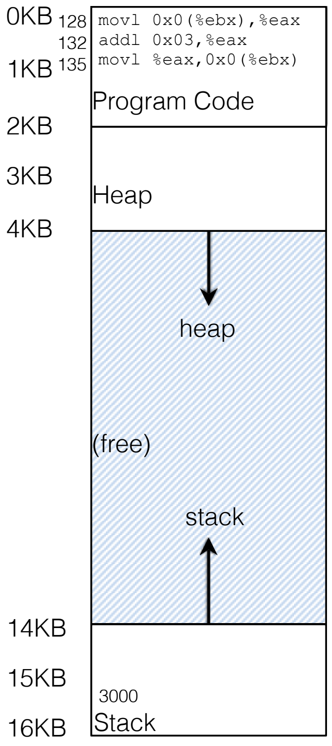
- **Load** a value from memory
- **Increment** by three
- **Store** the value back into memory

- Assembly

```
128 : movl 0x0(%ebx), %eax          ; load 0+ebx into eax
132 : addl $0x03, %eax              ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)          ; store eax back to mem
```

- Assume address of `'x'` in `ebx` register.
- **Load** the value at that address into `eax` register.
- **Add** 3 to `eax` register.
- **Store** the value in `eax` back into memory.

Example: Address Translation

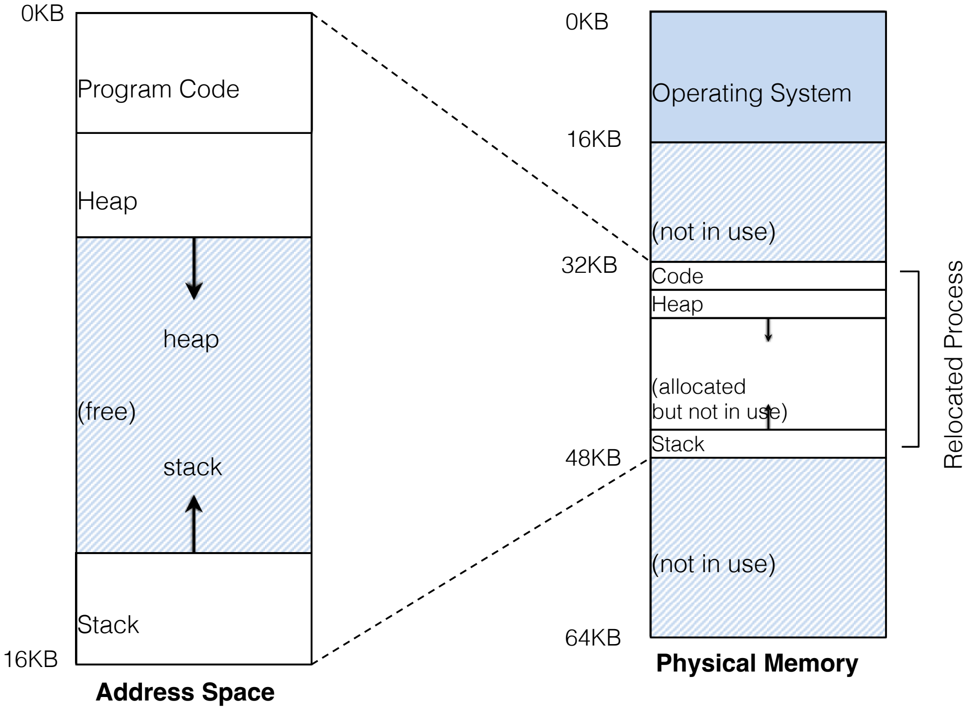


```
128 movl 0x0(%ebx),%eax
132 addl 0x03,%eax
135 movl %eax,0x0(%ebx)
```

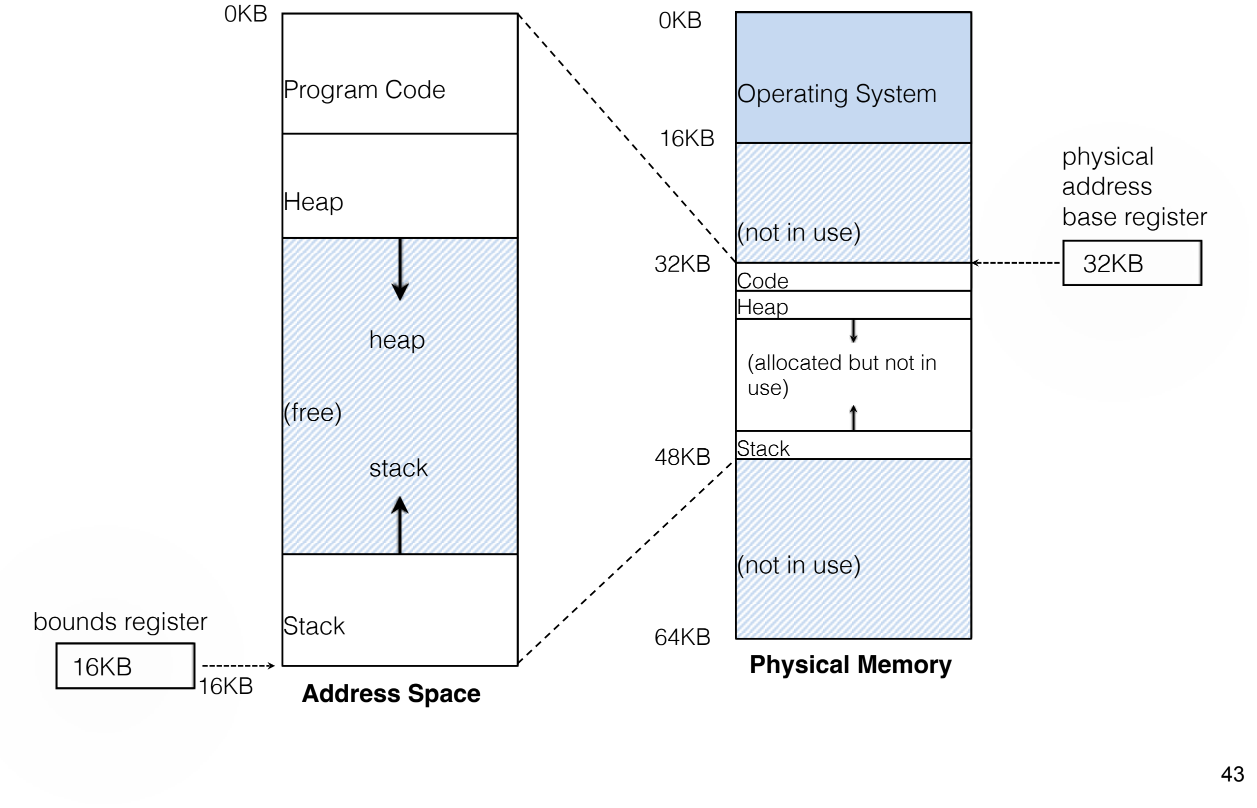
- Fetch instruction at address 128
- Execute instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute instruction (no memory reference)
- Fetch the instruction at address 135
- Execute instruction (store to address 15 KB)

But not all programs can be at location 0

A Single Relocated Process



Base and Bounds Register



0KB

Program Code

Heap

heap

(free)

stack

Stack

bounds register

16KB

16KB

Address Space

0KB

Operating System

16KB

(not in use)

32KB

Code

Heap

(allocated but not in use)

48KB

Stack

(not in use)

64KB

Physical Memory

physical address base register

32KB

Dynamic(Hardware base) Relocation

- OS decides where in physical memory a process is loaded.
  - Set the **base** register:
    physical address = virtual address + base
  - Virtual addresses must **not be greater than bound** or **negative:**
    0 <= virtual address < bound

Relocation and Address Translation

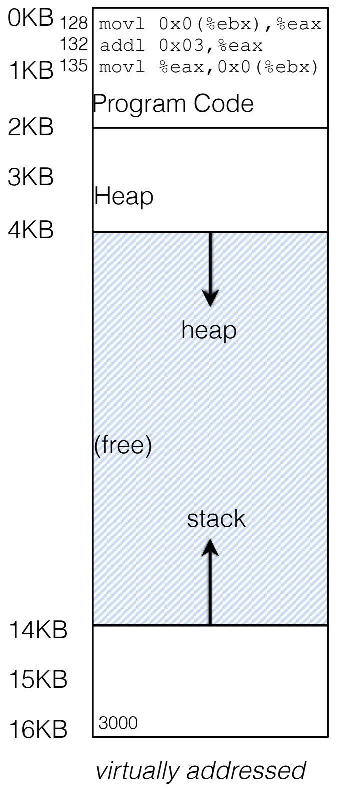
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

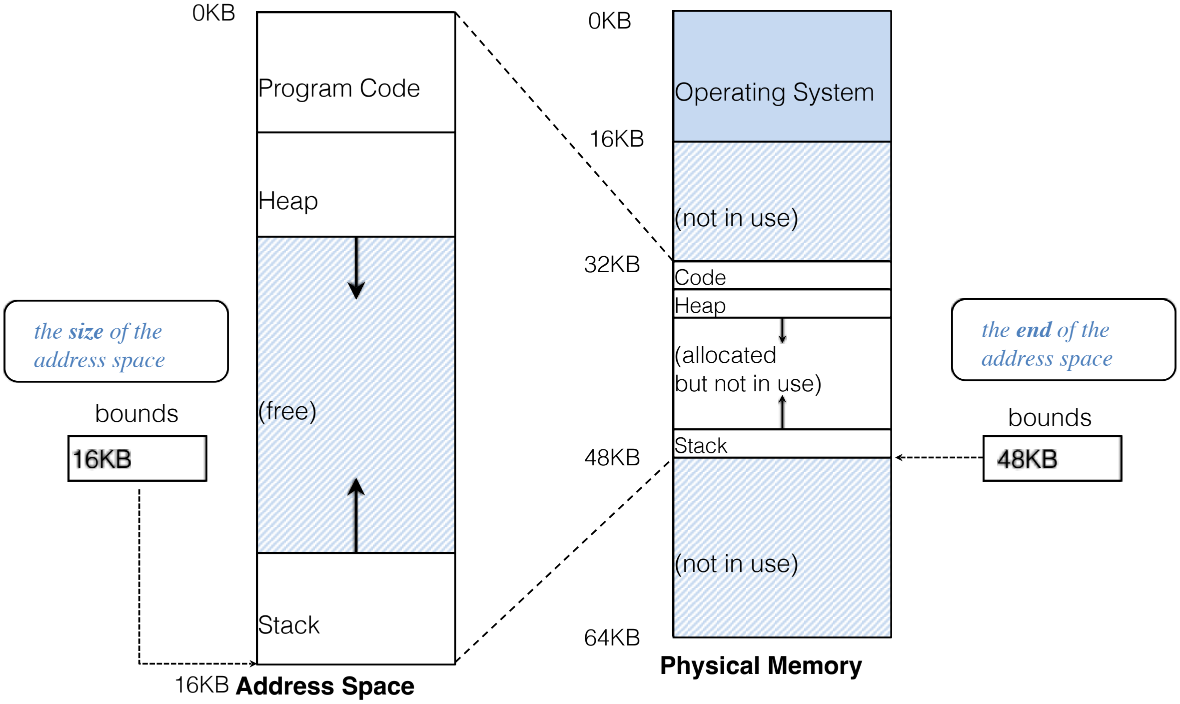
$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



Two ways to Use the Bounds Register



the ***size*** *of the address space*

the ***end*** *of the address space*

OS Issues for Memory Virtualizing

- OS intervenes at three critical junctures:
  - When a process starts running:
    - find space for address space in physical memory
  - When a process is terminated:
    - reclaims the memory for use
  - When context switch occurs:
    - Save and store the base-and-bounds pair