CPSC 261 – BASICS OF COMPUTER SYSTEMS

2017 Winter Term 2

Module 1: Memory models, C, pointers and assembly

1

LEARNING GOALS

- At the end of this module, you will be able to:
  - Explain the purpose of simple functions written in x86-64 assembly language.
  - Explain how a C program uses memory.
  - Write simple C functions that allocate and deallocate memory dynamically.
  - Recognize and fix common errors related to dynamic memory allocation.

MODULE SUMMARY

- Module Summary
  - The x86-64 assembly language
  - C and memory
  - Addresses and pointers
  - Dynamic memory allocation
  - Common programming mistakes with pointers
  - How a process' memory is organized
  - A look at the runtime stack
  - Implementing a dynamic memory allocator

AN ASIDE: LAB INFORMATION

- Linux is a free Unix-like operating system
  - Unix was invented at Bell Labs starting in 1969.
  - By 1973 it was in active use.
  - The original source code was encumbered with licensing concerns.
  - Linux is a re-implementation of the ideas.

AN ASIDE: IN-CLASS CODE

- All longer examples used in class will be available from a Stash repo.
- To create a local copy of the repo, use the command:
  - git clone https://stash.ugrad.cs.ubc.ca:8443/git/CS261_2017W2/examples.git
- It will create a directory examples with the examples released so far (each in its own subdirectory).
- To update the repo when new examples are added to it, change to directory examples, and then use
  - git pull

THE X86-64 ASSEMBLY LANGUAGE

- We will use the x86-64
  - 16 general purpose registers (1 stack pointer, 1 frame pointer when needed).
  - instructions: mov, add, sub and lots more.
  - data sizes of 1, 2, 4, 8 (and occasionally 16) bytes.
  - parameters in registers unless there are too many (by convention).

THE X86-64 ASSEMBLY LANGUAGE

- How do you learn a new assembly language?
  - Read a short description (32 bits x86 only)
    - http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
  - Read a longer description
    - http://csapp.cs.cmu.edu/public/1e/public/docs/asm64-handout.pdf
    - https://software.intel.com/en-us/node/181178
  - Or use a C compiler and read the assembly language code it produces.

X86-64 INSTRUCTIONS

- Instruction types:
  - Data movement (mov)
  - Arithmetic and logical (add, sub, mul, div, inc, and, or, not)
  - Comparisons (test, cmp)
  - Branches (jXX, jmp, loop)
  - Function call related (push, pop, call, ret)
  - Priviledged (iret, hlt, cli, sti)
    - Will be covered in more detail later in the course

X86-64 INSTRUCTIONS (CONT.)

- The instruction's last letter indicates operands' size
  - b = 8 bits, l = 32 bits, q = 64 bits for integers
  - s = 32 bits, d = 64 bits, t = 128 bits for floating points
- The register's first letter indicates its size
  - ah, al: 8 bits
  - ax: 16 bits
  - eax: 32 bits
  - rax: 64 bits
- Not all sizes are available for all registers.

X86-64 ADDRESSING MODES

- Addressing modes:
  - register direct: %rax
  - indirect: offset(%rax)
  - indexed: offset(%rax, %rbx)
  - indexed scaled: offset(%rax, %rbx, 4)
  - absolute: address
  - immediate: $value

ASSEMBLY CONVENTIONS

- There are conventions:
  - What registers are used for what?
  - How are values passed to functions?
  - How are values returned from functions?
  - What if a value is too large to fit in a register?
- Conventions may vary by:
  - CPU architecture
  - Operating system
  - Compiler

X86-64 CALLING CONVENTIONS

- Stack pointer: %rsp
- Arguments passed in registers, in this order: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Additional arguments passed in stack
- Returning value passed in %rax
  - Use %rax and %rdx if returning value is 128 bits
  - Use memory for larger values
- Caller-save registers (caller's responsibility to save before calling other functions): %rax, %r10, %r11 + arguments
- Callee-save registers (function must restore value before returning if changed): %rbx, %rbp, %r12, %r13, %r14
  - In x86-64: %r15*

SOME EXAMPLES

- Let us look at examples (in the examples repo):
  - odd.c
  - sum.c

REVIEW: ADDRESSES AND POINTERS

- Consider the following (incomplete) piece of code:

```
int *a, b;
int **c;

...

d = &c;
x = a + 7;
y = c[3];
```

- Describe the type of value stored in each variable without using the word "pointer".

REVIEW: ADDRESSES AND POINTERS

- What does the following C function print?

```
void do_something() {
  char mtl[] = "     ";
  char *qc, **cdn = &qc;

  mtl[0] = 'Y';
  mtl[4] = '\0';
  qc = mtl + 2;
  *qc = 'A';
  qc[-1] = 'P';
  *cdn = qc + 1;
  **cdn = 'Z';
  *(qc - 1) = 'E';
  printf("%s\n", mtl);
}
```

REVIEW: ADDRESSES AND POINTERS

- Rewrite the following piece of code using arrays to make it more readable.

```
int confusing(long *p, int n)
{
  long *q = p + n;
  while (n > 0 && *p++ == *--q)
  {
    n -= 2;
  }
  return n <= 0;
}
```

LOCAL VARIABLES AND ARGUMENTS

```
void b (int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
}
```

- Can `l0`, `l1`, `a0`, `a1` be allocated statically?
  - In other words, can their static address be determined at compilation time?
  - What about recursion?

LOCAL VARIABLES AND RECURSION

- How many different versions of `n` are there?

```
void a (int n) {
  if (n == 0) return;
  a(n - 1);
  printf("%d\n", n);
}
```

- What if there is no apparent recursion?

```
void b (int n) {
  int l = n * n;
  c(l - 1);
}
```

  - What if c() calls b()?

LIFE OF A LOCAL VARIABLE/ARGUMENT

- Scope
  - Local variables are only accessible within declaring procedure
  - Each execution has its own private copy
- Lifetime
  - Allocated when procedure starts
  - “Freed” when procedure returns (in most languages)

PROCEDURE ACTIVATION

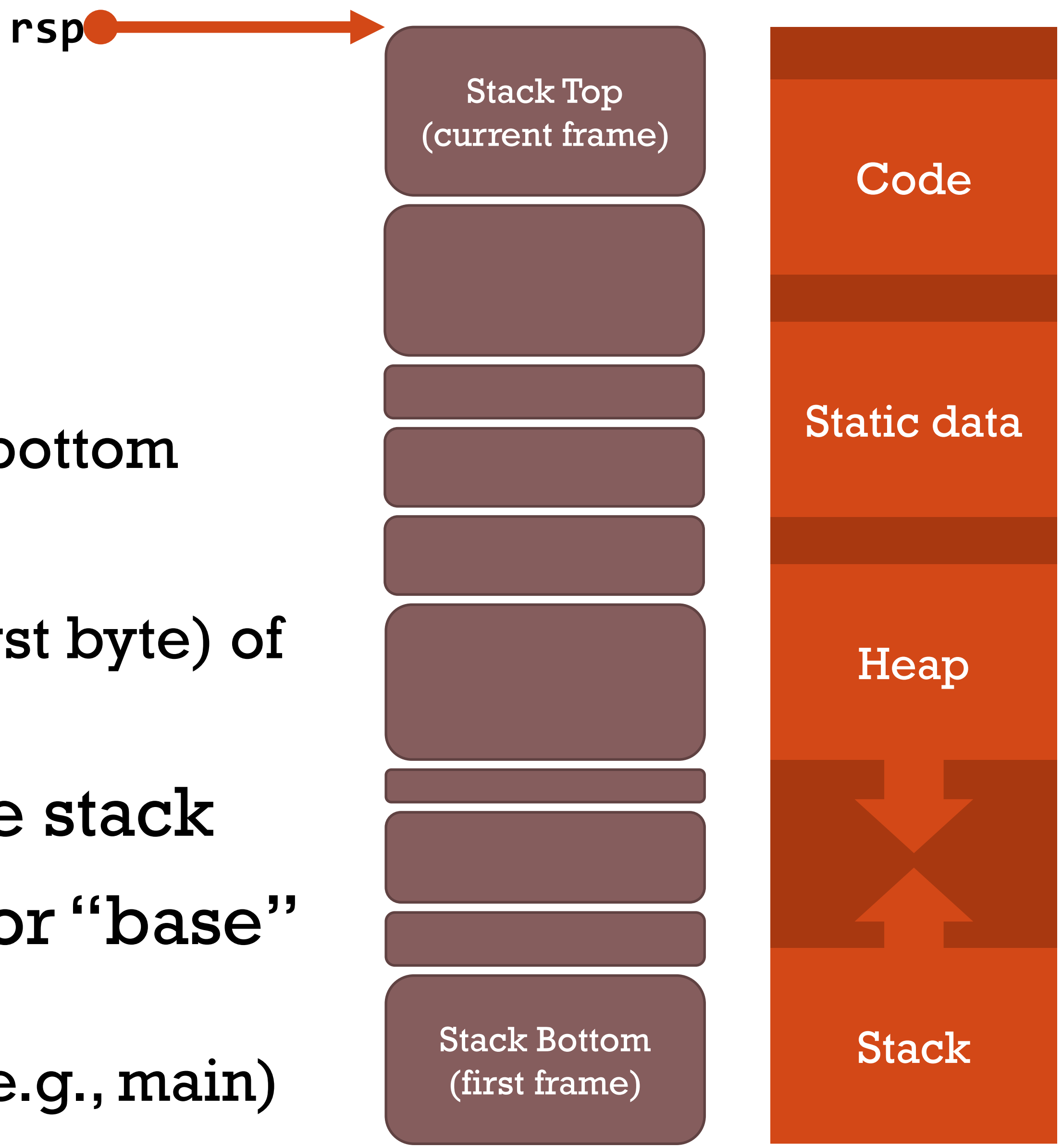
- Activation: execution of a procedure
  - Starts with procedure is called, ands when it returns
  - There can be many activations of same procedure alive at once
- Activation Frame
  - memory that stores activation's state
  - Includes local variables and arguments
- Should we allocate activation frames from the heap?
  - Call `malloc()` to create frame on procedure call?
  - Call `free()` on procedure return?

HEAP VS ALLOCATION FRAMES

- Order of frame allocation and deallocation is special
  - freed in reverse order of allocation
- Simple allocation for frames:
  - Reserve big chunk of memory for all frames
  - Initial address known
  - Simple, cheap allocation: add or subtract from a pointer
- Questions
  - What data structure is this like?
  - What restriction do we place on lifetime of local variables?

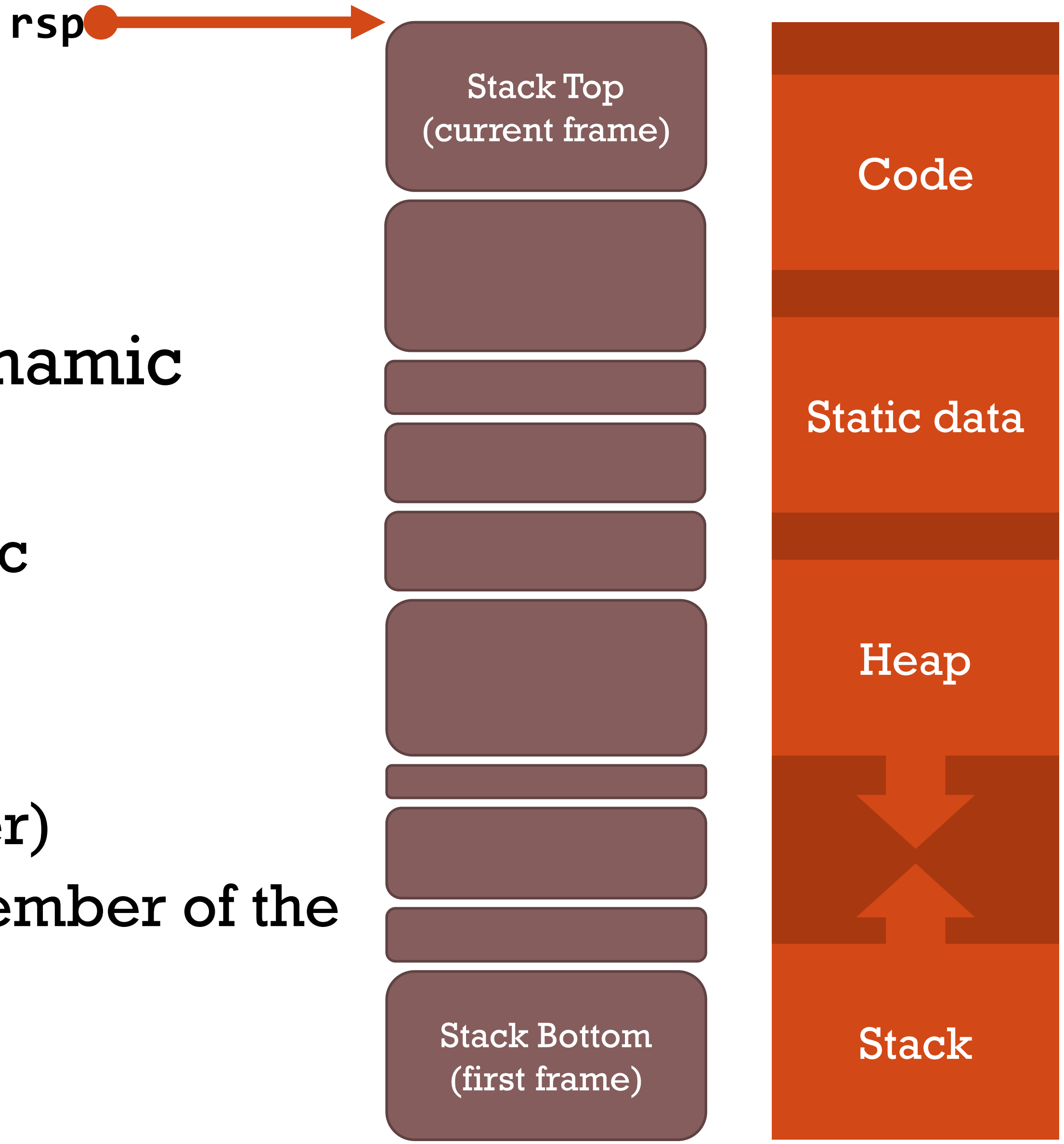
RUNTIME STACK

- Stack of activation frames
  - Stored in memory, grows up from bottom
- Stack pointer
  - Stores base address (address of first byte) of current frame
- Current frame is the "top" of the stack
- First activation is the "bottom" or "base" of the stack
  - Local variables of initial function (e.g., main)



VARIABLE ADDRESSES

- Value of the stack pointer is dynamic
- Local variables and arguments
  - Size of each frame is (usually) static
  - Offset from stack pointer is static
- Each frame is like a struct
  - Top of frame is in rsp (stack pointer)
  - Each variable in procedure is a member of the struct



WHAT IS STORED IN THE ACTIVATION FRAME?

- Local variables
- Arguments
  - Some architectures use registers for arguments
- Return address
- Other saved registers
  - Called function may change register values
  - Values that must be kept after call are saved

```
saved registers…
local variables…
return address
arguments…

saved registers…
```

SOME IMPLICATIONS

- What is the value of l in foo when it is active?

```
void goo() { int x = 3; }        goo();
void foo() { int l; }            foo();
```

- What is wrong with this?

```
int *foo() {
  int l;
  return &l;
}
```

PROCESS MEMORY ORGANIZATION

- When a program is executed:
  - Space is allocated for the shared libraries it needs
  - Instructions and initialized data are loaded in memory
  - Space is reserved for uninitialized data
- The stack and the heap are set up
  - The stack is managed by the compiler's code
  - The heap is managed by the user's program
- On a Linux system, we can look at `/proc/`*`pid`*`/maps` to the how memory is used for process *pid*.

PROCESS MEMORY EXAMPLE

```
00400000-00401000 r-xp 00000000 08:08 5244791            /home/patrice/fib
00600000-00601000 r--p 00000000 08:08 5244791            /home/patrice/fib
00601000-00602000 rw-p 00001000 08:08 5244791            /home/patrice/fib
7f86a3122000-7f86a32dd000 r-xp 00000000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a32dd000-7f86a34dd000 ---p 001bb000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a34dd000-7f86a34e1000 r--p 001bb000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a34e1000-7f86a34e3000 rw-p 001bf000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a34e3000-7f86a34e8000 rw-p 00000000 00:00 0
7f86a34e8000-7f86a35ed000 r-xp 00000000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a35ed000-7f86a37ec000 ---p 00105000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a37ec000-7f86a37ed000 r--p 00104000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a37ed000-7f86a37ee000 rw-p 00105000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a37ee000-7f86a3811000 r-xp 00000000 08:02 1308526    /lib/x86-64-linux-gnu/ld-2.19.so
7f86a39e5000-7f86a39e8000 rw-p 00000000 00:00 0
7f86a3a0e000-7f86a3a10000 rw-p 00000000 00:00 0
7f86a3a10000-7f86a3a11000 r--p 00022000 08:02 1308526    /lib/x86-64-linux-gnu/ld-2.19.so
7f86a3a11000-7f86a3a12000 rw-p 00023000 08:02 1308526    /lib/x86-64-linux-gnu/ld-2.19.so
7f86a3a12000-7f86a3a13000 rw-p 00000000 00:00 0
7fffc88a3000-7fffc88c4000 rw-p 00000000 00:00 0          [stack]
7fffc88cf000-7fffc88d1000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

DYNAMIC MEMORY ALLOCATION

- Allocating memory
  - We use:
    - void *malloc(size_t size);
    - void *calloc(size_t count, size_t size);
  - Typical usage:
    - Given
      - type *ptr;
    - To allocate memory for one object:
      - ptr = malloc(sizeof(type));
    - To allocate memory for an array of objects:
      - ptr = calloc(count, sizeof(type));

DYNAMIC MEMORY ALLOCATION

- Deallocating memory:
  - We use:
    - void free(void *ptr);
  - Example:
    - free(ptr_to_type);
  - This works for a single object, as well as for arrays

DYNAMIC MEMORY ALLOCATION

- Example: allocate and return a string with the English alphabet.

```
char *create_alphabet_string()
{



}
```

DYNAMIC MEMORY ALLOCATION

```
char *create_alphabet_string() {
    char *my_string = malloc(27 * sizeof(char));
    int i;
    for (i = 0; i < 26; i++) {
        my_string[i] = 'a' + i;
    }
    my_string[26] = '\0';
    return my_string;
}
```

DYNAMIC ALLOCATION: USAGE

- Example: array that grows as data grows
  - When adding an element, if array is full, allocate bigger space and copy old data to new space
- Example: linked list
  - Each element is a struct that contains a pointer to next element
  - Lab assignment #2

DANGLING POINTERS

- Dangling pointer: pointer that does not point to actual usable memory
- Examples:
  - Uninitialized pointer
  - Multiple pointers to same location, one is freed and the other is still in use
  - Calling free on the same memory twice
  - Function returns pointer to local variable
- Good practices to avoid dangling pointers:
  - Initialize all pointers to valid data (e.g., NULL)
  - If needed, implement reference counting

MEMORY LEAKS

- Memory leak: memory has been allocated but not deallocated
  - Usually caused by lost reference to dynamically allocated data
- Examples:
  - Function allocates memory, then returns without saving or returning memory
  - Function returns dynamically allocated memory, but return value is ignored
  - Last pointer to allocated space is changed to different value

AVOIDING MEMORY LEAKS

- Good programming practices to avoid leaks:
  - When possible, allocate and free data in same function
  - Check old value of a pointer before changing it
    - Point it to NULL if not assigned to anything
  - If needed, implement reference counting

OTHER COMMON MISTAKES WITH POINTERS

- Buffer overflow: using more data than allocated
  - Can be a problem with global and local arrays as well
  - Can be caused by off-by-one errors (e.g., not counting the string termination byte)

IMPLEMENTING A DYNAMIC MEMORY ALLOCATOR

- So far we have been using malloc/calloc and free
- But how are they implemented?
  - How is available memory maintained?
  - How to keep track of freed blocks?
  - When is it ok to reuse a block?

MEMORY ALLOCATOR REQUIREMENTS

- Handling arbitrary sequences of requests
  - The allocator can not control the requests for malloc/free
- Making immediate responses to requests
  - The allocator can not wait to process several requests at once, even if it would be more efficient
- Using only the heap for its data structures
  - We can use a constant amount of additional space only
- Not modifying allocated blocks
  - The user program assumes their contents won't change
  - It also assumes its location won’t change

MEMORY ALLOCATOR IMPLEMENTATION

- Implementation issues:
  - *Placement*: when malloc() is called, how to we find a free block that will be used to satisfy the request?
  - *Splitting*: if we only need part of a free block to satisfy a request, what do we do with the rest?
  - *Coalescing*: do we merge a newly free()'d block with adjacent free blocks?
- These issues arise in all implementations

MEMORY ALLOCATOR IMPLEMENTATION

- Goals:
  - Maximizing throughput
    - We want to respond to requests quickly
    - So we need to use simple data structures
  - Maximizing memory utilization
    - We want to avoid internal and external fragmentation

FRAGMENTATION

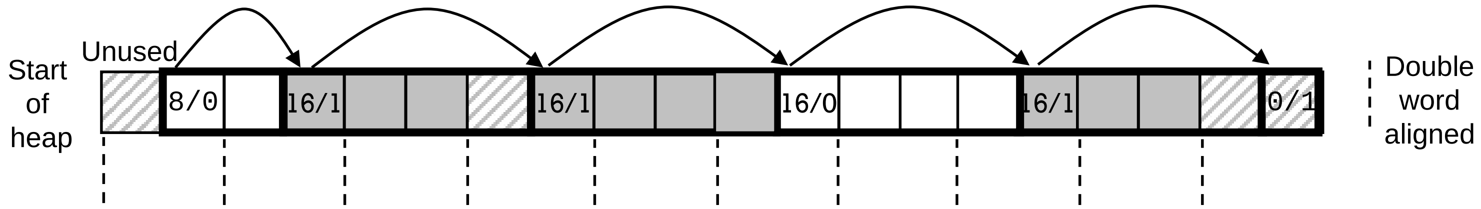
- Internal fragmentation
  - Most allocators impose a minimum size on the blocks they return to a process
    - Because of alignment requirements (pointer addresses must be a multiple of 4, or 8)
    - Because the allocation needs to store information inside the blocks once they are freed; so the blocks must be large enough
  - Hence a request for a very small amounts of memory returns a larger block than that requested
  - Some space inside the block is wasted

FRAGMENTATION

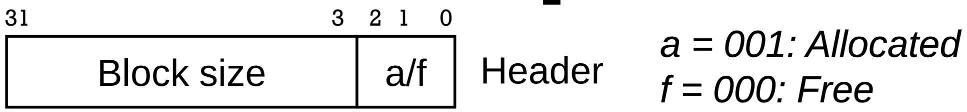
- External fragmentation
  - Unlike files on disk, the block returned by malloc() must consist of consecutive memory locations
  - Sometimes there may be enough free space in total, but no free block is large enough to satisfy the request
  - The more calls to malloc() and free() have been made with requests for different size blocks, the more external fragmentation is a problem

IMPLICIT FREE LIST

- A simple implementation: the implicit free list.



- We have a linked list of blocks.
  - The list contains both occupied and free blocks.
  - Each block contains its size.
  - Each block knows if it's free or occupied:

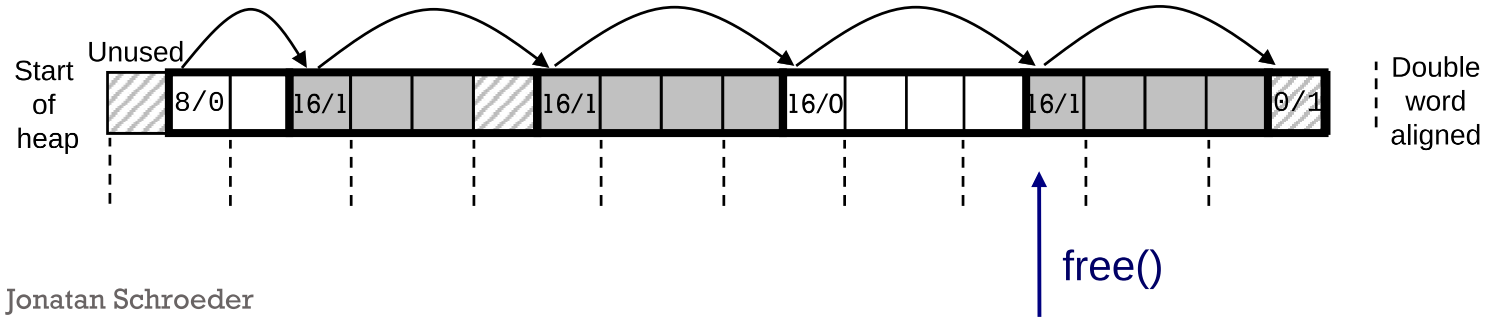


IMPLICIT FREE LIST: SPLITTING

- If the block used is larger than the requested size, we can
  - use the whole block (increases internal fragmentation)
  - divide the block in two (may end up with many small blocks)

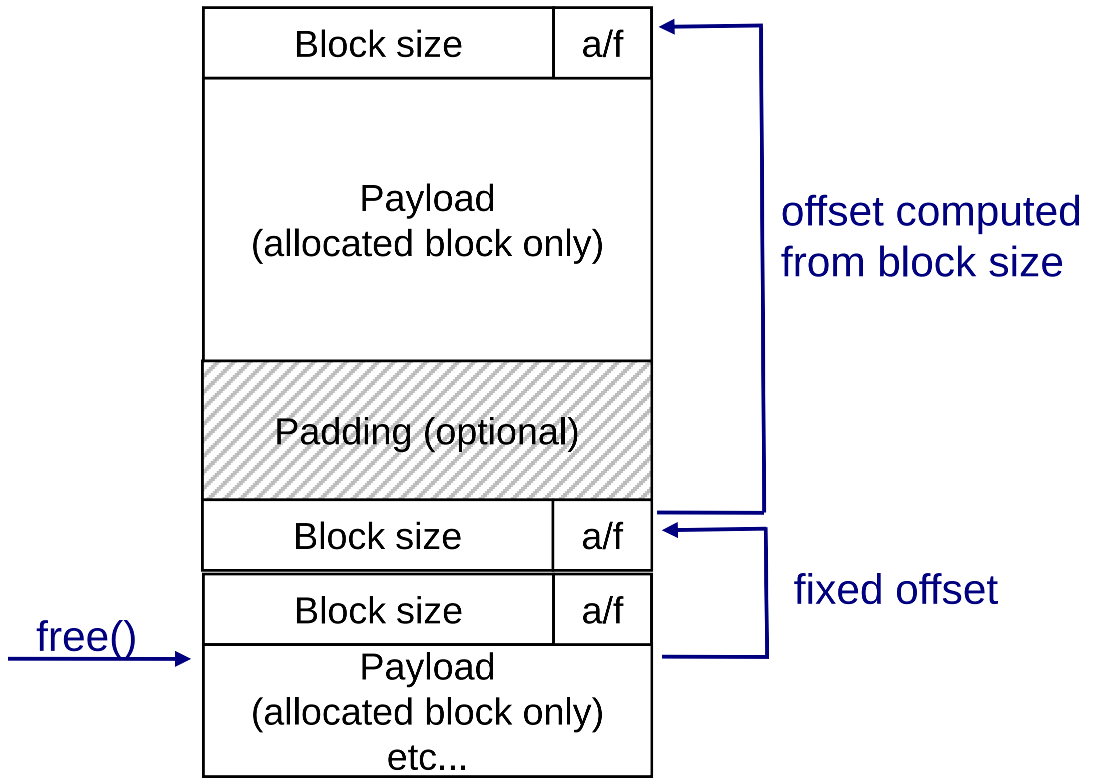
IMPLICIT FREE LIST: COALESCING

- When freeing a block, merge adjacent free blocks
- Avoids ending up with lots of small free blocks all adjacent
- We can do this on every free
  - Advantage: simpler
  - Disadvantage: free operation becomes slower
  - Alternative: wait until an allocation request fails
- Problem: how to find adjacent blocks



IMPLICIT FREE LIST: COALESCING

- We store the block size at the end of the block also:



IMPLICIT FREE LIST: PLACEMENT

- Placement:
  - *First-fit*: return the first free block that is large enough
    - retains large free block near end of the list
    - Disadvantage: search time if too many small blocks
  - *Next-fit*: similar but start searching from the last allocated block
    - Advantage: faster search time
    - Disadvantage: worse memory utilization than first-fit
  - *Best-fit*: find the free block whose size is closest to the requested size
    - Advantage: optimal use of memory
    - Disadvantage: slower

EXPLICIT FREE LIST

- *Explicit free list*: uses the payload in free blocks to point to other free blocks
  - Block search is faster (don't need to check blocks in use)
  - Doubly-linked list
  - Disadvantage: minimum payload size must be enough for two pointers
- Linked-list order (where to pointers point to)
  - Last-in First-out: free blocks go at the beginning of the list
    - free() takes constant time
  - In address order: pointers list blocks in address order
    - free() requires linear time (must search previous/next free blocks)
    - We get slightly better memory utilization (search is in memory order)

SEGREGATED FREE LIST

- *Segregated free list*: one linked list per block size
  - Blocks point to other blocks of similar size
- One approach (segregated fits):
  - malloc():
    - find a block large enough,
    - split it if desired, and insert the other piece in the appropriate free list.
  - free():
    - coalesce with adjacent free blocks if possible,
    - store the new free block in the appropriate free list.

SEGREGATED FREE LIST (CONT.)

- Advantages of segregated free lists:
  - Searching for free blocks is more efficient
    - We are only searching part of the heap
  - Memory utilization improves
    - First-fit search with a segregated list approximates a best-fit search of the entire heap
- The GNU malloc package, part of the standard C library on all Linux systems, uses segregated free lists