Chapter 9: Memory Management

Topics

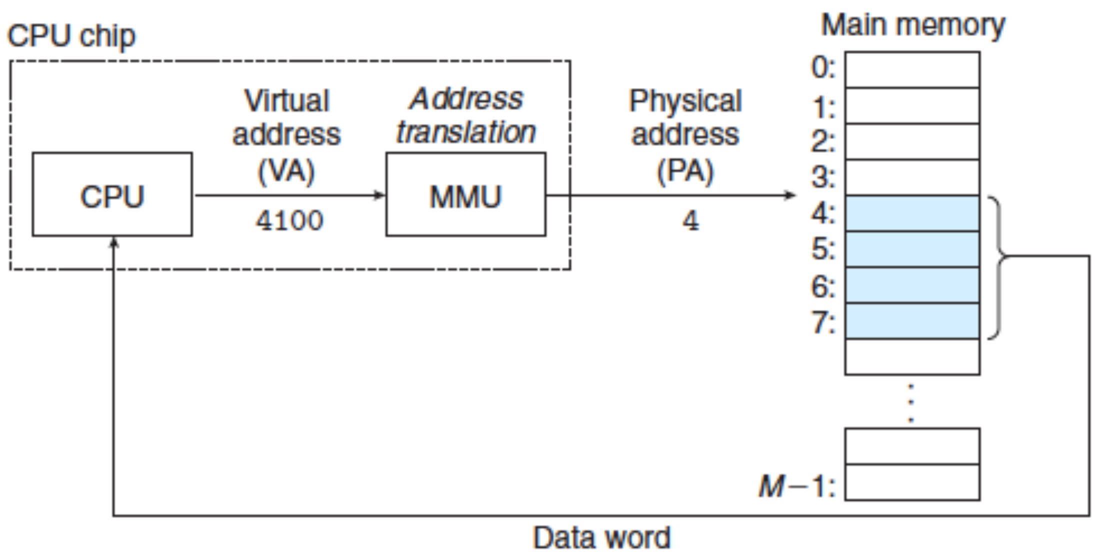
- Virtual Memory
- Heap Management

Announcements

- Shell lab is due Monday Dec 8 by 8 am
  - Interview grading time slots for next week available later this week
  - TA office hours Thursday and Friday
- Last Recitation Exercise #5 due Friday Dec 12 by 5 pm
  - Upload to moodle or hand into TA at TA office hours
- Final exam is Thursday Dec 18, 4:30-7 pm, more next week
- Reading:
  - Read Chapter 9, except 9.6 and 9.7 (no case study and no memory mapping, can also skip multi-level page tables)
- FCQs at end of class today

Recap

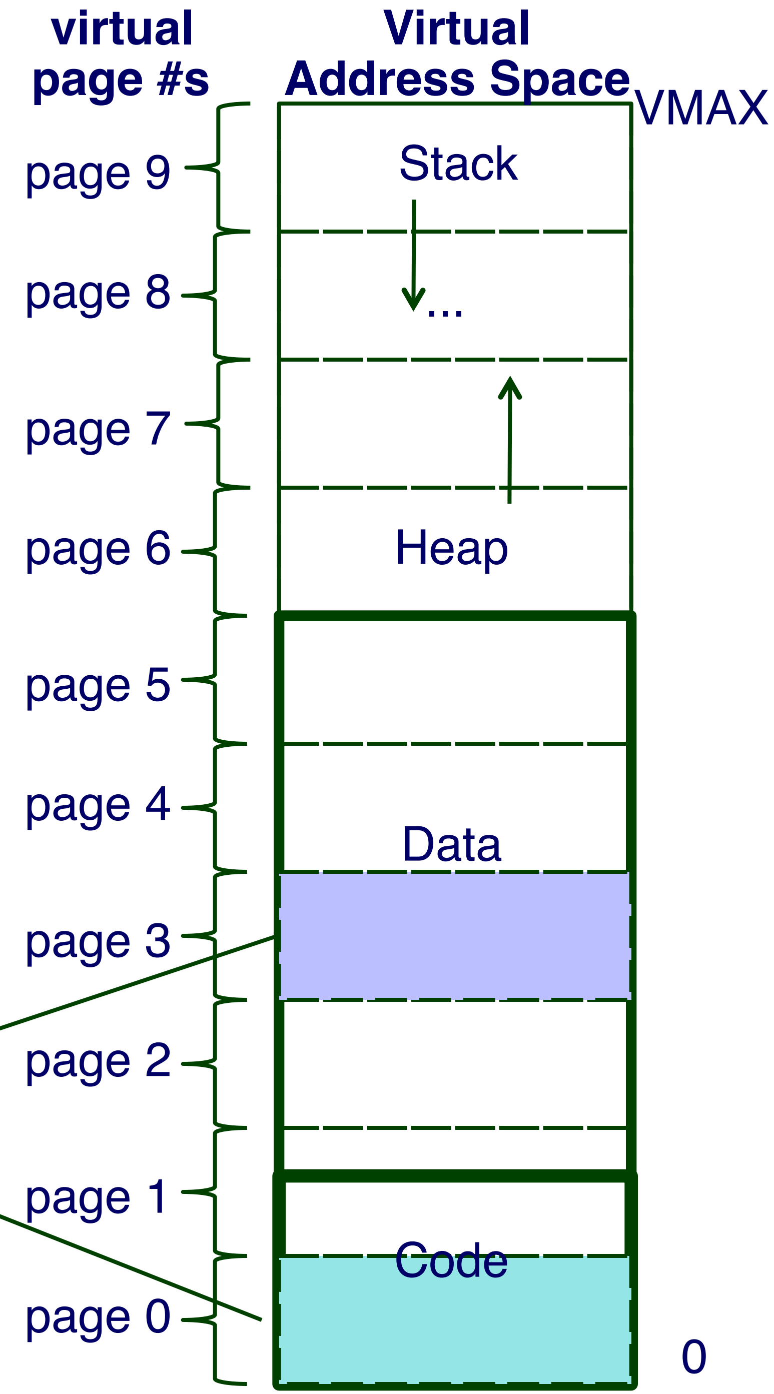
- **Virtual Memory**
  - **executables are compiled as if they would execute in their own virtual address space of memory addresses [0..VMAX]**
    - **code & data addresses are virtual**
    - **Many advantages to this approach**
  - **a Memory Management Unit (MMU) translates each virtual addresses reference into a physical address, and access memory with that physical address**

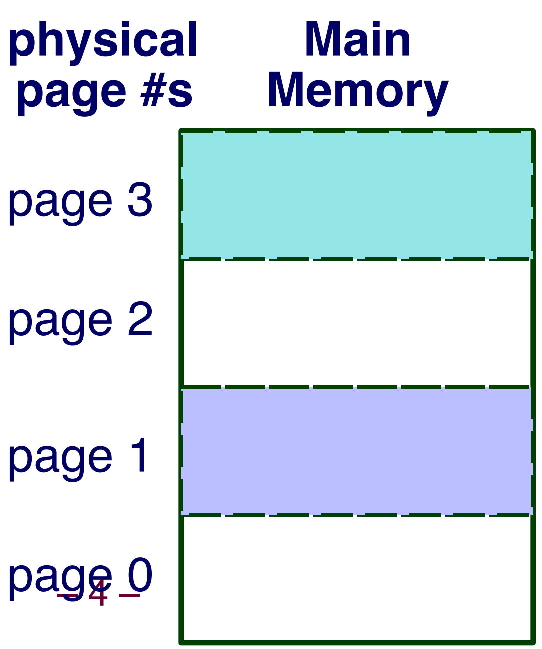


Recap: Page Tables

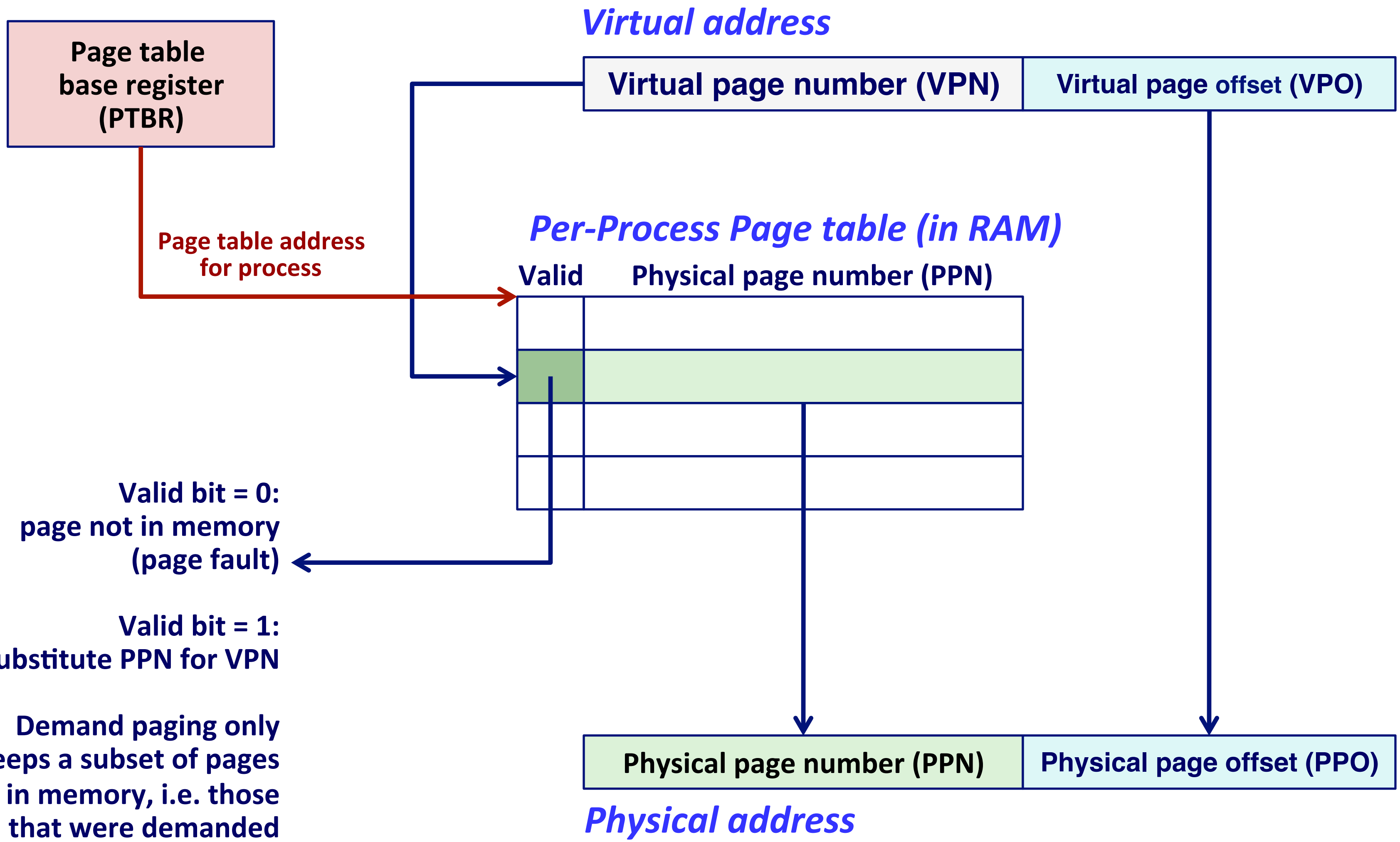
Virtual Page #	Physical Page #	valid?
0	3	1
1		0
2		0
3	1	1
4		0
5		0
6		0
7		0
8		0
9		0

valid bit indicates if page is in memory (more on this later)





Recap: Address Translation With a Page Table



Valid bit = 0:
page not in memory
(page fault)

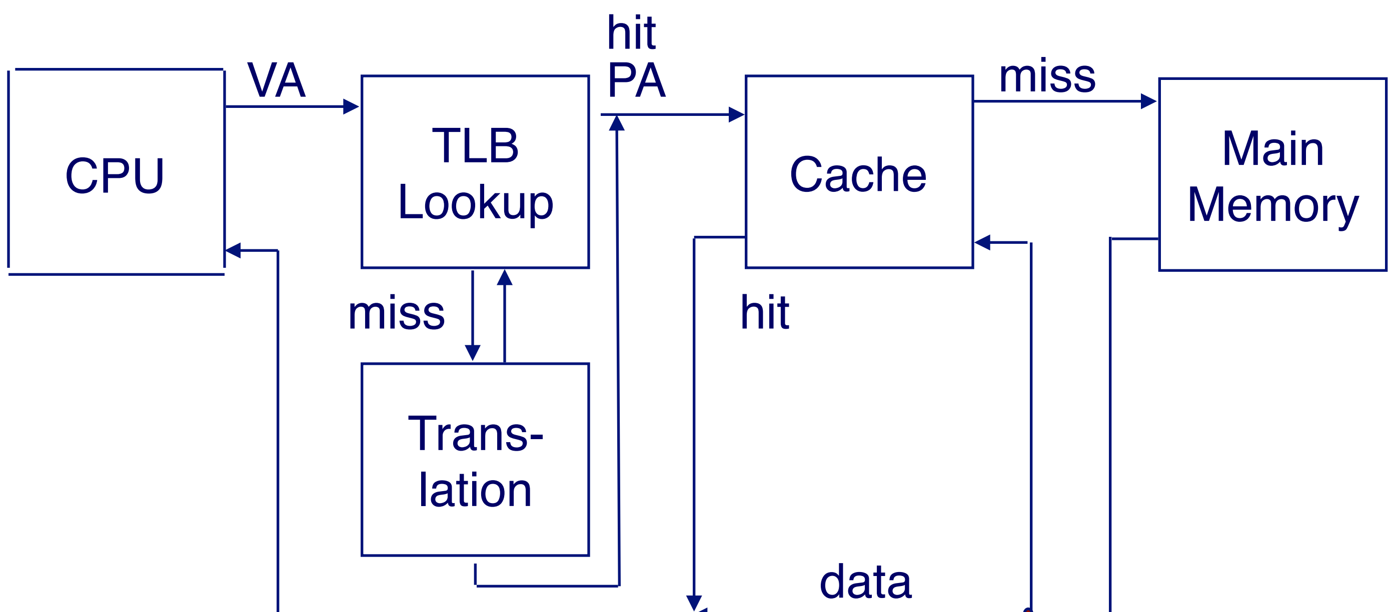
Valid bit = 1:
Substitute PPN for VPN

Demand paging only keeps a subset of pages in memory, i.e. those that were demanded for execution

Recap: Speeding up Translation with a TLB

"Translation Lookaside Buffer" (TLB)

- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages



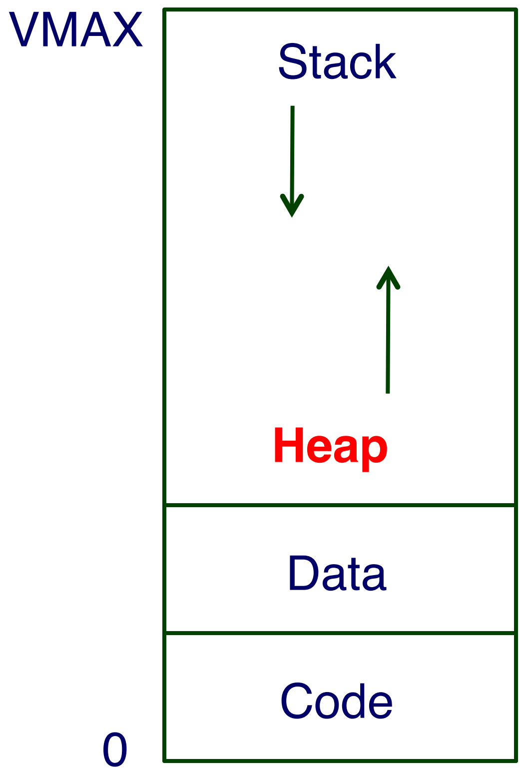
Complete example of virtual memory

- **See previous lecture's slides**

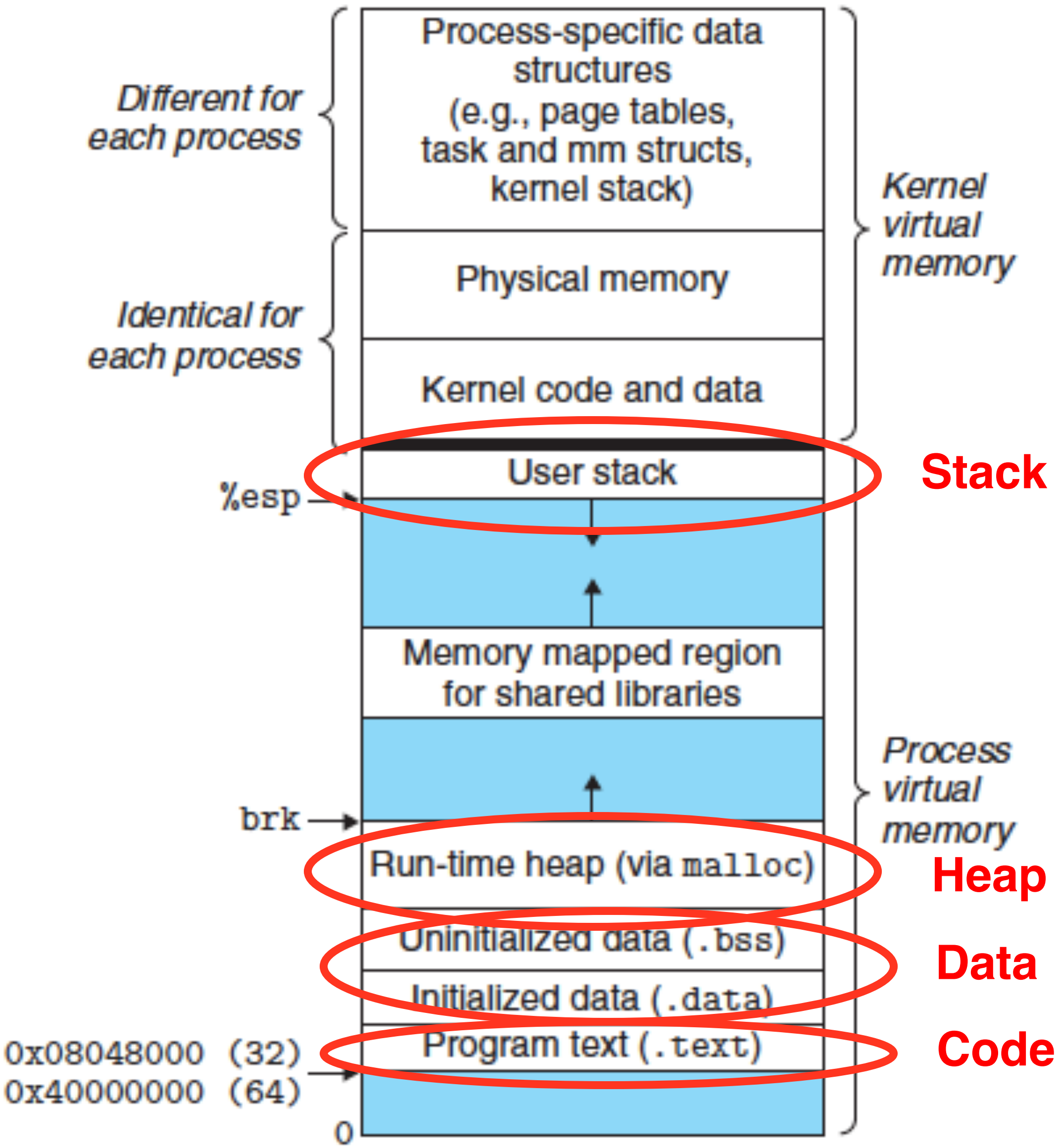
Dynamic Memory Allocation

Allocate variables dynamically at run time from the “Heap”

- Grows upwards in terms of memory addresses
- May not know until run time how large of an array, linked list, or data structure to allocate
- Useful to dynamically expand the size of a data structure, e.g. a linked list or a binary tree, by allocating more memory as needed.



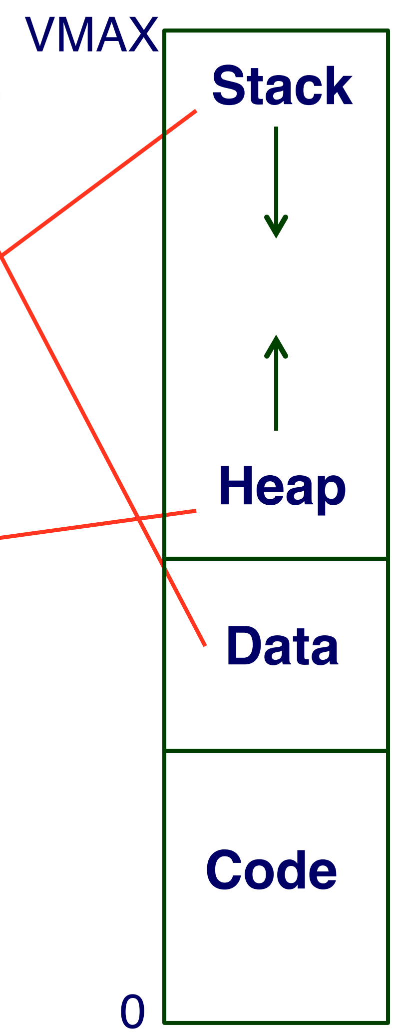
Example: Linux Address Space



Heap Allocation Example

```
int x=0,y[1000];
char *p;

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

     p = (char*) malloc(256);

     function1(p);

     free(p); /* return p to
available memory pool */

}

function1 (char *ptr) {

int i,j=50;

     for (i=0; i<100; i++) {
        *(ptr+i) = i*j;
     }

}
```

- Global variables allocated in data section of address space
- Local variables allocated on the stack
- Dynamic variables allocated on the heap
  - In C++, the “new” command is equivalent to malloc



Dynamic Memory Allocation

Application
Dynamic Memory Allocator
Heap Memory

Explicit vs. Implicit Memory Allocator

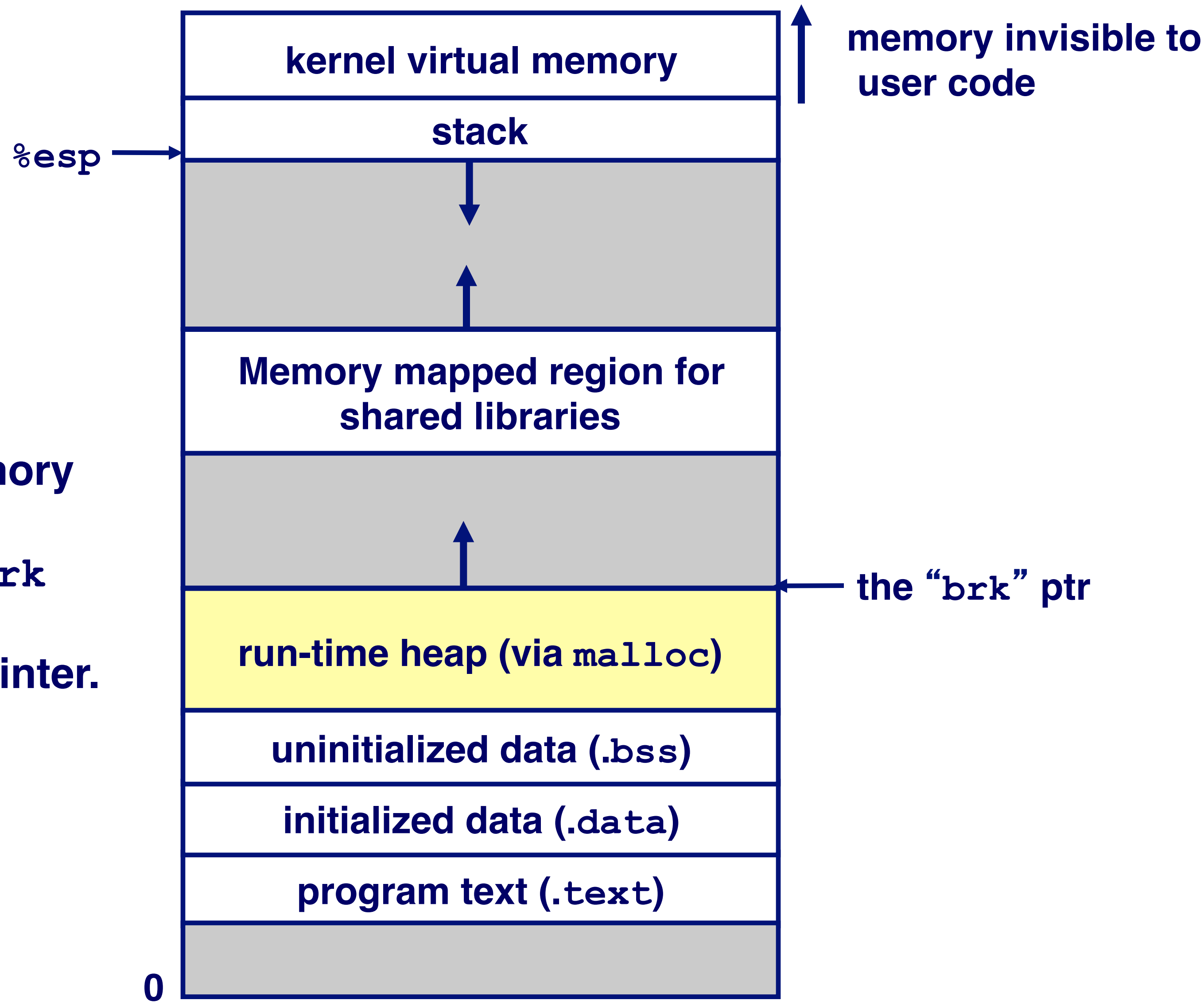
- **Explicit: application allocates and frees space**
  - E.g., `malloc` and `free` in C
  - In C++, the `new` command is equivalent to a `malloc,` and `delete` is equivalent to a `free`
- **Implicit: application allocates, but does not free space**
  - E.g. garbage collection in Java, ML or Lisp

Allocation

- **In both cases the memory allocator provides an abstraction of memory as a set of blocks**
- **Doles out free memory blocks to application**

Will discuss explicit memory allocation first

Process Memory Image



Allocators request additional heap memory from the operating system using the `sbrk` function, which increases the brk pointer. (can also shrink the heap)

Malloc Package

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

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

- If successful:
  - Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.
  - If `size == 0`, returns NULL
- If 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`.

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

- increases or decreases the size of the specified block of memory `p` and returns pointer to new block.
- Reallocates block if needed. Contents of new block unchanged up to min of old and new size.

```
void *calloc(size_t nelem, size_t elsize)
```

- Similar to malloc except initializes block of memory to zero

Malloc Example

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

  /* allocate a block of n ints */
  if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)
    p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)
    p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

Allocation Examples

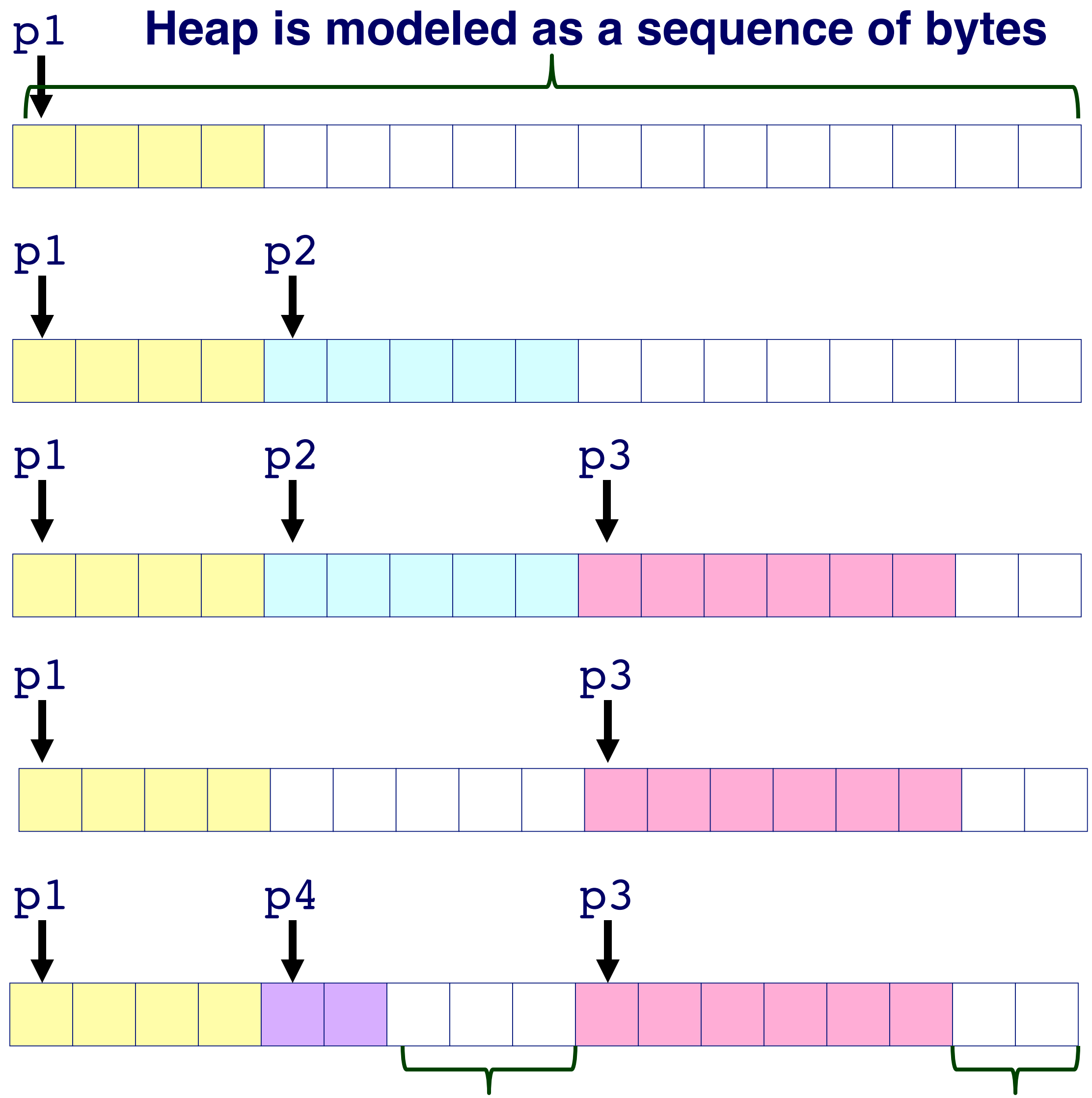
`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`

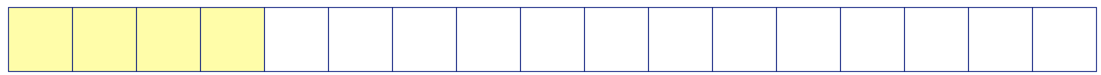


Fragmentation wastes space and may prevent allocation of big new blocks

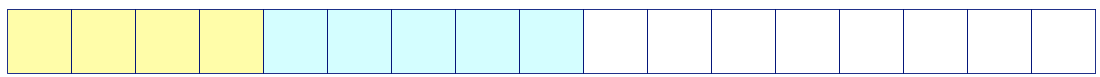
External Fragmentation

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

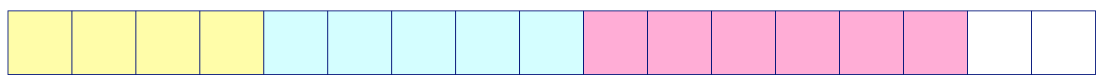
```
p1 = malloc(4)
```



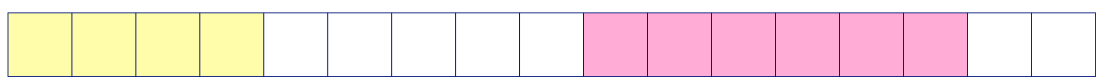
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

oops!

External fragmentation depends on the pattern of future requests, and thus is difficult to measure.

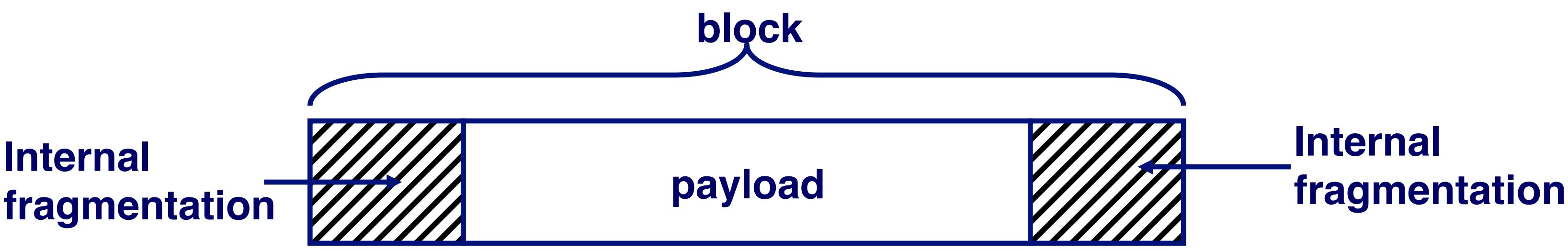
Internal Fragmentation

Poor memory utilization caused by *fragmentation*.

- Comes in two forms: internal and external fragmentation

Internal fragmentation

- For some block, internal fragmentation is the difference between the block size and the payload size.



- Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
- Depends only on the pattern of previous requests, and thus is easy to measure.

Goals of Good malloc/free

Primary goals

1. **Efficient memory utilization**
   - User allocated structures should be large fraction of the heap.
   - Want to minimize "fragmentation".
2. **Low latency/fast throughput for `malloc` and `free`**
   - Ideally should take constant time (not always possible)
   - Should certainly not take linear time in the number of blocks

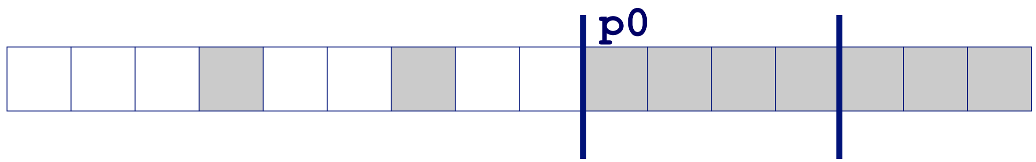
- Goals 1 and 2 are often conflicting!

Some other goals

- Good locality properties
  - Structures allocated close in time should be close in space
  - "Similar" objects should be allocated close in space
- Robust
  - Can check that `free(p1)` is on a valid allocated object `p1`
  - Can check that memory references are to allocated space

Implementation Issues

- How do we know how much memory to free given only 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 a freed block?

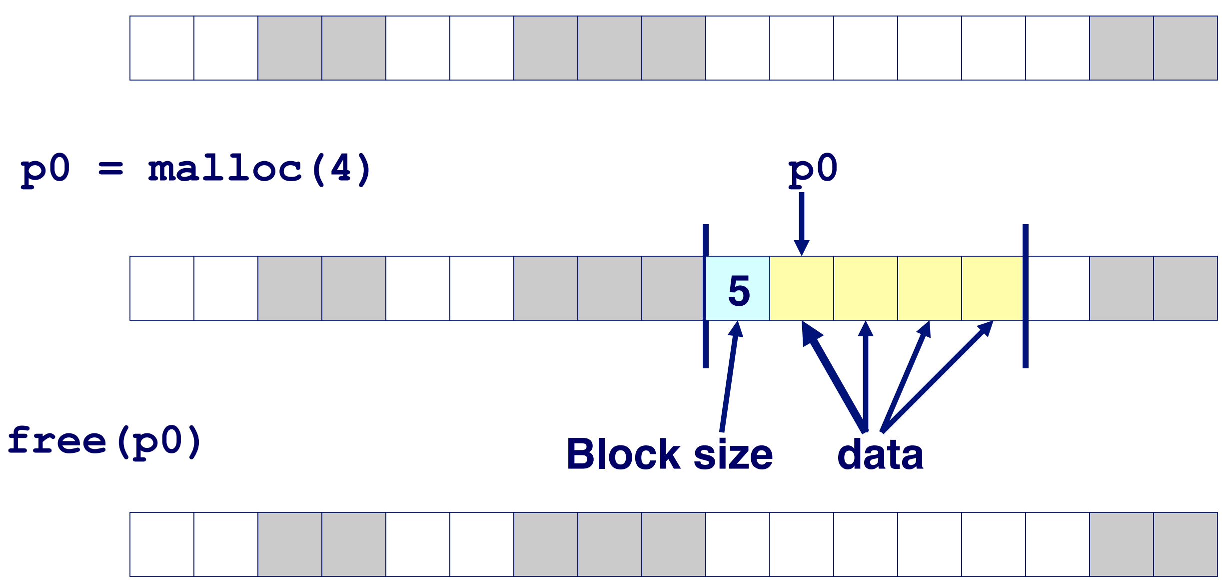



```
free(p0)

p1 = malloc(1)
```

Knowing How Much to Free

Standard method

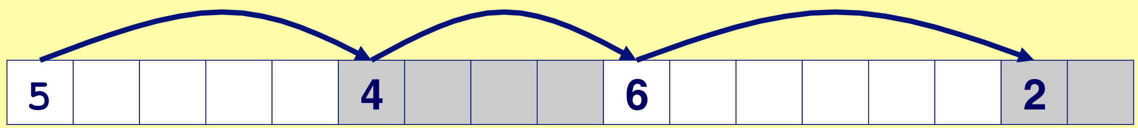
- Keep the length of a block in the word preceding the block.
  - This word is often called the *header field* or *header*
- Requires an extra word for every allocated block
- Let's assume each square in figure is large enough (say 4 bytes) to contain a word, e.g. an int or pointer



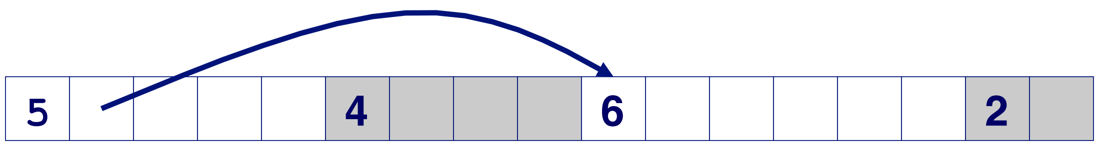
Note if p0 in free(p0) is not originally malloc'ed, this may cause an error

Keeping Track of Free Blocks

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



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



Method 3: ***Segregated free list***

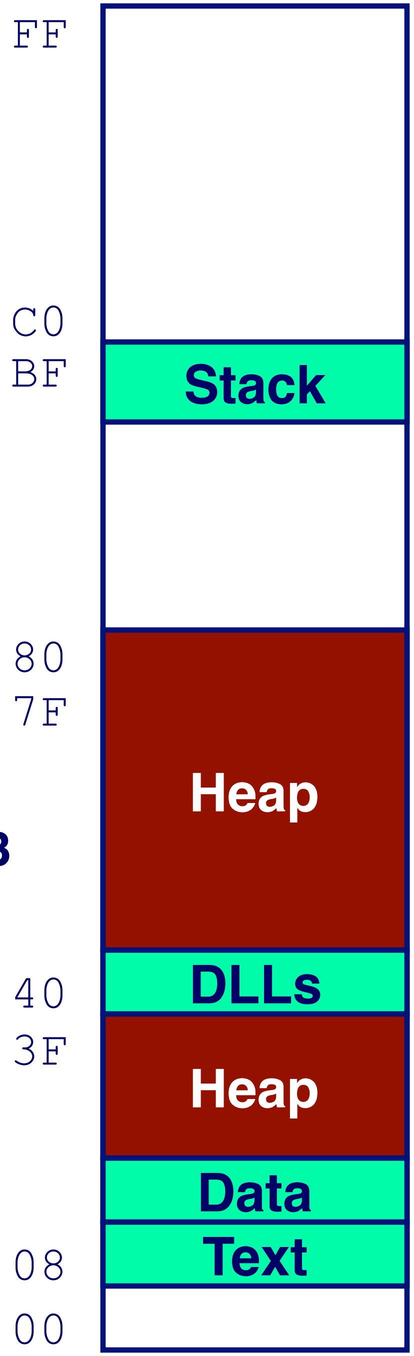
- Different free lists for different size classes

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

Supplementary Slides

Linux Memory Layout



Upper 2 hex digits of address

Red Hat v. 6.2 ~1920MB memory limit

Stack

- Runtime stack (8MB limit)

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

DLLs

- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

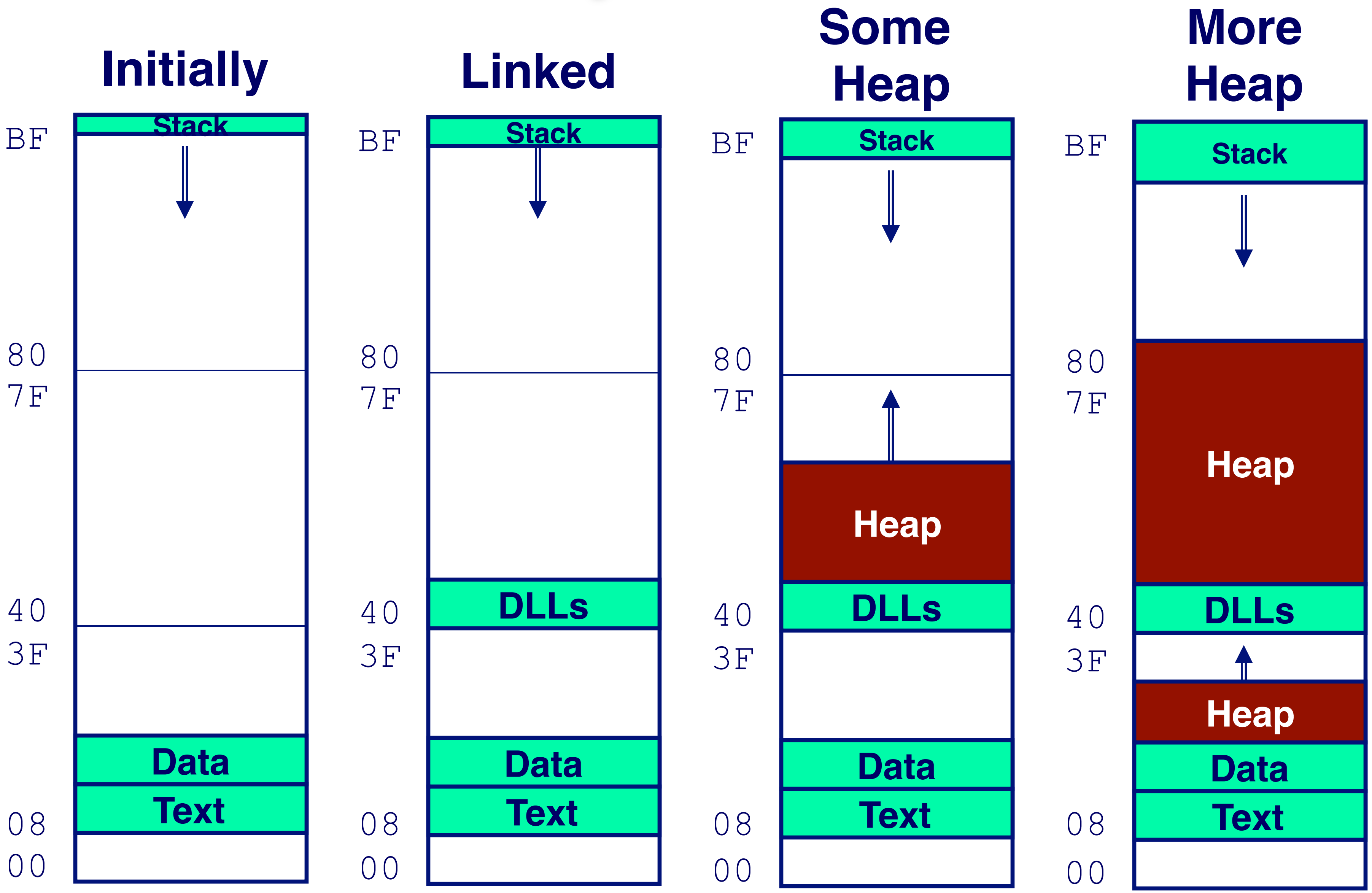
Data

- Statically allocated data
- E.g., arrays & strings declared in code

Text

- Executable machine instructions
- Read-only

Linux Memory Allocation



Constraints

Applications:

- Can issue arbitrary sequence of allocation and free requests
- Free requests must correspond to an allocated block

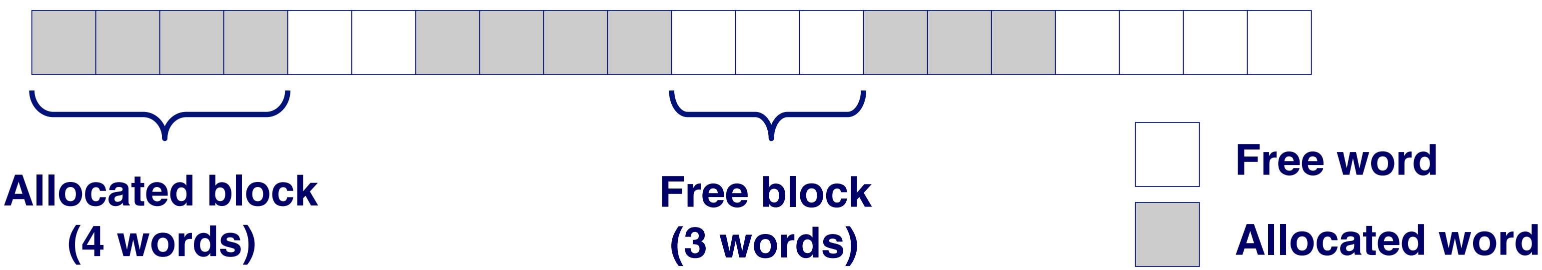
Allocators

- Can’t control number or size of allocated blocks
- Must respond immediately to all allocation 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
- Can’t move the allocated blocks once they are allocated
  - *i.e.*, compaction is not allowed
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU malloc (`libc` malloc) on Linux boxes
- Can only manipulate and modify free memory

Assumptions

Model the Heap memory as a sequence of words

- Memory is word addressed (each word can hold a pointer)



Performance Goals: Throughput

Given some arbitrary sequence of malloc and free requests:

- $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

Want to 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.

Performance Goals: Peak Memory Utilization

Given some sequence of malloc and free requests:

- $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

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 is denoted by H_k

- Assume that H_k is monotonically nondecreasing

Def: Peak memory utilization:

- After *k* requests, *peak memory utilization* is:
  - $U_k = (\max_{i<k} P_i) \;/\; H_k$