CS 61c: Great Ideas in Computer Architecture

Memory Management in C

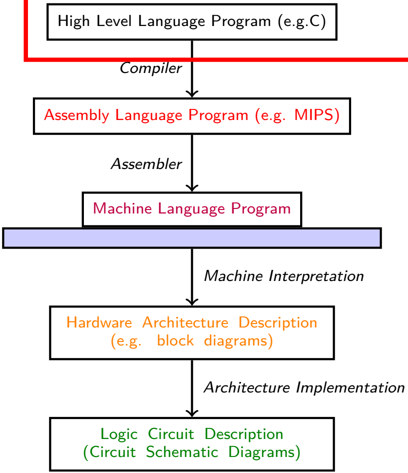
Instructor: Alan Christopher

June 26, 2014

Review

- Arrays
  - Can traverse using pointer or array syntax
  - Use null-terminated `char[]` for strings
- Pointer arithmetic moves the pointer by the size of the thing it's pointing to
  - No need for the programmer to worry about it

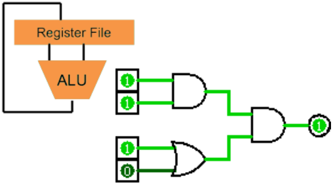
Great Idea #1: Levels of Representation/Interpretation




```
temp = v[k];
v[k] = v[k+1]
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
1000 1100 0100 1000 0000 0000 0000 0000
1000 1100 0100 1001 0000 0000 0000 0100
1010 1100 0100 1001 0000 0000 0000 0000
1010 1100 0100 1000 0000 0000 0000 0100
```

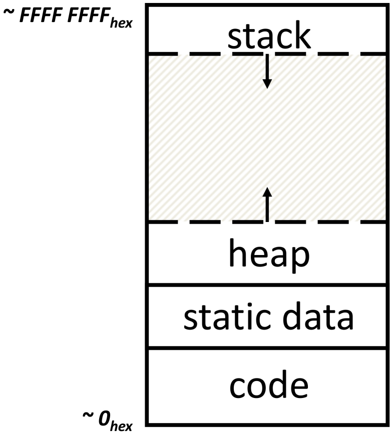


Outline

- Memory Layout
  - In C
  - Stack Mem
  - Static and Code Data
- Administrivia
- Dynamic Memory Allocation
  - Heap
  - Common Problems
  - Memory Management
- C Wrap-up
  - Linked List Example

Memory Layout

- Program's address space contains 4 regions:
  - **Stack:** local variables, grows downward
  - **Heap:** Space requested via `malloc()`, grows upward
  - **Static Data:** Global and static variables. Does not change size.
  - **Code:** Loaded when program starts, does not change
- OS responsible for detecting accesses to unallocated region.



Which variables go where

- **Static:**
  - Declared outside a function
- **Stack:**
  - Declared inside a function
  - note: `main()` is a function
  - Freed on function return
- **Heap:**
  - Dynamically allocated (e.g. with `malloc()`)

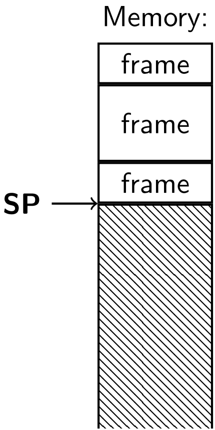
```
#include <stdio.h>

int varGlobal;

int main() {
  int varLocal;
  int *varDyn =
    malloc(sizeof(int));
}
```

The Stack

- Each stack frame is a contiguous block of memory holding the local variables of a single procedure
- A stack frame includes:
  - Location of caller function
  - Function arguments
  - Space for local variables
- Stack pointer (SP) tells where the lowest (current) stack frame is
- When a function returns its stack frame is thrown out, freeing memory for future function calls



An Example

- Last in, First out (LIFO) data structure

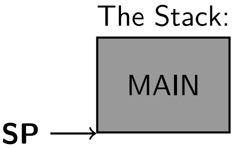
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

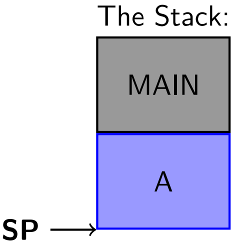
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

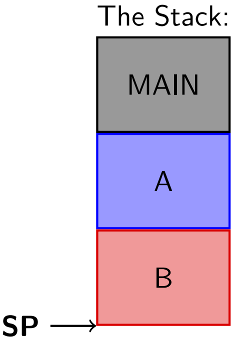
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

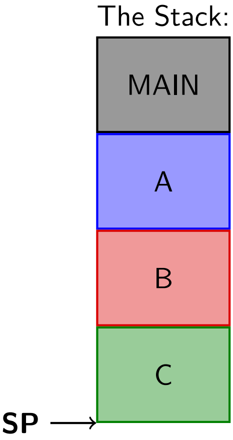
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

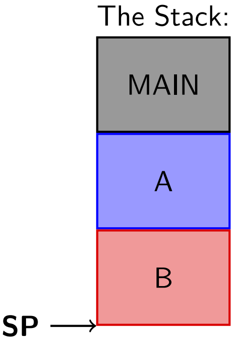
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

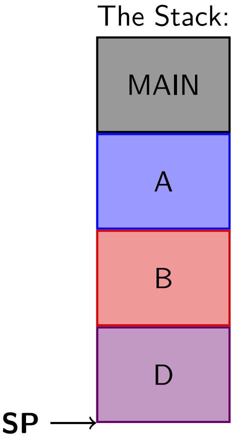
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

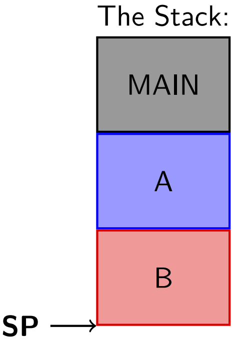
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

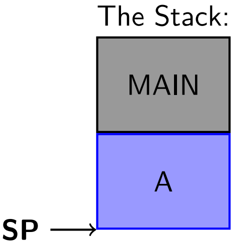
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



An Example

- Last in, First out (LIFO) data structure

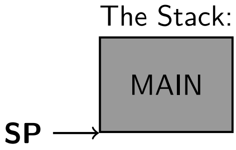
```
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    d(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```



Stack Misuse

- **Never** return pointers to locally allocated memory, e.g.

```
int *getPtr() {
    int y = 3;
    return &y;
}
```

  - Compiler will warn you if you do this, don't ignore it

- Things can get really wonky if you do this (Boardwork):

```
int main() {
    int *stackAddr, content;
    stackAddr = getPtr();
    content = *stackAddr;
    printf("%d", content\n); /* 3 */
    content = *stackAddr;
    printf("%d", content\n); /* -1216751336 */
}
```

Static and Code

Static:

- Place for variables that persist
  - Good for data that never expands, shrinks, or goes stale
  - E.g. String literals, global variables
- Size is constant, but contents can be modified

Code:

- Where the executable data is stored
  - We can represent anything with bits, including programs. More on how to do that later
- Does not change size
- Contents usually not allowed to be modified

Question: Which statement below is FALSE? All statements assume that each variable exists.

```
void funcA(){int x; printf("A");}
void funcB() {
    int y;
    printf("B");
    funcA();
}
void main() {char *s = "s"; funcB();}
```

(blue) `x` is at a lower address than `y`
(green) `x` and `y` are in adjacent stack frames
(purple) `x` is at a lower address than `*s`
(yellow) `y` is in the 2nd frame from the top of the Stack

Question: Which statement below is FALSE? All statements assume that each variable exists.

```
void funcA(){int x; printf("A");}
void funcB() {
    int y;
    printf("B");
    funcA();
}
void main() {char *s = "s"; funcB();}
```

(blue) x is at a lower address than y

(green) x and y are in adjacent stack frames

(purple) x is at a lower address than *s

(yellow) y is in the 2nd frame from the top of the Stack

Outline

Administrivia

- HW1 still due Sunday
- Project 1 released
  - Start early!
  - Start early!
  - Did I mention to start early? You should start early.

Outline

Dynamic Memory Allocation

- Sometimes you don't know how much memory you need beforehand
  - e.g. input files, user input
- Dynamically allocated memory goes on the *heap* – more permanent than the stack
- Needs as much space as possible without interfering with the stack
  - this is why we start the stack at the top of memory, and the heap towards the bottom

Allocating Memory

- Three functions for requisition memory: `malloc()`, `calloc()`, `realloc()`
- `malloc(n)`
  - Allocates a contiguous block of `n` **BYTES** of uninitialized memory.
  - Returns a pointer to the beginning of the allocated block; `NULL` if the request failed.
  - Different blocks not necessarily adjacent

Using malloc

- Almost always used for arrays or structs
- Good practice to use `sizeof` when allocating

```
int *p = malloc(n * sizeof(int));
```

  - Without the `sizeof` your code won't be very portable at all.
- Can use array or pointer syntax to access
- DON'T lose the original address
  - `p++` is a *terrible* idea if `p` was `malloc()`'d

Releasing Memory

- Release memory on the heap using `free()`
  - Memory is limited, should free when finished with it
- `free(p)`
  - Releases the whole block that `p` pointed to
  - `p` must point to the base of a `malloc()`'d block
  - Illegal to call `free()` on a block more than once

Dynamic Memory Example

- Need `#include <stdlib.h>`

```
typedef struct {
    int x;
    int y;
} point;
point *rect; /* 2 opposite corners = rectangle */
...
rect = malloc(2*sizeof(point));
/* Check malloc */
if (!rect) {
    printf("Out of memory!\n");
    exit(1);
}

/* Do NOT change rect in this region */
...
...
free(rect);
```

Question: We want the output `a[] = {0,1,2}` with no errors. Which lines do we need to change?

```
1  #define N 3
2  int *makeArray(int n) {
3      int *arr;
4      ar = (int *) malloc(n);
5      return arr;
6  }
7  int main() {
8      int i, *a = makeArray(N);
9      for (i=0; i<N; i++)
10         *a++ = i;
11     printf("a[] = {%d,%d,%d}",a[0],a[1],a[2]);
12     free(a);
13 }
```

(blue) 4, 12
(green) 5, 12
(purple) 4,10
(yellow) 5, 10

Question: We want the output `a[] = {0,1,2}` with no errors. Which lines do we need to change?

```
1   #define N 3
2   int *makeArray(int n) {
3       int *arr;
4       ar = (int *) malloc(n);
5       return arr;
6   }
7   int main() {
8       int i, *a = makeArray(N);
9       for (i=0; i<N; i++)
10          *a++ = i;
11      printf("a[] = {%d,%d,%d}",a[0],a[1],a[2]);
12      free(a);
13  }
```

(blue) 4, 12
(green) 5, 12
(purple) 4,10
(yellow) 5, 10

Know Your Memory Errors[1]

- Segmentation Fault (more common in 61c)
  "An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump"
- Bus error (less common in 61c)
  "A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error."

[1] Definitions from `http://www.hyperdictionary.com`

Common Problems

- Using uninitialized values
- Using memory that you don't own
  - Using `NULL` or garbage data as a pointer
  - De-allocated stack or heap variable
  - Out of bounds reference to stack or heap array
- Using memory you haven't allocated
- Freeing invalid memory
- Memory leaks

Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *p) {
    int j;
    *p = j;
}

void bar() {
    int i = 10;
    foo(&i);
    printf("i = %d\n", i);
}
```

Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *p) {
    int j;
    *p = j; // j is garbage
}

void bar() {
    int i = 10;
    foo(&i); // i now contains garbage
    printf("i = %d\n", i); // printing garbage
}
```

Memory You Don't Own I

- What is wrong with this code?

```
typedef struct node {
    struct node *next;
    int val;
} node;

int findLastNodeValue(node *head) {
    while (head->next)
        head = head->next;
    return head->val;
}
```

Memory You Don't Own I

- What is wrong with this code?

```
typedef struct node {
    struct node *next;
    int val;
} node;

// What if head is NULL?
int findLastNodeValue(node *head) {
    //Segfault here!
    while (head->next)
        head = head->next;
    return head->val;
}
```

Memory You Don't Own II

- What is wrong with this code?

```
char *append(const char *s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[MAXSIZE];
    int i = 0, j = 0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++)
        result[i] = s1[j];
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++)
        result[i] = s2[j];
    result[i] = '\0';
    return result;
}
```

Memory You Don't Own II

- What is wrong with this code?

```
char *append(const char *s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[MAXSIZE]; // local array is on stack
    int i = 0, j = 0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++)
        result[i] = s1[j];
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++)
        result[i] = s2[j];
    result[i] = '\0';
    //return value no longer valid after we return!
    return result;
}
```

Memory You Don't Own III

- What is wrong with this code?

```
typedef struct {
    char *name;
    int age;
} profile;

profile *person = malloc(sizeof(profile));
char *name = getName();
person->name = malloc(sizeof(char) * strlen(name));
strcpy(person->name, name);
... /* Do non-buggy stuff */
free(person);
free(person->name);
```

Memory You Don't Own III

- What is wrong with this code?

```
typedef struct {
    char *name;
    int age;
} profile;

profile *person = malloc(sizeof(profile));
char *name = getName();
// No space for the null terminator
person->name = malloc(sizeof(char) * strlen(name));
strcpy(person->name, name);
...
free(person);
// Oops, person was just deallocated, should
// have done this first
free(person->name);
```

Memory You Haven't Allocated I

- What is wrong with this code?

```
void str_manip() {
    const char *name = "Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0';
    printf("%s\n", str);
}
```

Memory You Haven't Allocated I

- What is wrong with this code?

```
void str_manip() {
    const char *name = "Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0'; // Out of bounds write
    printf("%s\n", str); // Out of bounds read
}
```

Memory You Haven't Allocated II

- What is wrong with this code?

```
char buffer[1024];
int main(int argc, char *argv[]) {
    strcpy(buffer, argv[1]);
    ...
}
```

Memory You Haven't Allocated II

- What is wrong with this code?

```
char buffer[1024];
int main(int argc, char *argv[]) {
    //What if strlen(argv[1]) > 1023?
    strcpy(buffer, argv[1]);
    ...
}
```

Freeing Invalid Memory

- What is wrong with this code?

```
void free_memX() {
    int fnh = 0;
    free(&fnh);
}

void free_memY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum + 1);
    free(fum);
    free(fum);
}
```

Freeing Invalid Memory

- What is wrong with this code?

```
void free_memX() {
    int fnh = 0;
    free(&fnh); // Not heap allocated
}

void free_memY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum + 1); // Does not point to start of block
    free(fum);
    free(fum); // Double-free
}
```

Memory Leaks I

- What is wrong with this code?

```
int *pi;
void foo() {
    pi = malloc(8 * sizeof(int));
    ...
    free(pi);
}

void main() {
    pi = malloc(4*sizeof(int));
    foo();
}
```

Memory Leaks I

- What is wrong with this code?

```
int *pi;
void foo() {
    // Overwrites old pointer
    // 4*sizeof(int) bytes from main leaked
    pi = malloc(8 * sizeof(int));
    ...
    free(pi);
}

void main() {
    pi = malloc(4*sizeof(int));
    foo();
}
```

Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: collects information on memory management while program runs
  - Doesn't work to find ALL memory bugs (this is an incredibly challenging problem), but will detect leaks for you

- You'll be using valgrind in lab 4, and on your project to check for memory leaks.

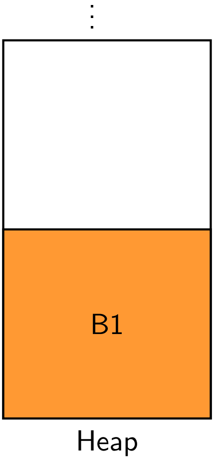
Technology Break

Memory Management

- Many calls to `malloc()` and `free()` with many different size blocks – where are they placed?
- Want system to be fast with minimal memory overhead
  - In contrast to an automatic garbage collection system, like in Java or Python
- Want to avoid *fragmentation*, the tendency of available memory to get separated into small chunks

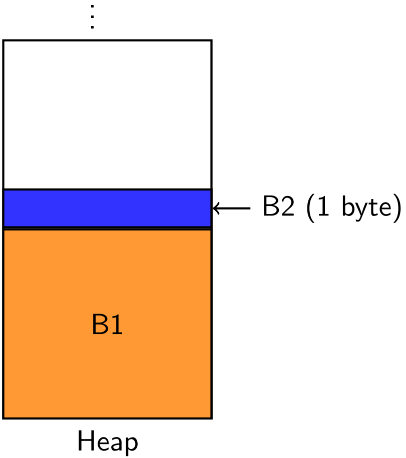
Fragmentation Example

1. Block 1: `malloc(100)`



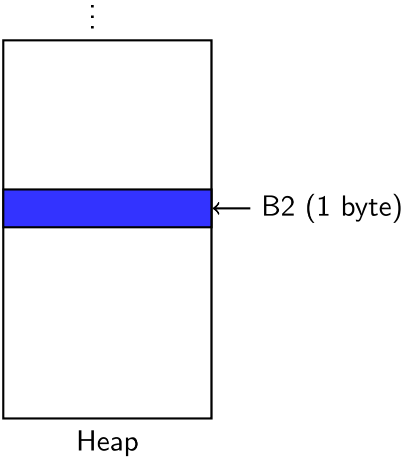
Fragmentation Example

1. Block 1: `malloc(100)`
2. Block 2: `malloc(1)`



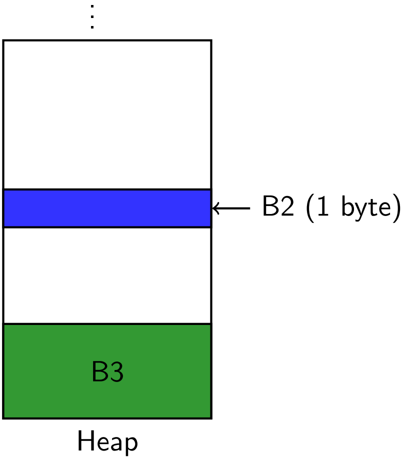
Fragmentation Example

1. Block 1: `malloc(100)`
2. Block 2: `malloc(1)`
3. Block 1: `free()`



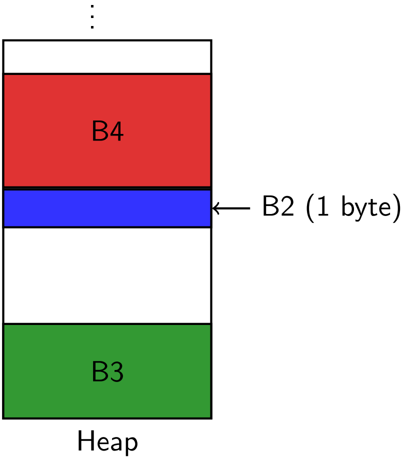
Fragmentation Example

1. Block 1: `malloc(100)`
2. Block 2: `malloc(1)`
3. Block 1: `free()`
4. Block 3: `malloc(50)`
   - Note, could go above B2



Fragmentation Example

1. Block 1: `malloc(100)`
2. Block 2: `malloc(1)`
3. Block 1: `free()`
4. Block 3: `malloc(50)`
   - Note, could go above B2
5. Block 4: `malloc(60)`



Basic Allocation Strategy: K&R

- Section 8.7 offers an implementation of memory managment (linked list of free blocks)
- This is just one of many possible memory management algorithms
  - Just to give you a taste
  - No single best approach for every application

K&R Implementation

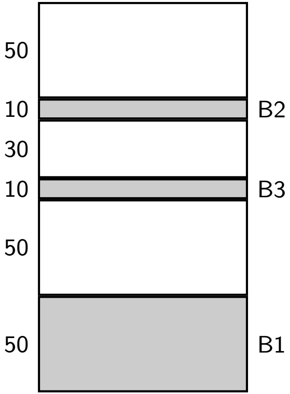
- Each block holds its own size and a pointer to the next block
- `free()` adds block to the list, combines with adjacent free blocks
- `malloc()` searches free list for block large enough to meet request
  - If multiple blocks fit request, which one do we use?

Choosing a Block

- **Best-fit:** Choose smallest block that fits request
  - Tries to limit wasted fragmentation space, but takes more time and leaves a lot of small blocks
- **First-fit:** Choose first block that is large enough (always starts from the beginning)
  - Fast, but tends to concentrate small blocks at the beginning
- **Next-fit:** Like first-fit, but resume search from where we last left off
  - Fast, and does not concentrate small blocks at front

Question: Which allocation system and set of requests will create a contiguous allocated region in the Heap? B3 was the last fulfilled request.

(blue) Best-fit:`malloc(50)`, `malloc(50)`
(green) First-fit:`malloc(50)`, `malloc(30)`
(purple) Next-fit:`malloc(30)`, `malloc(50)`
(yellow) Next-fit:`malloc(50)`, `malloc(30)`



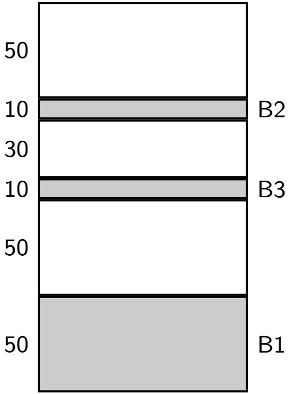
Question: Which allocation system and set of requests will create a contiguous allocated region in the Heap? B3 was the last fulfilled request.

(blue) Best-fit:`malloc(50)`, `malloc(50)`

(green) First-fit:`malloc(50)`, `malloc(30)`

(purple) Next-fit:`malloc(30)`, `malloc(50)`

(yellow) Next-fit:`malloc(50)`, `malloc(30)`



Outline

Linked List Example

- We want to generate a linked list of strings
  - This example uses structs, pointers, `malloc()`, and `free()`
- First, we'll need a structure for list nodes

```
typedef struct node {
    char *value;
    struct node *next;
} node;
```

Adding a node to the list

```
char *s1 = "start", *s2 = "middle";
char *s3 = "end";
node *list = NULL;

/* Creates the list {"start, "middle", "end"} */
list = prepend(s3, list);
list = prepend(s2, list);
list = prepend(s1, list);
```

Adding a node ot the list

- Let's examine the 3rd call ("start"):

```
node *prepend(char *s, node *lst) {
    node *node = malloc(sizeof(node));
    node->value = malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = lst;
    return node;
}
```

Adding a node ot the list

- Let's examine the 3rd call ("start"):

```
node *prepend(char *s, node *lst) {
    node *node = malloc(sizeof(node));
    node->value = malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = lst;
    return node;
}
```

- Boardwork!

Removing a node

- Now let's remove "start" from the list:

```
node *del_front(node *lst) {
    node *tmp = lst->next;
    free(lst->value);
    free(lst);
    return tmp;
}
```

Removing a node

- Now let's remove "start" from the list:

```
node *del_front(node *lst) {
    node *tmp = lst->next;
    free(lst->value);
    free(lst);
    return tmp;
}
```

- Boardwork!

Additional Functionality

- How might you implement the following (left as exercises to the reader):
  - Append node to end of a list
  - Delete/free an entire list
  - Join two lists together
  - Sort a list

Summary

- C memory layout
  - **Static Data:** globals and string literals
  - **Code:** copy of machine code
  - **Stack:** local variables
  - **Heap:** dynamic storage via `malloc()` and `free()`
- Memory management
  - Want fast, with minimal fragmentation