Introducing Malloc

Dr Andrew Scott

a.scott@lancaster.ac.uk

1

malloc( )

- Provides dynamic memory allocation
  - Allocates memory from process heap
- Returns pointer to requested amount of memory
- Pointer remains valid until call to *free( )*
  - No automatic garbage collection -- DIY
- *malloc( )* must be followed by one, and only one, *free( )*

2

In use…

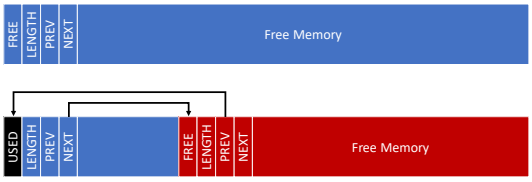
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *
first( ) {
    char * first_buff =  ( char * ) malloc( 7 );

    return strcpy( first_buff,  "Hello" );
}

char *
second( ) {
    char * second_buff = ( char * ) malloc( 7 );

    return strcpy( second_buff, "World!" );
}

int
main( ) {
    char * str1 = first(  );
    char * str2 = second( );

    printf( "%s %s\n", str1, str2 );

    free( str1 ); free( str2 );
}
```

3

A simple *malloc( ): Structure*

4

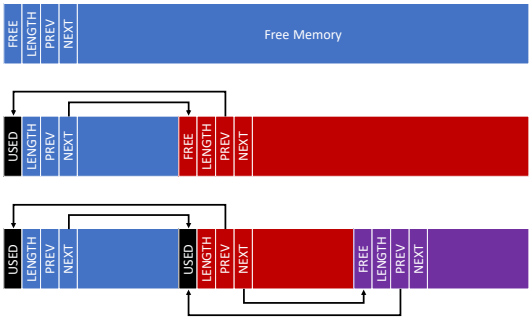
Simple *malloc( ):* allocating



- Simple approach…
  - Chain all blocks together: *Previous and Next pointers*
  - Search chain/ list for free space when needed
    - Maintain *Free/ Used* indicator

5

Simple *malloc( ):* allocating II

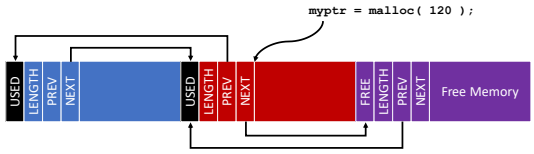


6

Simple *malloc( ):* the user view…

- *malloc( )* returns start address of free area
- All within one process space – no protection
  - User must never write outside returned memory
    - Would overwrite metadata and break lists

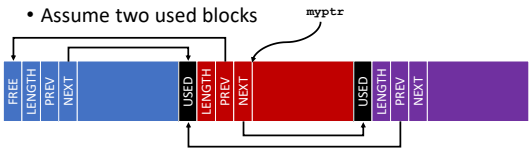
…but it's their data they'd be corrupting



7

Simple *malloc( )*: freeing space

- Assume two used blocks



- *free(* myptr *)* : Coalesce adjacent free blocks

8

Heap Allocation Test: The real *malloc( ) and free( )*

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

int
main( ) {
    void *   addr1, addr2, addr3, addr4, addr5, addr6;

    addr1 = malloc( 1024 );  addr2 = malloc( 1024 );

    free( addr1 );  free( addr2 );

    addr3 = malloc( 1024 );  addr4 = malloc( 1024 );

    free( addr3 );  free( addr4 );

    addr5 = malloc( 512 );  addr6 = malloc( 512 );

    free( addr5 );  free( addr6 );

    printf( "Addr1 = %p, Addr2 = %p\n", addr1, addr2 );
    printf( "Addr3 = %p, Addr4 = %p\n", addr3, addr4 );
    printf( "Addr5 = %p, Addr6 = %p\n", addr5, addr6 );
}
```

9

malloc( ) and free( ) Test Result

- Recall we:
  - Allocate 1024, 1024 …then free both
  - Allocate 1024, 1024 …then free both
  - Allocate 512, 512

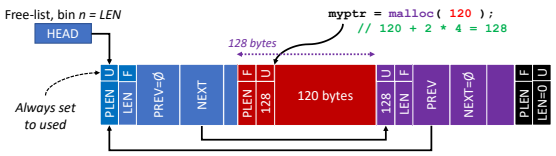
```
Addr1 = 0x260, Addr2 = 0x670
Addr3 = 0x670, Addr4 = 0x260
Addr5 = 0xa80, Addr6 = 0xc90
```

- Notice
  - Freeing to head of list, so Addr4 lower address than Addr3
    - So using a separate free list
  - Using separate free lists for different allocation sizes
    - Didn't use free 1024 byte blocks for serving 512 byte requests

10

A Better *malloc( )*

- Free areas have PREV and NEXT pointers to form free list
  - As these are pointers, they are each a double word (8 bytes)
  - Not needed for used areas… can be overwritten by user application
- 'Fake'/ guard (P)LEN:U values at ends, so not special case



11

A better *malloc( ):* note…

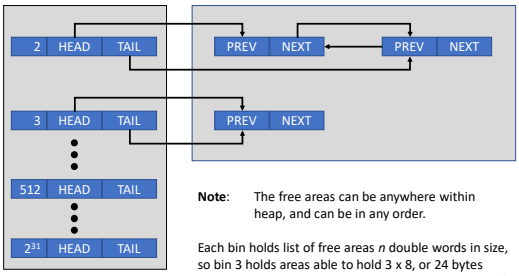
- Keep length values to enable coalescing free areas
  - Length of current block, LEN (4 bytes… smaller than ptr.)
  - Length of previous block, PLEN (4 bytes)
  - Steal bit from Length fields for Used (U)/ Free (F) flag
- Maintain free lists: only need pointers for free areas
  - Multiple (typically 128) free-lists/ *bins* for useful sizes
    - Fixed size 2*, 3, … 64 double word blocks (16..512 bytes)
    - Rest for variable length allocations up to 2^{31} bytes

Return double word aligned addresses so user can safely use for pointers to any object type

* Need min. 8 bytes for pointer; if NEXT and PREV, min 16 bytes, i.e. start at 2

12

malloc( ) Free Bins



128 Bins in total

Note: The free areas can be anywhere within heap, and can be in any order.

Each bin holds list of free areas *n* double words in size, so bin 3 holds areas able to hold 3 x 8, or 24 bytes
-- but these will additionally have 2 x 4 byte LEN/F fields occupying 28 bytes in total

13

Growing the Heap: *sbrk( )*

- When *malloc( )* runs out of space it calls *sbrk( )*
- Heap has *high water mark*, the program *break*
  - Read using: *sbrk(0)*
  - Can be pushed up *sbrk(+ve)* or pulled down *sbrk(-ve)*
    - *In practice, generally snapped to page* boundary*
  - Can be set explicitly using *brk(addr)*

* As we'll see later, hardware chunks memory into (typically) 4K blocks called *pages*

14

Allocating whole pages: *mmap( )*

- For large requests, whole *pages* allocated in separate memory region using *mmap( )*
  - More efficient and avoids large void on heap seen if two small allocations bookend a free multi-page area

```
/*
 *     Allocate single page, at any/ best (NULL) address,
 *             PRIVATE (non-shared)
 *             and ANONYMOUS, so not backed by a file
 */
char * p = mmap( NULL, sysconf( _SC_PAGE_SIZE ),
                 PROT_READ   | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS,
                -1, 0
           );
```

15

mmap( ), full example…

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

int
main( ) {
        char * p = mmap( NULL, sysconf( _SC_PAGE_SIZE ),
                         PROT_READ   | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS,
                        -1, 0
                       );

        if( p <= 0 ) {
                perror( "mmap" );
                exit( EXIT_FAILURE );
        }

        strcpy( p, "Hello World!" );

        puts( p );
}
```

16

If sbrk adds memory, why malloc?

- *sbrk( )* returns or adjusts high water mark
  - No knowledge of what is below this mark
- *malloc( )* tracks/ manages free space
  - Knows which areas free and which are used
  - Splits free areas down to required size
- Beware: Don't forget that *malloc( )* uses *sbrk( )*
  - *malloc( )* can't assume it's the only thing using heap memory
  - Neither can you!
    - For *p = sbrk(n)* you can only use byte addresses *p..p+n-1*, *n* > 0

17

malloc( ) Final Thoughts

- Number of alternative approaches, such as…
  - Tree structures, and Buddy algorithm that we'll see later
- Three main problems
  - How to limit heap size by avoiding unnecessary free space
    - i.e. how to keep *sbrk( )* high water mark as low as possible
  - How to limit, or manage, memory fragmentation
    - Small amounts of unused space between allocated memory
    - Note: we can't shuffle things in memory as we don't know how programs are using memory
  - How to choose between two or more free areas
    - And should newly freed areas be at head or tail of free list

18