Pointers and Heap Manager

COMPSCI210 Recitation

7 Sep 2012

Vamsi Thummala

Agenda

Pointers/Alignment/Casting in C

Heap Manager: Dynamic memory allocation

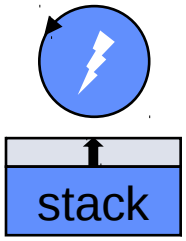
Review: Processes

virtual address space

The address space is a private name space for a set of memory segments used by the process.

The kernel must initialize the process memory for the program to run.

thread

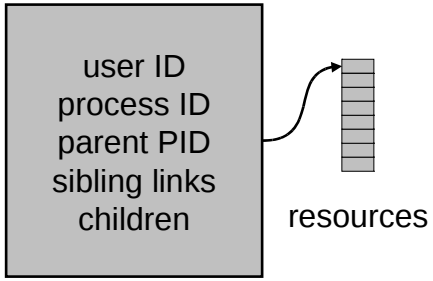


Each process has a thread bound to the VAS.

The thread has a stack addressable through the VAS.

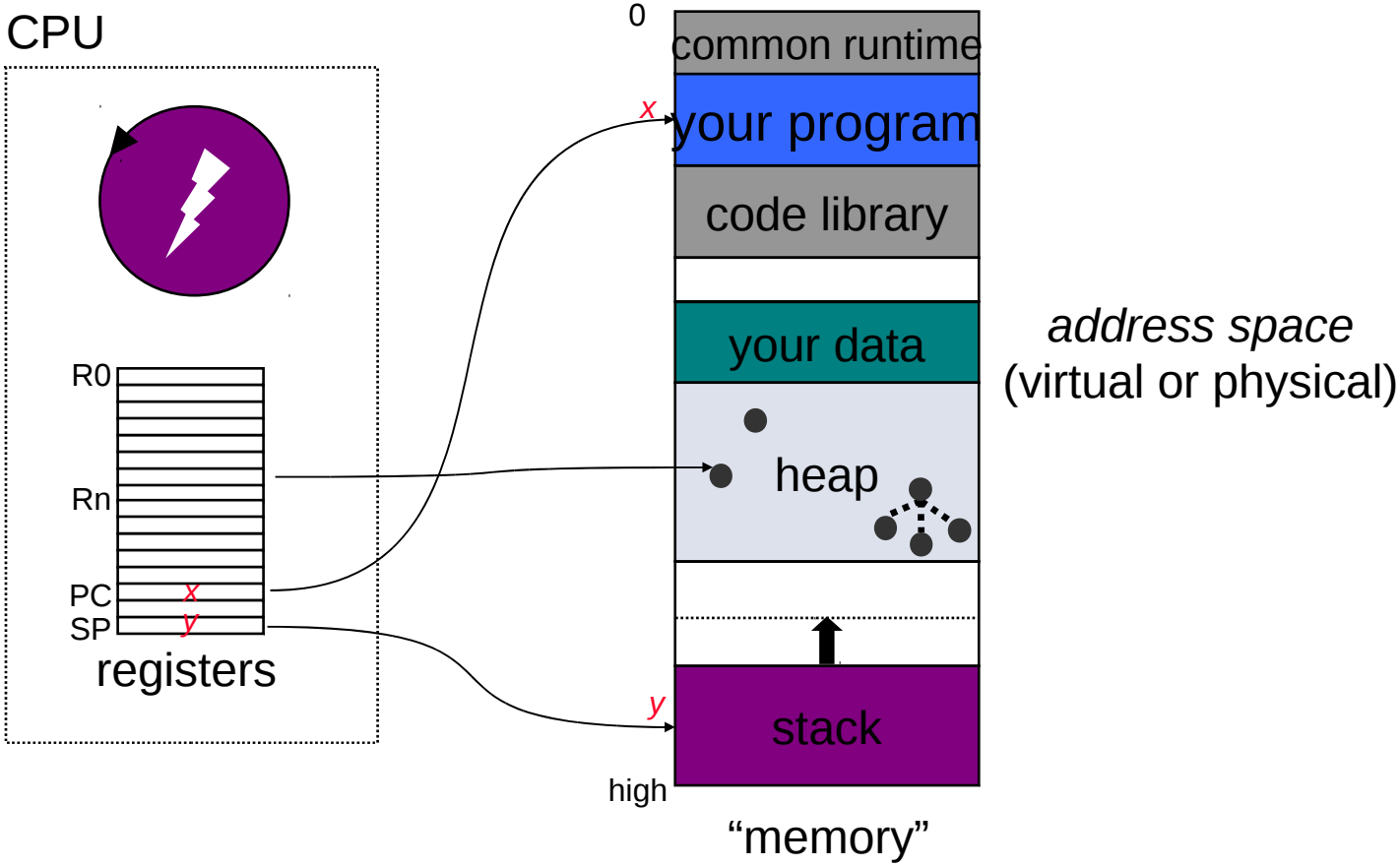
The kernel can suspend/restart the thread wherever and whenever it wants.

process descriptor (PCB)

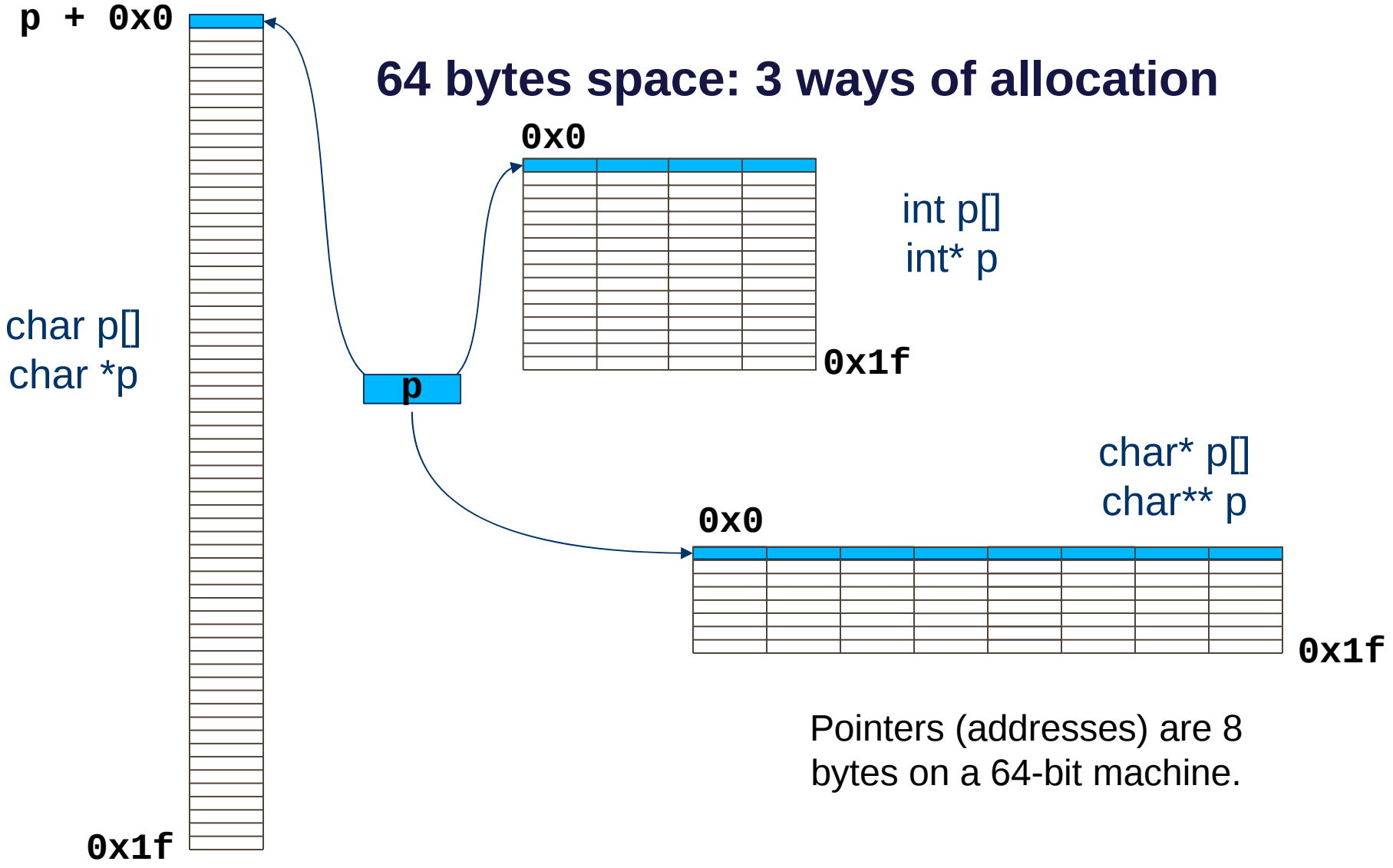


The OS maintains some state for each process in the kernel's internal data structures: a file descriptor table, links to maintain the process tree, and a place to store the exit status.

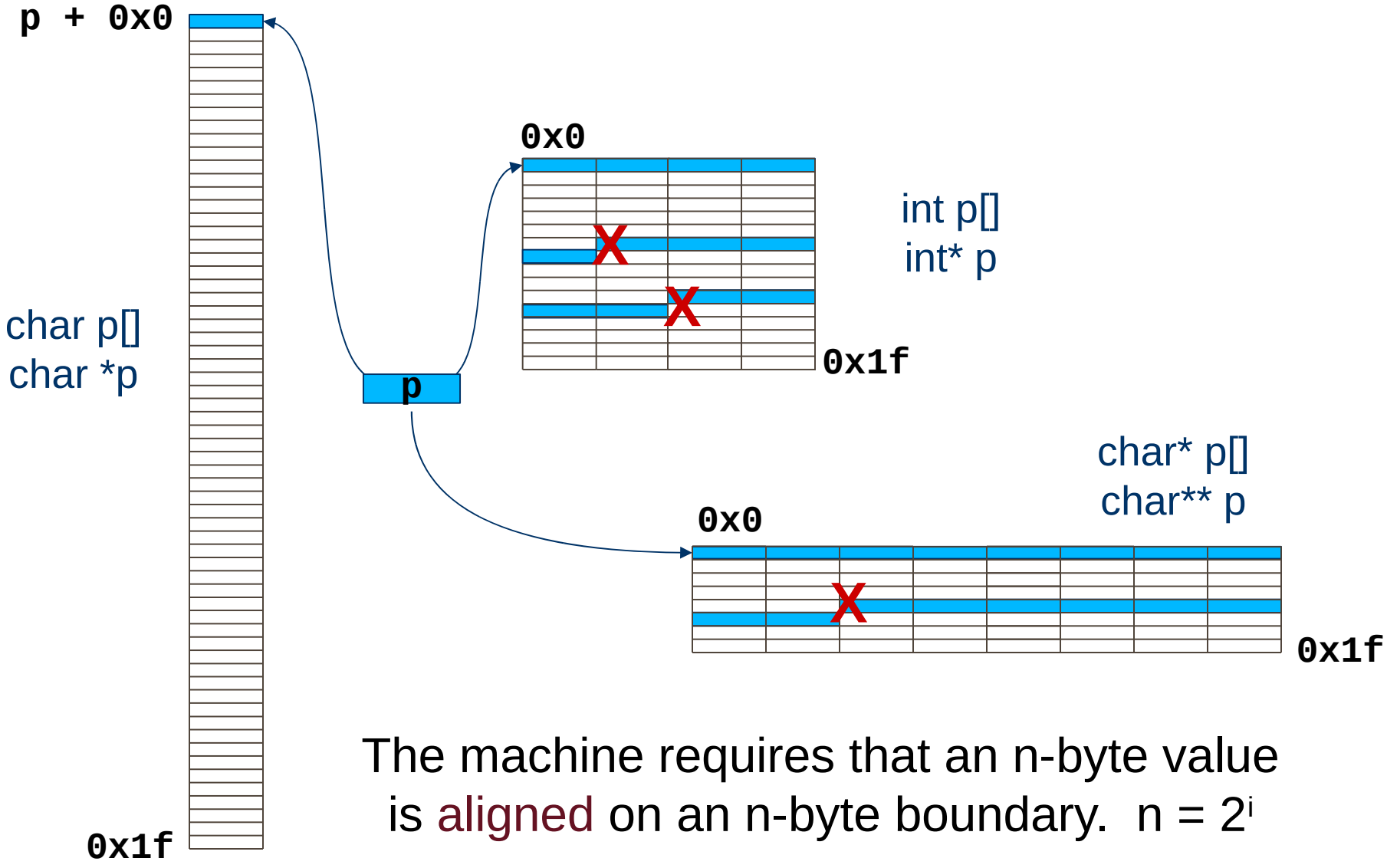
Review: Address space



Pointers, data types



Alignment



The machine requires that an n-byte value is aligned on an n-byte boundary. $n = 2^i$

Pointer casting

What actually happens in a pointer cast?

Nothing! It’s just an assignment. Remember all pointers are the same size.

The magic happens in dereferencing and arithmetic

Pointer dereferencing

What gets “returned?”

```
int * ptr1 = malloc(100);
*ptr1 = 0xdeadbeef;


int val1 = *ptr1;

int val2 = (int) *((char *) ptr1);


What are val1 and val2?
```

Pointer dereferencing

What gets “returned?”

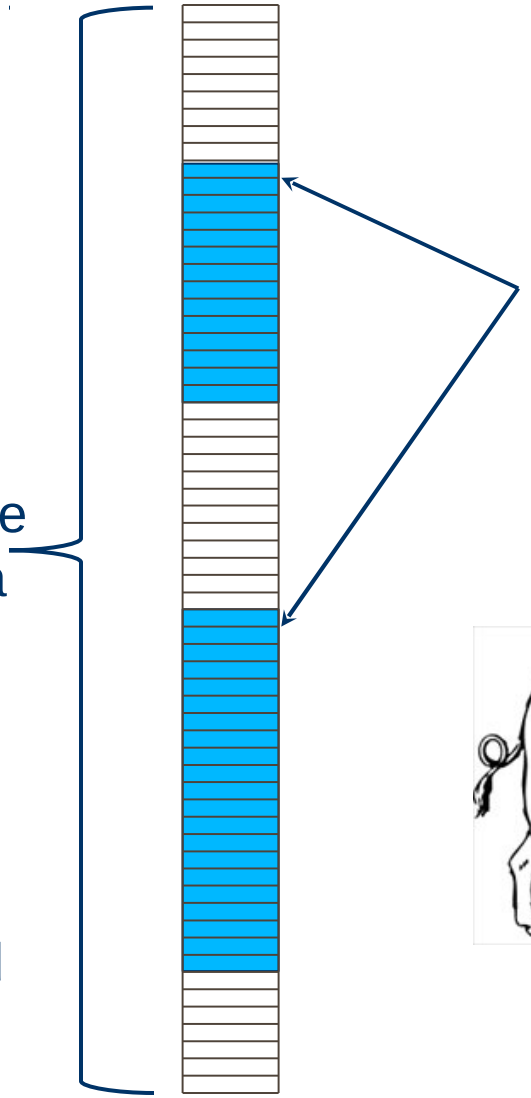
```
int * ptr1 = malloc(sizeof(int));
*ptr1 = 0xdeadbeef;

int val1 = *ptr1;

int val2 = (int) *((char *) ptr1);

//  val1 = 0xdeadbeef;

//  val2 = 0xffffffef;

 What happened??
```

Heap allocation

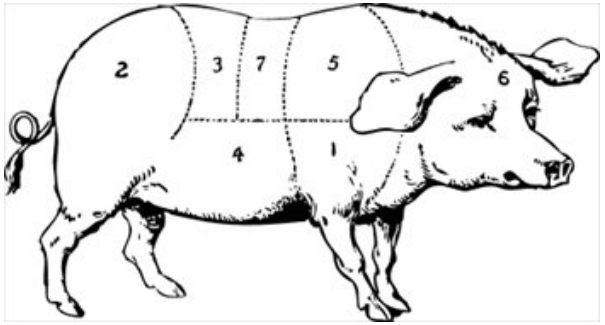
A contiguous chunk of memory obtained from OS kernel.
E.g., with Unix *sbrk*() system call.

A runtime library obtains the block and manages it as a "heap" for use by the programming language environment, to store dynamic objects.

E.g., with Unix *malloc* and *free* library calls.



Allocated heap blocks for structs or objects.
Align!



Design considerations

I found a chunk that fits the necessary payload… should I look for a better fit or not?

Splitting a free block:

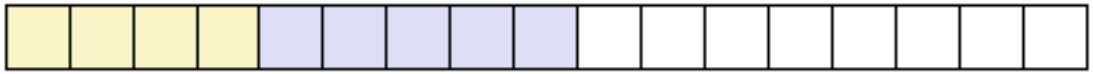
```
void* ptr = malloc(200);

free(ptr);

ptr = malloc(50); //use same space, then “mark” remaining
                  bytes as free
```

```
void* ptr = malloc(200);

free(ptr);

ptr = malloc(192);//use same space, then “mark” remaining
                  bytes as free??
```

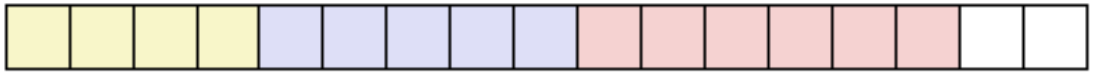
Allocation Example

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

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



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



```
free(p2)
```

```
p4 = malloc(2)
```

Fragmentation

Internal fragmentation

Result of **payload** being smaller than block size.

```
void * m1 = malloc(3); void * m1 = malloc(3);

m1,m2 both have to be aligned to 8 bytes…


External fragmentation
```

External Fragmentation

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

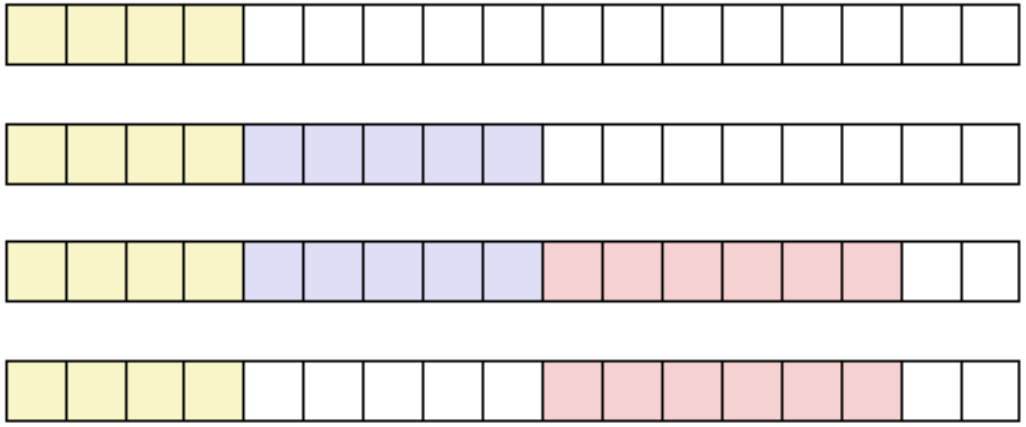
```
p1 = malloc(4)
```

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

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

```
free(p2)
```

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



Oops! (what would happen now?)

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

Implementation Hurdles

How do we know where the chunks are?

How do we know how big the chunks are?

How do we know which chunks are free?

Remember: no queuing of buffer calls to malloc and free… must deal with them real-time.

Remember: calls to `free only takes a pointer, not a pointer and a size.`

`Solution:` **`Need a data structure to store information on the “chunks”`**

`Where do I keep this data structure?`

The data structure

Requirements:

The data structure needs to tell us where the chunks are, how big they are, and whether they're free

We need to be able to CHANGE the data structure during calls to malloc and free

We need to be able to find the **next free chunk** that is "a good fit for" a given payload

We need to be able to quickly mark a chunk as free/allocated

We need to be able to detect when we're out of chunks.

- What do we do when we're out of chunks?

The data structure

It would be convenient if it worked like:

```
    malloc_struct malloc_data_structure;

    …
ptr = malloc(100, &malloc_data_structure);

    …

    free(ptr, &malloc_data_structure);

    …
```

Instead all we have is the memory we are giving out.

All of it does not have to be payload! We can use some of that for our data structure.

The data structure

The data structure IS your memory!

A start:

<h1> <pl1> <h2> <pl2> <h3> <pl3>

What goes in the header?

- That’s your job!

Lets say somebody calls free(p2), how can I coalesce?

- Maybe you need a **footer**? Maybe not?

The data structure

Check the example metadata structure provided in cps210_mm.h

```
typedef struct metadata {
        size_t size;
        struct metadata* next;
        struct metadata* prev;
} metadata_t;
```

Design considerations

Free blocks: address-ordered or LIFO or FIFO

What’s the difference?

Pros and cons?

What are the efficiency tradeoffs?