CS 3214: Project 3

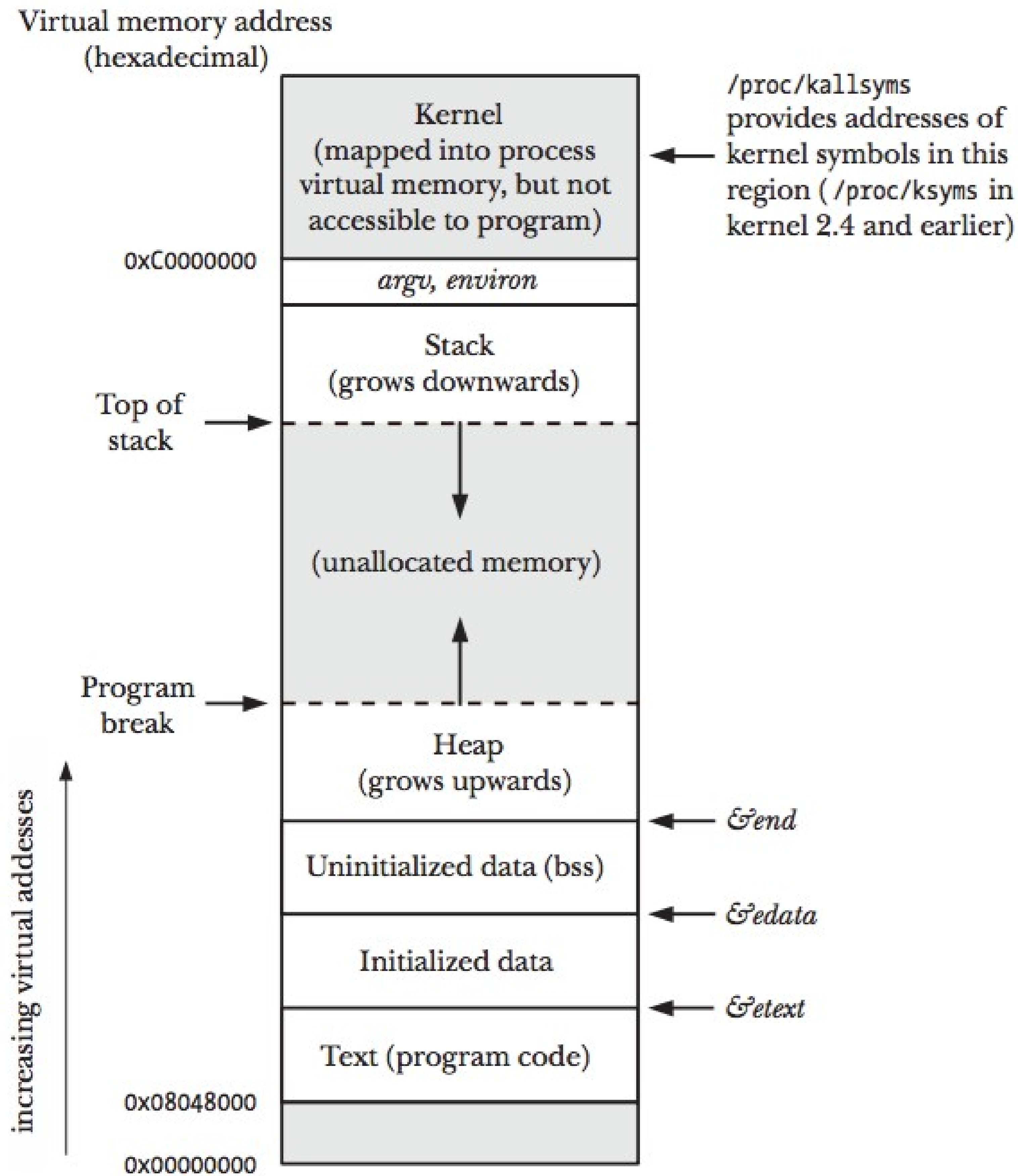
Memory Allocator

Help Session

Topics REVISED

- Overview of Memory Management
- Basics / Getting Started
- Project Structure
- GNU Perf Utils & Tools
- Logistics / Grades
  - Testing Framework
- Heap Inspector Tool

Overview of Memory Management



Virtual memory address
(hexadecimal)
Kernel
(mapped into process
virtual memory, but not
accessible to program)
/proc/kallsyms
provides addresses of
kernel symbols in this
region (/proc/ksyms in
kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack
(grows downwards)
Top of
stack
(unallocated memory)
Program
break
Heap
(grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addresses

The Heap

- Persistent, unmanaged memory granted to processes
- Sometimes memory allocation strategies will be coupled with Garbage Collector (GC)
  - Hold on to memory for too long: memory leak
  - Free memory too early: memory corruption
- Usually managed by malloc() in libc

brk() and sbrk()

- sbrk()
  - Increases the size of the data segment
- brk()
  - Sets the ending address of the data segment

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0x7facdc778000

brk(0x1830000)

munmap(0x7facdc7a1000, 4096)
```

The Goal

- Lots of allocators out in the wild
  - Some general purpose, some for specific applications
  - Google’s TCMalloc
- Time versus space tradeoff
- *Instantly* know an available place in the heap to fit the allocation request *exactly*

Basics / Getting Started

Getting Started

- Fork the repo
  - https://git.cs.vt.edu/cs3214-staff/malloclab
  - Set to private
  - Similar to shell: grading on git usage

Functions

- You will write these functions, each following the conventions of malloc():

```
int mm_init (void);
void * mm_malloc (size_t size);
void mm_free (void * ptr);
void * mm_realloc(void * ptr, size_t size);
```

- Like the real malloc(), you must be able to handle a variety of allocation sizes

Getting Started

- Writing one file, mm.c
- Write helper functions to perform pointer math
  - Makes debugging easier
- Review the provided sample implementation, mm-gback-implicit.c

Provided Functions

- Extend the heap bytes and return the start address:

```
void * mem_sbrk(int incr);
```

- Return address of the first heap byte:

```
void * mem_heap_lo(void);
```

- Return address of the last heap byte:

```
void * mem_heap_hi(void);
```

Provided Functions

- Returns the current size of the heap in bytes:

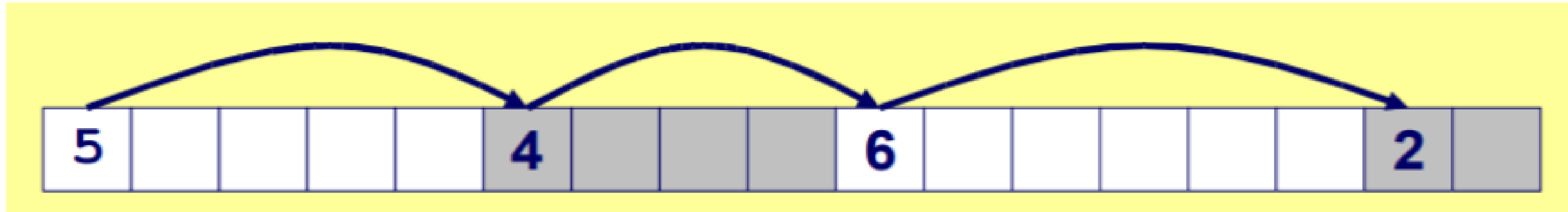
```
size_t  mem_heapsize(void);
```

- Returns the system's page size in bytes (4K on Linux systems):

```
size_t  mem_pagesize(void);
```

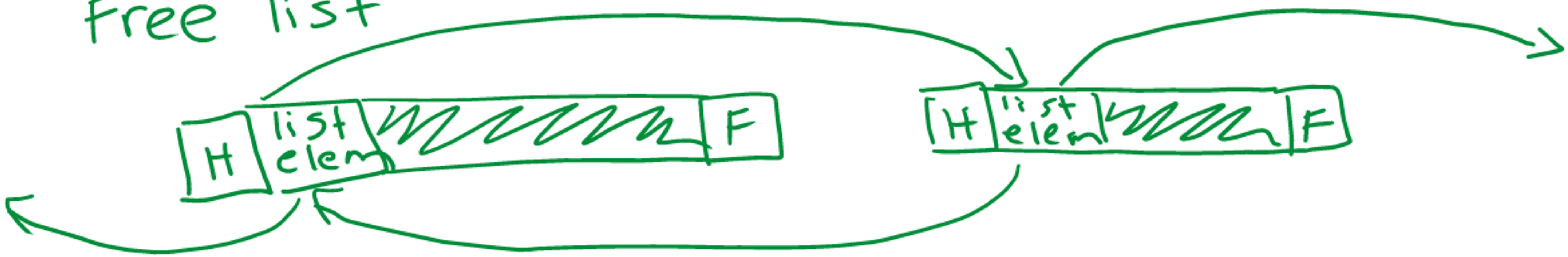
mm-gback-implicit.c

- Sample solution which might be a good starting point
  - Be mindful of word conversions
  - Determine if size should be inclusive to the boundary tag
- Know the design decisions / function preconditions you are inheriting!

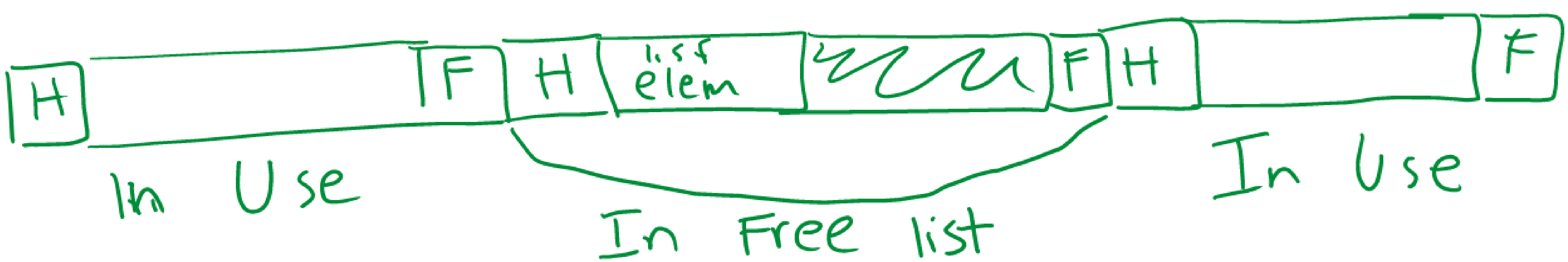


Explicit

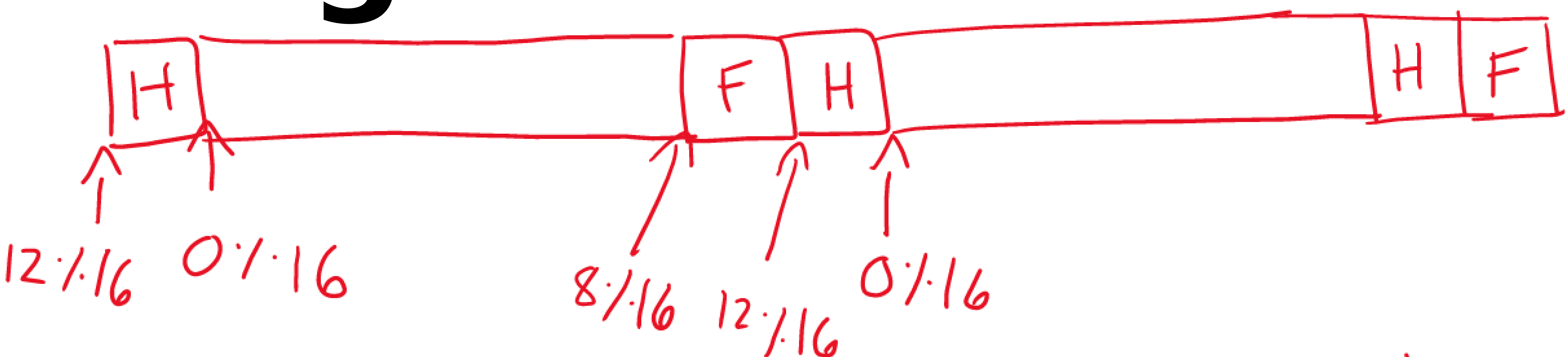
Free list



Heap



Alignment



All payloads must be aligned at 0 %16.

Since headers and footers are 4 bytes each, need to pad and align accordingly

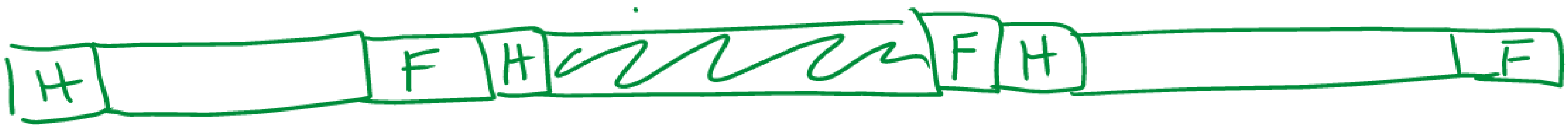
Fragmentation / Coalescing

Free block of size 128

User makes malloc(16) call

Split the free block so that the unused space from the 16 byte payload can still be used for other data.

Fragmentation / Coalescing



The user makes a free call to the last block

We now have two adjacent free blocks. If we combine them together, the free block is able to hold larger blocks of data.

No header and footer in middle

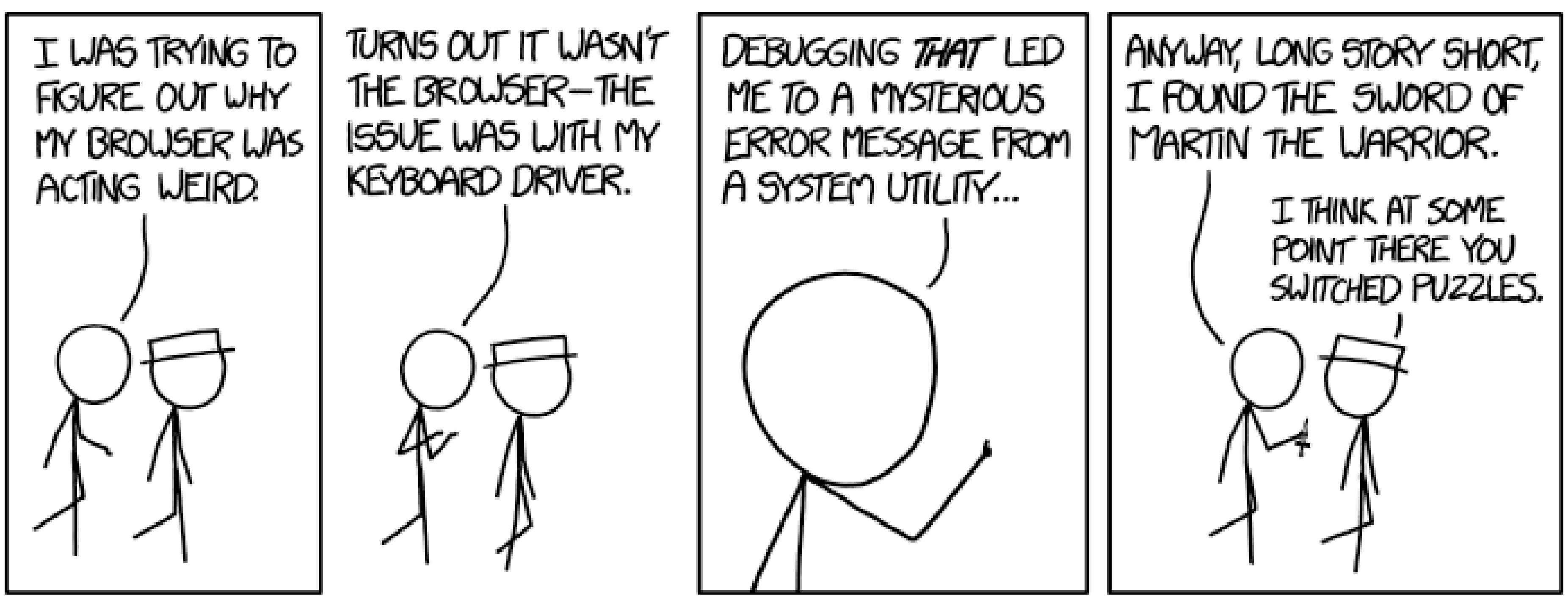
Project Structure & Suggestions

Suggestions

- Consider performance implications from the start
  - Can I only include this structure in the tag in certain situations?
  - Do I have all the fields I need?
  - Am I willing to trade space for performance here?
- Consider edge conditions
- Define suitable C structures to minimize casting
- Use assert() statements liberally
  - Explicitly state pre and post conditions

Assert Statements

- Figure out where the bug occurs, rather than a side effect of the bug
  - Is find_fit *actually* returning a block?
  - Is the block *actually* not in a list?



Suggestions

- Use void * pointer arithmetic encapsulated in helper functions
- Do your implementation in stages
- Use a profiler such as gprof
- Start early
  - Try different implementation strategies
  - Performance will take the majority of the time
  - Always think about design!
- Alter the CFLAGS to include -g

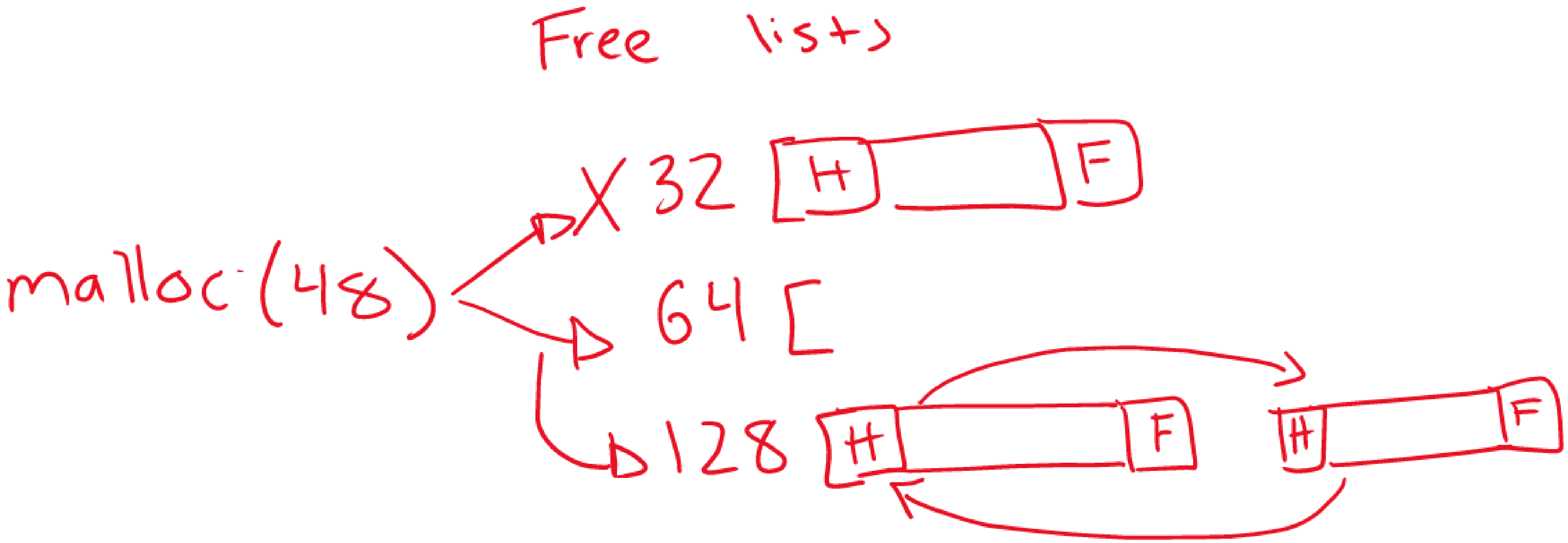
Suggestions

- Make sure the start of all memory addresses returned from mm_malloc() are aligned correctly:

```
size = (size + ALIGNMENT - 1) & ~(ALIGNMENT - 1);
```

- Be careful about lists and traversals
- As with thread-pool: iterate and empirically test
  - Explicit list
  - Segregated list
  - Red-Black tree

Segregated Free Lists



We know a block of size 48 won't fit in a 32-byte block, so we look through the 64-byte block free list. Since that list is empty, we go on to the next biggest list no until we find a free block or determine no free blocks.

mm_checkheap()

- Sanity checker that can print out the blocks in the heap
  - Written for your benefit
  - Based off of an implicit list traversal
- Remember to remove during test!

GDB

- Dump the contents of the heap out
- Can now use the heap inspector instead!

```
(gdb) define xxd
>dump binary memory dump.bin $arg0 $arg0+$arg1
>shell xxd dump.bin
>end
(gdb) xxd mem_heap_lo() 200
```

```
0000000: 0100 0000 0104 0000 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Allocation Patterns

- Empirically test!
- Many smaller requests which are short lived
- Fewer larger requests which are held for longer periods of time

GNU Perf Utils & Other Tools

Perf Utils

- Add the -pg flag during compilation
  - This will create a file gmon.out when your code is run
- Check the output:

```
$ gprof mdriver gmon.out > prof_output
```

- Remember to remove this flag during performance testing!

Perf Utils Output

– Shows each time a function is called, along with a call graph, to identify bottlenecks:

```
Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
 59.20      3.25     3.25                             eval_mm_valid_inner
 12.20      3.92     0.67    182955     0.00     0.00  add_range
 12.02      4.58     0.66   1696347     0.00     0.00  mm_malloc
  8.38      5.04     0.46 288964584     0.00     0.00  list_next
  7.10      5.43     0.39 300120288     0.00     0.00  list_end
  0.36      5.45     0.02   1696347     0.00     0.00  mm_free
  0.36      5.47     0.02       330     0.06     0.06  mm_init
  0.18      5.48     0.01  11155704     0.00     0.00  list_begin
  0.18      5.49     0.01       330     0.03     4.33  eval_mm_speed
```

Structs / Bitfields

- Allow information to densely packed, if size constraints are known
  - Performance considerations?

```
struct packed_data {
    unsigned int in_use:1;
    unsigned int size:31;
    char payload[0];
}
```

Project Logistics

Logistics

- Please submit code that compiles
- Test using the driver before submitting!
  - Don’t just run the tests individually
- When grading, these tests will be run 3-5 times, and if you crash a single time, it’s considered failing
- Parts:
  - Correctness
  - Performance
  - Multi-threading (extra-credit)
    - Refer to mm_ts.c

Logistics: Grading

- Grade breakdown (100 points total):
  - 40 points for correctness (**MIN REQUIREMENT)**
  - 40 points single threaded performance
    - Space utilization
    - Throughput
  - 20 points for documentation/style/git
    - At least 5 assert statements

- Extra credit: Additionally support multithreading

Test Driver

```
cd malloclab/
./mdriver
```

- Run with -v / -V for verbose output
- Run with -f to customize traces
- Run with -s vary allocation size

Performance

Perf index = 44 (util) + 0 (thru) = 45/100

- Throughput
  - Number of requests per second
- Utilization
  - How much space the heap has been expanded by versus the space user data takes
  - Overhead
  - Fragmentation

Results for mm malloc:

trace	name	valid	util	ops	secs	Kops
0	amptjp-bal.rep	yes	99%	5694	0.010260	555
1	cccp-bal.rep	yes	99%	5848	0.009955	587
2	cp-decl-bal.rep	yes	99%	6648	0.016045	414
3	expr-bal.rep	yes	100%	5380	0.010824	497
4	coalescing-bal.rep	yes	67%	14400	0.000665	21659
5	random-bal.rep	yes	92%	4800	0.007215	665
6	random2-bal.rep	yes	92%	4800	0.007079	678
7	binary-bal.rep	yes	55%	12000	0.173721	69
8	binary2-bal.rep	yes	51%	24000	0.326384	74
9	realloc-bal.rep	yes	27%	14401	0.100668	143
10	realloc2-bal.rep	yes	34%	14401	0.003196	4506
Total			74%	112372	0.666013	169

Test Trace Files

```
3000000        // Heap size
2847         // Unique identifiers
5694         // Number of operations
1
a 0 2040
f 0
```

- Located in /home/courses/cs3214/malloclab/traces
- Allocation sizes and references

Reference

- [L-MEM1] Dynamic Memory Management (malloc/free)
  - Implicit vs Explicit
  - Fragmentation
  - Coalescing Policies

Heap Inspector Demo

Questions?

Thank you for attending!